

Condor[®] Version 7.2.3 Manual

Condor Team, University of Wisconsin–Madison

May 28, 2009

CONTENTS

1	Overview	1
1.1	High-Throughput Computing (HTC) and its Requirements	1
1.2	Condor's Power	2
1.3	Exceptional Features	3
1.4	Current Limitations	4
1.5	Availability	5
1.6	Contributions to Condor	8
1.7	Contact Information	9
1.8	Privacy Notice	10
2	Users' Manual	11
2.1	Welcome to Condor	11
2.2	Introduction	11
2.3	Matchmaking with ClassAds	12
2.3.1	Inspecting Machine ClassAds with condor_status	13
2.4	Road-map for Running Jobs	14
2.4.1	Choosing a Condor Universe	15
2.5	Submitting a Job	19
2.5.1	Sample submit description files	20

2.5.2	About Requirements and Rank	22
2.5.3	Submitting Jobs Using a Shared File System	24
2.5.4	Submitting Jobs Without a Shared File System: Condor's File Transfer Mechanism	26
2.5.5	Environment Variables	33
2.5.6	Heterogeneous Submit: Execution on Differing Architectures	34
2.6	Managing a Job	38
2.6.1	Checking on the progress of jobs	38
2.6.2	Removing a job from the queue	40
2.6.3	Placing a job on hold	40
2.6.4	Changing the priority of jobs	41
2.6.5	Why does the job not run?	41
2.6.6	In the log file	42
2.6.7	Job Completion	45
2.7	Priorities and Preemption	47
2.7.1	Job Priority	47
2.7.2	User priority	47
2.7.3	Details About How Condor Jobs Vacate Machines	48
2.8	Java Applications	48
2.8.1	A Simple Example Java Application	49
2.8.2	Less Simple Java Specifications	51
2.8.3	Chirp I/O	53
2.9	Parallel Applications (Including MPI Applications)	55
2.9.1	Prerequisites to Running Parallel Jobs	55
2.9.2	Parallel Job Submission	55
2.9.3	Parallel Jobs with Separate Requirements	57
2.9.4	MPI Applications Within Condor's Parallel Universe	58
2.9.5	Outdated Documentation of the MPI Universe	59
2.10	DAGMan Applications	64

2.10.1	DAGMan Terminology	64
2.10.2	Input File Describing the DAG	65
2.10.3	Submit Description File	69
2.10.4	Job Submission	70
2.10.5	Job Monitoring, Job Failure, and Job Removal	71
2.10.6	Advanced Features of DAGMan	72
2.10.7	Job Recovery: The Rescue DAG	86
2.10.8	File Paths in DAGs	89
2.10.9	Visualizing DAGs with <i>dot</i>	90
2.11	Virtual Machine Applications	91
2.11.1	The Submit Description File	91
2.11.2	Checkpoints	94
2.11.3	Disk Images	94
2.11.4	Job Completion in the vm Universe	94
2.12	Time Scheduling for Job Execution	95
2.12.1	Job Deferral	95
2.12.2	CronTab Scheduling	98
2.13	Stork Applications	102
2.13.1	Submitting Stork Jobs	102
2.13.2	Managing Stork Jobs	104
2.13.3	Fault Tolerance	104
2.13.4	Running Stork Jobs Under DAGMan	106
2.13.5	The Lease Manager	106
2.14	Job Monitor	107
2.14.1	Transition States	107
2.14.2	Events	107
2.14.3	Selecting Jobs	107
2.14.4	Zooming	108

2.14.5	Keyboard and Mouse Shortcuts	108
2.15	Special Environment Considerations	108
2.15.1	AFS	108
2.15.2	NFS Automounter	109
2.15.3	Condor Daemons That Do Not Run as root	109
2.15.4	Job Leases	110
2.16	Potential Problems	111
2.16.1	Renaming of argv[0]	111
3	Administrators' Manual	112
3.1	Introduction	112
3.1.1	The Different Roles a Machine Can Play	113
3.1.2	The Condor Daemons	114
3.2	Installation	116
3.2.1	Obtaining Condor	117
3.2.2	Preparation	118
3.2.3	Newer Unix Installation Procedure	123
3.2.4	Condor is installed Under Unix ... now what?	125
3.2.5	Installation on Windows	127
3.2.6	RPMs	137
3.2.7	Upgrading - Installing a Newer Version of Condor	137
3.2.8	Installing the CondorView Client Contrib Module	138
3.2.9	Dynamic Deployment	140
3.3	Configuration	142
3.3.1	Introduction to Configuration Files	142
3.3.2	The Special Configuration Macros \$ENV(), \$RANDOM_CHOICE(), and \$RANDOM_INTEGER()149	
3.3.3	Condor-wide Configuration File Entries	150
3.3.4	Daemon Logging Configuration File Entries	157

3.3.5	DaemonCore Configuration File Entries	162
3.3.6	Network-Related Configuration File Entries	165
3.3.7	Shared File System Configuration File Macros	168
3.3.8	Checkpoint Server Configuration File Macros	172
3.3.9	condor_master Configuration File Macros	173
3.3.10	condor_startd Configuration File Macros	179
3.3.11	condor_schedd Configuration File Entries	195
3.3.12	condor_shadow Configuration File Entries	202
3.3.13	condor_starter Configuration File Entries	203
3.3.14	condor_submit Configuration File Entries	205
3.3.15	condor_preen Configuration File Entries	207
3.3.16	condor_collector Configuration File Entries	208
3.3.17	condor_negotiator Configuration File Entries	211
3.3.18	condor_procd Configuration File Macros	216
3.3.19	condor_credd Configuration File Macros	216
3.3.20	condor_gridmanager Configuration File Entries	216
3.3.21	condor_job_router Configuration File Entries	220
3.3.22	condor_lease_manager Configuration File Entries	221
3.3.23	grid_monitor Configuration File Entries	222
3.3.24	Configuration File Entries Relating to Grid Usage and Glidein	223
3.3.25	Configuration File Entries for DAGMan	223
3.3.26	Configuration File Entries Relating to Security	229
3.3.27	Configuration File Entries Relating to PrivSep	232
3.3.28	Configuration File Entries Relating to Virtual Machines	233
3.3.29	Configuration File Entries Relating to High Availability	235
3.3.30	Configuration File Entries Relating to Quill	239
3.3.31	MyProxy Configuration File Macros	242
3.3.32	Configuration File Macros Affecting APIs	242

3.3.33	Stork Configuration File Macros	243
3.4	User Priorities and Negotiation	244
3.4.1	Real User Priority (RUP)	244
3.4.2	Effective User Priority (EUP)	245
3.4.3	Priorities and Preemption	245
3.4.4	Priority Calculation	247
3.4.5	Negotiation	247
3.4.6	The Layperson's Description of the Pie Spin and Pie Slice	248
3.4.7	Group Accounting	249
3.4.8	Group Quotas	250
3.5	Policy Configuration for the <i>condor_startd</i>	252
3.5.1	Startd ClassAd Attributes	253
3.5.2	The START expression	253
3.5.3	The IS_VALID_CHECKPOINT_PLATFORM expression	254
3.5.4	The RANK expression	255
3.5.5	Machine States	256
3.5.6	Machine Activities	260
3.5.7	State and Activity Transitions	261
3.5.8	State/Activity Transition Expression Summary	270
3.5.9	Policy Settings	272
3.6	Security	281
3.6.1	Condor's Security Model	282
3.6.2	Security Negotiation	284
3.6.3	Authentication	287
3.6.4	The Unified Map File for Authentication	298
3.6.5	Encryption	299
3.6.6	Integrity	301
3.6.7	Authorization	302

3.6.8	Security Sessions	306
3.6.9	Host-Based Security in Condor	307
3.6.10	Using Condor w/ Firewalls, Private Networks, and NATs	315
3.6.11	User Accounts in Condor	315
3.6.12	Privilege Separation	320
3.6.13	Support for <i>glexec</i>	324
3.7	Networking (includes sections on Port Usage and GCB)	325
3.7.1	Port Usage in Condor	325
3.7.2	Configuring Condor for Machines With Multiple Network Interfaces	329
3.7.3	Generic Connection Brokering (GCB)	332
3.7.4	Using TCP to Send Updates to the <i>condor_collector</i>	345
3.8	The Checkpoint Server	346
3.8.1	Preparing to Install a Checkpoint Server	347
3.8.2	Installing the Checkpoint Server Module	347
3.8.3	Configuring your Pool to Use Multiple Checkpoint Servers	348
3.8.4	Checkpoint Server Domains	349
3.9	DaemonCore	351
3.9.1	DaemonCore and Unix signals	352
3.9.2	DaemonCore and Command-line Arguments	352
3.10	The High Availability of Daemons	354
3.10.1	High Availability of the Job Queue	354
3.10.2	High Availability of the Central Manager	356
3.11	Quill	362
3.11.1	Installation and Configuration	362
3.11.2	Four Usage Examples	367
3.11.3	Quill and Security	368
3.11.4	Quill and Its RDBMS Schema	369
3.12	Setting Up for Special Environments	389

3.12.1	Using Condor with AFS	389
3.12.2	Configuring Condor for Multiple Platforms	391
3.12.3	Full Installation of condor_compile	394
3.12.4	The <i>condor_kbdd</i>	395
3.12.5	Configuring The CondorView Server	396
3.12.6	Running Condor Jobs within a VMware or Xen Virtual Machine Environment	398
3.12.7	Configuring The Startd for SMP Machines	399
3.12.8	Condor's Dedicated Scheduling	408
3.12.9	Configuring Condor for Running Backfill Jobs	412
3.12.10	Group ID-Based Process Tracking	420
3.12.11	Concurrency Limits	420
3.13	Java Support Installation	422
3.14	Virtual Machines	424
3.14.1	Configuration Parameters	424
3.15	Power Management	426
3.15.1	Entering a Low Power State	426
3.15.2	Returning From a Low Power State	427
3.15.3	Keeping a ClassAd for a Hibernating Machine	428
3.15.4	Linux Platform Details	428
3.15.5	Windows Platform Details	428
4	Miscellaneous Concepts	430
4.1	Condor's ClassAd Mechanism	430
4.1.1	Syntax	431
4.1.2	Evaluation Semantics	439
4.1.3	ClassAds in the Condor System	441
4.2	Condor's Checkpoint Mechanism	443
4.2.1	Standalone Checkpointing	444

4.2.2	Checkpoint Safety	445
4.2.3	Checkpoint Warnings	445
4.2.4	Checkpoint Library Interface	446
4.3	Computing On Demand (COD)	447
4.3.1	Overview of How COD Works	448
4.3.2	Authorizing Users to Create and Manage COD Claims	448
4.3.3	Defining a COD Application	448
4.3.4	Managing COD Resource Claims	452
4.3.5	Limitations of COD Support in Condor	459
4.4	Job Hooks	460
4.4.1	Hooks that Fetch Work	460
4.4.2	Hooks for a Job Router	467
4.5	Application Program Interfaces	468
4.5.1	Web Service	468
4.5.2	The DRMAA API	480
4.5.3	The Command Line Interface	482
4.5.4	The Condor GAHP	482
4.5.5	The Condor Perl Module	482
5	Grid Computing	491
5.1	Introduction	491
5.2	Connecting Condor Pools with Flocking	492
5.2.1	Flocking Configuration	492
5.2.2	Job Considerations	494
5.3	The Grid Universe	494
5.3.1	Condor-C, The condor Grid Type	494
5.3.2	Condor-G, the gt2 and gt4 Grid Types	498
5.3.3	The nordugrid Grid Type	510

5.3.4	The uncore Grid Type	510
5.3.5	The pbs Grid Type	511
5.3.6	The lsf Grid Type	511
5.3.7	The amazon Grid Type	512
5.3.8	Matchmaking in the Grid Universe	513
5.4	Glidein	519
5.4.1	What <i>condor_glidein</i> Does	519
5.4.2	Configuration Requirements in the Local Pool	519
5.4.3	Running Jobs on the Remote Grid Resource After Glidein	520
5.5	Dynamic Deployment	520
5.6	The Condor Job Router	521
5.6.1	Routing Mechanism	522
5.6.2	Job Submission with Job Routing Capability	522
5.6.3	An Example Configuration	524
5.6.4	Routing Table Entry ClassAd Attributes	526
5.6.5	Example: constructing the routing table from ReSS	528
6	Platform-Specific Information	529
6.1	Linux	529
6.1.1	Linux Kernel-specific Information	530
6.1.2	Red Hat Version 9.x	530
6.1.3	Red Hat Fedora 1, 2, and 3	530
6.2	Microsoft Windows	531
6.2.1	Limitations under Windows	531
6.2.2	Supported Features under Windows	532
6.2.3	Secure Password Storage	532
6.2.4	Executing Jobs as the Submitting User	533
6.2.5	Executing Jobs with the User's Profile Loaded	534

6.2.6	Details on how Condor for Windows starts/stops a job	535
6.2.7	Security Considerations in Condor for Windows	537
6.2.8	Network files and Condor	538
6.2.9	Interoperability between Condor for Unix and Condor for Windows	544
6.2.10	Some differences between Condor for Unix -vs- Condor for Windows . . .	544
6.3	Macintosh OS X	545
6.4	AIX	545
6.4.1	AIX 5.2L	545
6.4.2	AIX 5.1L	545
7	Frequently Asked Questions (FAQ)	547
7.1	Obtaining & Installing Condor	547
7.2	Setting up Condor	553
7.3	Running Condor Jobs	556
7.4	Condor on Windows	566
7.5	Grid Computing	573
7.6	Troubleshooting	575
7.7	Other questions	578
8	Version History and Release Notes	579
8.1	Introduction to Condor Versions	579
8.1.1	Condor Version Number Scheme	579
8.1.2	The Stable Release Series	580
8.1.3	The Development Release Series	580
8.2	Upgrade Surprises	580
8.3	Stable Release Series 7.2	581
8.4	Development Release Series 7.1	592
8.5	Stable Release Series 7.0	606
8.6	Development Release Series 6.9	623

8.7 Stable Release Series 6.8	648
9 Command Reference Manual (man pages)	672
<i>cleanup_release</i>	673
<i>condor_advertise</i>	675
<i>condor_check_userlogs</i>	679
<i>condor_checkpoint</i>	680
<i>condor_chirp</i>	683
<i>condor_cod</i>	686
<i>condor_cold_start</i>	689
<i>condor_cold_stop</i>	692
<i>condor_compile</i>	695
<i>condor_config_bind</i>	698
<i>condor_config_val</i>	700
<i>condor_configure</i>	704
<i>condor_convert_history</i>	709
<i>condor_dagman</i>	711
<i>condor_fetchlog</i>	715
<i>condor_findhost</i>	718
<i>condor_glidein</i>	720
<i>condor_history</i>	727
<i>condor_hold</i>	730
<i>condor_load_history</i>	733
<i>condor_master</i>	735
<i>condor_master_off</i>	737
<i>condor_off</i>	738
<i>condor_on</i>	741
<i>condor_power</i>	744

<i>condor_preen</i>	746
<i>condor_prio</i>	748
<i>condor_q</i>	750
<i>condor_qedit</i>	758
<i>condor_reconfig</i>	760
<i>condor_reconfig_schedd</i>	763
<i>condor_release</i>	764
<i>condor_reschedule</i>	766
<i>condor_restart</i>	769
<i>condor_rm</i>	772
<i>condor_router_history</i>	775
<i>condor_run</i>	777
<i>condor_set_shutdown</i>	781
<i>condor_stats</i>	783
<i>condor_status</i>	787
<i>condor_store_cred</i>	793
<i>condor_submit</i>	795
<i>condor_submit_dag</i>	825
<i>condor_transfer_data</i>	830
<i>condor_updates_stats</i>	832
<i>condor_userlog</i>	835
<i>condor_userprio</i>	838
<i>condor_vacate</i>	841
<i>condor_vacate_job</i>	844
<i>condor_version</i>	847
<i>condor_wait</i>	849
<i>filelock_midwife</i>	852
<i>filelock_undertaker</i>	854

<i>install_release</i>	856
<i>stork_q</i>	858
<i>stork_list_cred</i>	860
<i>stork_rm</i>	862
<i>stork_rm_cred</i>	864
<i>stork_store_cred</i>	866
<i>stork_status</i>	868
<i>stork_submit</i>	870
<i>uniq_pid_midwife</i>	874
<i>uniq_pid_undertaker</i>	876

LICENSING AND COPYRIGHT

Condor is released under the Apache License, Version 2.0.

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, WI.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or

otherwise, or (ii) ownership of fifty percent (50) outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. **Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

1.1 High-Throughput Computing (HTC) and its Requirements

For many research and engineering projects, the quality of the research or the product is heavily dependent upon the quantity of computing cycles available. It is not uncommon to find problems that require weeks or months of computation to solve. Scientists and engineers engaged in this sort of work need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High-Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of Floating point Operations Per Second (FLOPS). A growing community is not concerned about operations per second, but operations per month or per year. Their problems are of a much larger scale. They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can complete.

The key to HTC is to efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on a large, centralized mainframe or a supercomputer to do computational work. A large number of individuals and groups needed to pool their financial resources to afford such a machine. Users had to wait for their turn on the mainframe, and they had a limited amount of time allocated. While this environment was inconvenient for users, the utilization of the mainframe was high; it was busy nearly all the time.

As computers became smaller, faster, and cheaper, users moved away from centralized mainframes and purchased personal desktop workstations and PCs. An individual or small group could afford a computing resource that was available whenever they wanted it. The personal computer is slower than the large centralized machine, but it provides exclusive access. Now, instead of one giant computer for a large institution, there may be hundreds or thousands of personal computers. This

is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole may rise dramatically as the result of such a change, but because of distributed ownership, individuals have not been able to capitalize on the institutional growth of computing power. And, while distributed ownership is more convenient for the users, the utilization of the computing power is lower. Many personal desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as being away at lunch, in meetings, or at home sleeping).

1.2 Condor's Power

Condor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively utilizes the computing power of workstations that communicate over a network. Condor can manage a dedicated cluster of workstations. Its power comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership.

A user submits the job to Condor. Condor finds an available machine on the network and begins running the job on that machine. Condor has the capability to detect that a machine running a Condor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. Condor continues job on the new machine from precisely where it left off.

In those cases where Condor can checkpoint and migrate a job, Condor makes it easy to maximize the number of machines which can run a job. In this case, there is no requirement for machines to share file systems (for example, with NFS or AFS), so that machines across an entire enterprise can run a job, including machines in different administrative domains.

Condor can be a real time saver when a job must be run many (hundreds of) different times, perhaps with hundreds of different data sets. With one command, all of the hundreds of jobs are submitted to Condor. Depending upon the number of machines in the Condor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment.

Condor does not require an account (login) on machines where it runs a job. Condor can do this because of its *remote system call* technology, which traps library calls for such operations as reading or writing from disk files. The calls are transmitted over the network to be performed on the machine where the job was submitted.

Condor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. Condor implements *ClassAds*, a clean design that simplifies the user's submission of jobs.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the Condor pool advertise their resource properties, both static and dynamic, such as

available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a *resource offer* ad. A user specifies a *resource request* ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. Condor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.

1.3 Exceptional Features

Checkpoint and Migration. Where programs can be linked with Condor libraries, users of Condor may be assured that their jobs will eventually complete, even in the ever changing environment that Condor utilizes. As a machine running a job submitted to Condor becomes unavailable, the job can be check pointed. The job may continue after migrating to another machine. Condor's periodic checkpoint feature periodically checkpoints a job even in lieu of migration in order to safeguard the accumulated computation time on a job from being lost in the event of a system failure such as the machine being shutdown or a crash.

Remote System Calls. Despite running jobs on remote machines, the Condor standard universe execution mode preserves the local execution environment via remote system calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

No Changes Necessary to User's Source Code. No special programming is required to use Condor. Condor is able to run non-interactive programs. The checkpoint and migration of programs by Condor is transparent and automatic, as is the use of remote system calls. If these facilities are desired, the user only re-links the program. The code is neither recompiled nor changed.

Pools of Machines can be Hooked Together. Flocking is a feature of Condor that allows jobs submitted within a first pool of Condor machines to execute on a second pool. The mechanism is flexible, following requests from the job submission, while allowing the second pool, or a subset of machines within the second pool to set policies over the conditions under which jobs are executed.

Jobs can be Ordered. The ordering of job execution required by dependencies among jobs in a set is easily handled. The set of jobs is specified using a directed acyclic graph, where each job is a node in the graph. Jobs are submitted to Condor following the dependencies given by the graph.

Condor Enables Grid Computing. As grid computing becomes a reality, Condor is already there. The technique of glidein allows jobs submitted to Condor to be executed on grid machines

in various locations worldwide. As the details of grid computing evolve, so does Condor's ability, starting with Globus-controlled resources.

Sensitive to the Desires of Machine Owners. The owner of a machine has complete priority over the use of the machine. An owner is generally happy to let others compute on the machine while it is idle, but wants it back promptly upon returning. The owner does not want to take special action to regain control. Condor handles this automatically.

ClassAds. The ClassAd mechanism in Condor provides an extremely flexible, expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that a job run on a machine with 64 Mbytes of RAM, but state a preference for 128 Mbytes, if available. A workstation owner can state a preference that the workstation runs jobs from a specified set of users. The owner can also require that there be no interactive workstation activity detectable at certain hours before Condor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

1.4 Current Limitations

Limitations on Jobs which can Checkpointed Although Condor can schedule and run any type of process, Condor does have some limitations on jobs that it can transparently checkpoint and migrate:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.

9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.
11. On Linux, your job must be statically linked. *condor_compile* does this by default. Dynamic linking is allowed on Solaris.
12. Reading to or writing from files larger than 2 GB is not supported.

Note: these limitations *only* apply to jobs which Condor has been asked to transparently checkpoint. If job checkpointing is not desired, the limitations above do not apply.

Security Implications. Condor does a significant amount of work to prevent security hazards, but loopholes are known to exist. Condor can be instructed to run user programs only as the UNIX user nobody, a user login which traditionally has very restricted access. But even with access solely as user nobody, a sufficiently malicious individual could do such things as fill up `/tmp` (which is world writable) and/or gain read access to world readable files. Furthermore, where the security of machines in the pool is a high concern, only machines where the UNIX user root on that machine can be trusted should be admitted into the pool. Condor provides the administrator with extensive security mechanisms to enforce desired policies.

Jobs Need to be Re-linked to get Checkpointing and Remote System Calls Although typically no source code changes are required, Condor requires that the jobs be re-linked with the Condor libraries to take advantage of checkpointing and remote system calls. This often precludes commercial software binaries from taking advantage of these services because commercial packages rarely make their object code available. Condor's other services are still available for these commercial packages.

1.5 Availability

Condor is currently available as a free download from the Internet via the World Wide Web at URL <http://www.cs.wisc.edu/condor/downloads-v2>. Binary distributions of Condor are available for the platforms detailed in Table 1.1. A platform is an architecture/operating system combination. Condor binaries are available for most major versions of Unix, as well as Windows.

In the table, *clipped* means that Condor does not support checkpointing or remote system calls on the given platform. This means that *standard* jobs are not supported, only *vanilla* jobs. See section 2.4.1 on page 15 for more details on job universes within Condor and their abilities and limitations.

For 7.0.0 and later releases, the Condor source code is available for public download alongside the binary distributions.

<i>Architecture</i>	<i>Operating System</i>
Hewlett Packard PA-RISC (both PA7000 and PA8000 series)	- HPUNIX 11.00 (clipped)
Sun SPARC Sun4m, Sun4c, Sun Ultra-SPARC	- Solaris 8, 9 (clipped) - Solaris 10 (clipped) (Using the Solaris 9 binaries)
Intel x86	- Red Hat Linux 9 - RedHat Enterprise Linux 3 - RedHat Enterprise Linux 4 (Using RHEL3 binaries) - RedHat Enterprise Linux 5 - Fedora Core 1, 2, 3, 4, 5 (Using RHEL3 binaries) - Debian Linux 3.1 (sarge) (Using RHEL3 binaries) - Debian Linux 4.0 (etch) - Debian Linux 5.0 (lenny) - Windows 2000 Professional and Server (Win NT 5.0) (clipped) - Windows 2003 Server (Win NT 5.2) (clipped) - Windows 2008 Server (Win NT 6.0) (clipped) - Windows XP Professional (Win NT 5.1) (clipped) - Windows Vista (Win NT 6.0) (clipped) - Macintosh OS X 10.4 (clipped)
PowerPC	- Macintosh OS X 10.4 (clipped) - AIX 5.2, 5.3 (clipped) - Yellowdog Linux 5.0 (clipped) - SuSE Linux Enterprise Server 9 (clipped)
Itanium IA64	- Red Hat Enterprise Linux 3 (clipped)
Opteron x86_64	- Red Hat Enterprise Linux 3 - Red Hat Enterprise Linux 5 - Debian Linux 5.0 (lenny)

Table 1.1: Condor Version 7.2.3 supported platforms

NOTE: Other Linux distributions likely work, but are not tested or supported.

Condor is also available, but is not currently distributed as tested binaries for the platforms shown in Table 1.2.

For more platform-specific information about Condor's support for various operating systems,

<i>Platform</i>	<i>Notes</i>
FreeBSD 6, 7 (clipped) on Intel x86	Known to compile
FreeBSD 7 (clipped) on Itanium IA64	Known to compile

Table 1.2: Other Condor Version 7.2.3 available platforms

see Chapter 6 on page 529.

Jobs submitted to the standard universe utilize *condor_compile* to relink programs with libraries provided by Condor. Table 1.3 lists supported compilers by platform. Other compilers may work, but are not supported.

Platform	Compiler	Notes
Red Hat Enterprise Linux 3, 4, 5 on x86	gcc, g++, and g77	as shipped
Red Hat Debian Linux 3.1 (sarge) on x86	gcc up to version 3.4.1	
Red Hat Debian Linux 5.0 (lenny) on x86 and x86_64	gcc, g++, gfortran as shipped	
Fedora Core 1, 2, 3, 4, 5, 6, 7 on x86	gcc, g++, and g77	as shipped

Table 1.3: Supported compilers under Condor Version 7.2.3

The following table, Table 1.4, identifies which platforms support the transfer of large files (greater than 2 Gbyte in length). For vanilla universe jobs and those platforms where large file transfer is supported, the support is automatic.

<i>Platform</i>	<i>Large File Transfer Supported?</i>
Hewlett Packard PA-RISC with HPUNIX 11.00	Yes
Sun SPARC Sun4m, Sun4c, Sun UltraSPARC with Solaris 8, 9	Yes
Intel x86 with Red Hat Enterprise Linux 3, 4, 5, Debian Linux 3.1, 4.0, 5.0	Yes
Intel x86 with Fedora Core 1, 2, 3, 4, 5, 6, 7	Yes
Intel x86 with Windows 2000 Professional and Server	Yes
Intel x86 with 2003 Server (Win NT 5.0)	Yes
Intel x86 with Windows XP Professional (Win NT 5.1)	Yes
Intel x86 with Windows Vista	Yes
PowerPC with Macintosh OS X	No
PowerPC with AIX 5.2	Yes
PowerPC with Yellowdog Linux 5.0	Yes
Itanium with Red Hat Enterprise Linux 3	Yes
Opteron x86_64 with Red Hat Enterprise Linux 3, 4, 5, Debian Linux 5.0	Yes

Table 1.4: Supported platforms for large file transfer of vanilla universe job files

1.6 Contributions to Condor

The quality of the Condor project is enhanced by the contributions of external organizations. We gratefully acknowledge the following contributions.

- The Globus Alliance (<http://www.globus.org>), for code and assistance in developing Condor-G and the Grid Security Infrastructure (GSI) for authentication and authorization.
- The GOZAL Project from the Computer Science Department of the Technion Israel Institute of Technology (<http://www.technion.ac.il/>), for their enhancements for Condor's High Availability. The *condor_had* daemon allows one of multiple machines to function as the central manager for a Condor pool. Therefore, if an acting central manager fails, another can take its place.
- Micron Corporation (<http://www.micron.com/>) for the MSI-based installer for Condor on Windows.
- Paradyn Project (<http://www.paradyn.org/>) and the Universitat Autònoma de Barcelona (<http://www.caos.uab.es/>) for work on the Tool Daemon Protocol (TDP).

Our Web Services API acknowledges the use of gSOAP with their requested wording:

- Part of the software embedded in this product is gSOAP software. Portions created by gSOAP are Copyright (C) 2001-2004 Robert A. van Engelen, Genivia inc. All Rights Reserved.

THE SOFTWARE IN THIS PRODUCT WAS IN PART PROVIDED BY GENIVIA INC AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Some distributions of Condor include the Google Coredumper library (<http://goog-coredumper.sourceforge.net/>). The Google Coredumper library is released under these terms:

Copyright (c) 2005, Google Inc.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.7 Contact Information

The latest software releases, publications/papers regarding Condor and other High-Throughput Computing research can be found at the official web site for Condor at <http://www.cs.wisc.edu/condor>.

In addition, there is an e-mail list at condor-world@cs.wisc.edu. The Condor Team uses this e-mail list to announce new releases of Condor and other major Condor-related news items. To subscribe or unsubscribe from the the list, follow the instructions at <http://www.cs.wisc.edu/condor/mail-lists/>. Because many of us receive too much e-mail as it is, you will be happy to know that the Condor World e-mail list group is moderated, and only major announcements of wide interest are distributed.

Our users support each other by belonging to an unmoderated mailing list targeted at solving problems with Condor. Condor team members attempt to monitor traffic to Condor Users, responding as they can. Follow the instructions at <http://www.cs.wisc.edu/condor/mail-lists/>.

Finally, you can reach the Condor Team directly. The Condor Team is comprised of the developers and administrators of Condor at the University of Wisconsin-Madison. Condor questions, comments, pleas for help, and requests for commercial contract consultation or support are all welcome; send Internet e-mail to <mailto:condor-admin@cs.wisc.edu>. Please include your name, organization, and telephone number in your message. If you are having trouble with Condor, please help us troubleshoot by including as much pertinent information as you can, including snippets of Condor log files.

1.8 Privacy Notice

The Condor software periodically sends short messages to the Condor Project developers at the University of Wisconsin, reporting totals of machines and jobs in each running Condor system. An example of such a message is given below.

The Condor Project uses these collected reports to publish summary figures and tables, such as the total of Condor systems worldwide, or the geographic distribution of Condor systems. This information helps the Condor Project to understand the scale and composition of Condor in the real world and improve the software accordingly.

The Condor Project will not use these reports to publicly identify any Condor system or user without permission. The Condor software does not collect or report any personal information about individual users.

We hope that you will contribute to the development of Condor through this reporting feature. However, you are free to disable it at any time by changing the configuration variables `CONDOR_DEVELOPERS` and `CONDOR_DEVELOPERS_COLLECTOR`, both described in section 3.3.16 of this manual.

Example of data reported:

This is an automated email from the Condor system
on machine "your.condor.pool.com". Do not reply.

This Collector has the following IDs:

CondorVersion: 6.6.0 Nov 12 2003
CondorPlatform: INTEL-LINUX-GLIBC22

	Machines	Owner	Claimed	Unclaimed	Matched	Preempting
INTEL/LINUX	810	52	716	37	0	5
INTEL/WINNT50	120	5	115	0	0	0
SUN4u/SOLARIS28	114	12	92	9	0	1
SUN4x/SOLARIS28	5	1	0	4	0	0
Total	1049	70	923	50	0	6
RunningJobs			IdleJobs			
920			3868			

2.1 Welcome to Condor

Presenting Condor Version 7.2.3! Condor is developed by the Condor Team at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Computer Sciences department more than 10 years ago. This Condor pool has since served as a major source of computing cycles to UW faculty and students. For many, it has revolutionized the role computing plays in their research. An increase of one, and sometimes even two, orders of magnitude in the computing throughput of a research organization can have a profound impact on its size, complexity, and scope. Over the years, the Condor Team has established collaborations with scientists from around the world, and it has provided them with access to surplus cycles (one scientist has consumed 100 CPU years!). Today, our department's pool consists of more than 700 desktop Unix workstations and more than 100 Windows 2000 machines. On a typical day, our pool delivers more than 500 CPU days to UW researchers. Additional Condor pools have been established over the years across our campus and the world. Groups of researchers, engineers, and scientists have used Condor to establish compute pools ranging in size from a handful to hundreds of workstations. We hope that Condor will help revolutionize your compute environment as well.

2.2 Introduction

In a nutshell, Condor is a specialized batch system for managing compute-intensive jobs. Like most batch systems, Condor provides a queuing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to Condor, Condor puts the jobs in a queue, runs them, and then informs the user as to the result.

Batch systems normally operate only with dedicated machines. Often termed compute servers, these dedicated machines are typically owned by one organization and dedicated to the sole purpose of running compute jobs. Condor can schedule jobs on dedicated machines. But unlike traditional batch systems, Condor is also designed to effectively utilize non-dedicated machines to run jobs. By being told to only run compute jobs on machines which are currently not being used (no keyboard activity, no load average, no active telnet users, etc), Condor can effectively harness otherwise idle machines throughout a pool of machines. This is important because often times the amount of compute power represented by the aggregate total of all the non-dedicated desktop workstations sitting on people's desks throughout the organization is far greater than the compute power of a dedicated central resource.

Condor has several unique capabilities at its disposal which are geared toward effectively utilizing non-dedicated resources that are not owned or managed by a centralized resource. These include transparent process checkpoint and migration, remote system calls, and ClassAds. Read section 1.2 for a general discussion of these features before reading any further.

2.3 Matchmaking with ClassAds

Before you learn about how to submit a job, it is important to understand how Condor allocates resources. Understanding the unique framework by which Condor matches submitted jobs with machines is the key to getting the most from Condor's scheduling algorithm.

Condor simplifies job submission by acting as a matchmaker of ClassAds. Condor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers list constraints that need to be satisfied. For instance, a buyer has a maximum spending limit, and a seller requires a minimum purchase price. Furthermore, both want to rank requests to their own advantage. Certainly a seller would rank one offer of \$50 dollars higher than a different offer of \$25. In Condor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, along with other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which all the different owners have graciously allowed their machine to be part of the Condor pool. You may advertise that your machine is only willing to run jobs at night and when there is no keyboard activity on your machine. In addition, you may advertise a preference (rank) for running jobs submitted by you or one of your co-workers.

Likewise, when submitting a job, you specify a ClassAd with your requirements and preferences. The ClassAd includes the type of machine you wish to use. For instance, perhaps you are looking for the fastest floating point performance available. You want Condor to rank available machines based upon floating point performance. Or, perhaps you care only that the machine has a minimum of 128 Mbytes of RAM. Or, perhaps you will take any machine you can get! These job attributes

and requirements are bundled up into a job ClassAd.

Condor plays the role of a matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. Condor makes certain that all requirements in both ClassAds are satisfied.

2.3.1 Inspecting Machine ClassAds with `condor_status`

Once Condor is installed, you will get a feel for what a machine ClassAd does by trying the `condor_status` command. Try the `condor_status` command to get a summary of information from ClassAds about the resources available in your pool. Type `condor_status` and hit enter to see a summary similar to the following:

Name	Arch	OpSys	State	Activity	LoadAv	Mem	ActvtyTime
adriana.cs	INTEL	SOLARIS251	Claimed	Busy	1.000	64	0+01:10:00
alfred.cs.	INTEL	SOLARIS251	Claimed	Busy	1.000	64	0+00:40:00
amul.cs.wi	SUN4u	SOLARIS251	Owner	Idle	1.000	128	0+06:20:04
anfrom.cs.	SUN4x	SOLARIS251	Claimed	Busy	1.000	32	0+05:16:22
anthrax.cs	INTEL	SOLARIS251	Claimed	Busy	0.285	64	0+00:00:00
astro.cs.w	INTEL	SOLARIS251	Claimed	Busy	0.949	64	0+05:30:00
aura.cs.wi	SUN4u	SOLARIS251	Owner	Idle	1.043	128	0+14:40:15

...

The `condor_status` command has options that summarize machine ads in a variety of ways. For example,

`condor_status -available` shows only machines which are willing to run jobs now.

`condor_status -run` shows only machines which are currently running jobs.

`condor_status -l` lists the machine ClassAds for all machines in the pool.

Refer to the `condor_status` command reference page located on page 787 for a complete description of the `condor_status` command.

Figure 2.1 shows the complete machine ClassAd for a single workstation: `alfred.cs.wisc.edu`. Some of the listed attributes are used by Condor for scheduling. Other attributes are for information purposes. An important point is that *any* of the attributes in a machine ad can be utilized at job submission time as part of a request or preference on what machine to use. Additional attributes can be easily added. For example, your site administrator can add a physical location attribute to your machine ClassAds.

```

MyType = "Machine"
TargetType = "Job"
Name = "alfred.cs.wisc.edu"
Machine = "alfred.cs.wisc.edu"
StartdIpAddr = "<128.105.83.11:32780>"
Arch = "INTEL"
OpSys = "SOLARIS251"
UidDomain = "cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
State = "Unclaimed"
EnteredCurrentState = 892191963
Activity = "Idle"
EnteredCurrentActivity = 892191062
VirtualMemory = 185264
Disk = 35259
KFlops = 19992
Mips = 201
LoadAvg = 0.019531
CondorLoadAvg = 0.000000
KeyboardIdle = 5124
ConsoleIdle = 27592
Cpus = 1
Memory = 64
AFSCell = "cs.wisc.edu"
START = LoadAvg - CondorLoadAvg <= 0.300000 && KeyboardIdle > 15 * 60
Requirements = TRUE
Rank = Owner == "johndoe" || Owner == "friendofjohn"
CurrentRank = - 1.000000
LastHeardFrom = 892191963

```

Figure 2.1: Sample output from *condor_status -l alfred*

2.4 Road-map for Running Jobs

The road to using Condor effectively is a short one. The basics are quickly and easily learned.

Here are all the steps needed to run a job using Condor.

Code Preparation. A job run under Condor must be able to run as a background batch job. Condor runs the program unattended and in the background. A program that runs in the background will not be able to do interactive input and output. Condor can redirect console output (stdout and stderr) and keyboard input (stdin) to and from files for you. Create any needed files that contain the proper keystrokes needed for program input. Make certain the program will run correctly with the files.

The Condor Universe. Condor has several runtime environments (called a *universe*) from which to choose. Of the universes, two are likely choices when learning to submit a job to Condor: the standard universe and the vanilla universe. The standard universe allows a job running

under Condor to handle system calls by returning them to the machine where the job was submitted. The standard universe also provides the mechanisms necessary to take a checkpoint and migrate a partially completed job, should the machine on which the job is executing become unavailable. To use the standard universe, it is necessary to relink the program with the Condor library using the *condor_compile* command. The manual page for *condor_compile* on page 695 has details.

The vanilla universe provides a way to run jobs that cannot be relinked. There is no way to take a checkpoint or migrate a job executed under the vanilla universe. For access to input and output files, jobs must either use a shared file system, or use Condor's File Transfer mechanism.

Choose a universe under which to run the Condor program, and re-link the program if necessary.

Submit description file. Controlling the details of a job submission is a submit description file.

The file contains information about the job such as what executable to run, the files to use for keyboard and screen data, the platform type required to run the program, and where to send e-mail when the job completes. You can also tell Condor how many times to run a program; it is simple to run the same program multiple times with multiple data sets.

Write a submit description file to go with the job, using the examples provided in section 2.5.1 for guidance.

Submit the Job. Submit the program to Condor with the *condor_submit* command.

Once submitted, Condor does the rest toward running the job. Monitor the job's progress with the *condor_q* and *condor_status* commands. You may modify the order in which Condor will run your jobs with *condor_prio*. If desired, Condor can even inform you in a log file every time your job is checkpointed and/or migrated to a different machine.

When your program completes, Condor will tell you (by e-mail, if preferred) the exit status of your program and various statistics about its performances, including time used and I/O performed. If you are using a log file for the job (which is recommended) the exit status will be recorded in the log file. You can remove a job from the queue prematurely with *condor_rm*.

2.4.1 Choosing a Condor Universe

A *universe* in Condor defines an execution environment. Condor Version 7.2.3 supports several different universes for user jobs:

- Standard
- Vanilla
- MPI
- Grid

- Java
- Scheduler
- Local
- Parallel
- VM

The **universe** under which a job runs is specified in the submit description file. If a universe is not specified, the default is standard.

The standard universe provides migration and reliability, but has some restrictions on the programs that can be run. The vanilla universe provides fewer services, but has very few restrictions. The MPI universe is for programs written to the MPICH interface. See section 2.9.5 for more about MPI and Condor. The MPI Universe has been superseded by the parallel universe. The grid universe allows users to submit jobs using Condor's interface. These jobs are submitted for execution on grid resources. The java universe allows users to run jobs written for the Java Virtual Machine (JVM). The scheduler universe allows users to submit lightweight jobs to be spawned by the *condor_schedd* daemon on the submit host itself. The parallel universe is for programs that require multiple machines for one job. See section 2.9 for more about the Parallel universe. The vm universe allows users to run jobs where the job is no longer a simple executable, but a disk image, facilitating the execution of a virtual machine.

Standard Universe

In the standard universe, Condor provides *checkpointing* and *remote system calls*. These features make a job more reliable and allow it uniform access to resources from anywhere in the pool. To prepare a program as a standard universe job, it must be relinked with *condor_compile*. Most programs can be prepared as a standard universe job, but there are a few restrictions.

Condor checkpoints a job at regular intervals. A *checkpoint image* is essentially a snapshot of the current state of a job. If a job must be migrated from one machine to another, Condor makes a checkpoint image, copies the image to the new machine, and restarts the job continuing the job from where it left off. If a machine should crash or fail while it is running a job, Condor can restart the job on a new machine using the most recent checkpoint image. In this way, jobs can run for months or years even in the face of occasional computer failures.

Remote system calls make a job perceive that it is executing on its home machine, even though the job may execute on many different machines over its lifetime. When a job runs on a remote machine, a second process, called a *condor_shadow* runs on the machine where the job was submitted.

When the job attempts a system call, the *condor_shadow* performs the system call instead and sends the results to the remote machine. For example, if a job attempts to open a file that is stored on the submitting machine, the *condor_shadow* will find the file, and send the data to the machine where the job is running.

To convert your program into a standard universe job, you must use *condor_compile* to relink it with the Condor libraries. Put *condor_compile* in front of your usual link command. You do not need to modify the program's source code, but you do need access to the unlinked object files. A commercial program that is packaged as a single executable file cannot be converted into a standard universe job.

For example, if you would have linked the job by executing:

```
% cc main.o tools.o -o program
```

Then, relink the job for Condor with:

```
% condor_compile cc main.o tools.o -o program
```

There are a few restrictions on standard universe jobs:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.
9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.

11. On Linux, your job must be statically linked. *condor_compile* does this by default. Dynamic linking is allowed on Solaris.
12. Reading to or writing from files larger than 2 GB is not supported.

Vanilla Universe

The vanilla universe in Condor is intended for programs which cannot be successfully re-linked. Shell scripts are another case where the vanilla universe is useful. Unfortunately, jobs run under the vanilla universe cannot checkpoint or use remote system calls. This has unfortunate consequences for a job that is partially completed when the remote machine running a job must be returned to its owner. Condor has only two choices. It can suspend the job, hoping to complete it at a later time, or it can give up and restart the job *from the beginning* on another machine in the pool.

Since Condor's remote system call features cannot be used with the vanilla universe, access to the job's input and output files becomes a concern. One option is for Condor to rely on a shared file system, such as NFS or AFS. Alternatively, Condor has a mechanism for transferring files on behalf of the user. In this case, Condor will transfer any files needed by a job to the execution site, run the job, and transfer the output back to the submitting machine.

Under Unix, the Condor presumes a shared file system for vanilla jobs. However, if a shared file system is unavailable, a user can enable the Condor File Transfer mechanism. On Windows platforms, the default is to use the File Transfer mechanism. For details on running a job with a shared file system, see section 2.5.3 on page 24. For details on using the Condor File Transfer mechanism, see section 2.5.4 on page 26.

Grid Universe

The Grid universe in Condor is intended to provide the standard Condor interface to users who wish to start jobs intended for remote management systems. Section 5.3 on page 494 has details on using the Grid universe. The manual page for *condor_submit* on page 795 has detailed descriptions of the grid-related attributes.

Java Universe

A program submitted to the Java universe may run on any sort of machine with a JVM regardless of its location, owner, or JVM version. Condor will take care of all the details such as finding the JVM binary and setting the classpath.

Scheduler Universe

The scheduler universe allows users to submit lightweight jobs to be run immediately, alongside the *condor_schedd* daemon on the submit host itself. Scheduler universe jobs are not matched with

a remote machine, and will never be preempted. The job's requirements expression is evaluated against the *condor_schedd*'s ClassAd.

Originally intended for meta-schedulers such as *condor_dagman*, the scheduler universe can also be used to manage jobs of any sort that must run on the submit host.

However, unlike the local universe, the scheduler universe does not use a *condor_starter* daemon to manage the job, and thus offers limited features and policy support. The local universe is a better choice for most jobs which must run on the submit host, as it offers a richer set of job management features, and is more consistent with other universes such as the vanilla universe. The scheduler universe may be retired in the future, in favor of the newer local universe.

Local Universe

The local universe allows a Condor job to be submitted and executed with different assumptions for the execution conditions of the job. The job does not wait to be matched with a machine. It instead executes right away, on the machine where the job is submitted. The job will never be preempted. The job's requirements expression is evaluated against the *condor_schedd*'s ClassAd.

Parallel Universe

The parallel universe allows parallel programs, such as MPI jobs, to be run within the opportunistic Condor environment. Please see section 2.9 for more details.

VM Universe

Condor facilitates the execution of VMware and Xen virtual machines with the vm universe.

Please see section 2.11 for details.

2.5 Submitting a Job

A job is submitted for execution to Condor using the *condor_submit* command. *condor_submit* takes as an argument the name of a file called a submit description file. This file contains commands and keywords to direct the queuing of jobs. In the submit description file, Condor finds everything it needs to know about the job. Items such as the name of the executable to run, the initial working directory, and command-line arguments to the program all go into the submit description file. *condor_submit* creates a job ClassAd based upon the information, and Condor works toward running the job.

The contents of a submit file can save time for Condor users. It is easy to submit multiple runs of a program to Condor. To run the same program 500 times on 500 different input data sets, arrange

your data files accordingly so that each run reads its own input, and each run writes its own output. Each individual run may have its own initial working directory, `stdin`, `stdout`, `stderr`, command-line arguments, and shell environment. A program that directly opens its own files will read the file names to use either from `stdin` or from the command line. A program that opens a static filename every time will need to use a separate subdirectory for the output of each run.

The `condor_submit` manual page is on page 795 and contains a complete and full description of how to use `condor_submit`.

2.5.1 Sample submit description files

In addition to the examples of submit description files given in the `condor_submit` manual page, here are a few more.

Example 1

Example 1 is the simplest submit description file possible. It queues up one copy of the program `foo` (which had been created by `condor_compile`) for execution by Condor. Since no platform is specified, Condor will use its default, which is to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. No input, output, and error commands are given in the submit description file, so the files `stdin`, `stdout`, and `stderr` will all refer to `/dev/null`. The program may produce output by explicitly opening a file and writing to it. A log file, `foo.log`, will also be produced that contains events the job had during its lifetime inside of Condor. When the job finishes, its exit conditions will be noted in the log file. It is recommended that you always have a log file so you know what happened to your jobs.

```
#####
#
# Example 1
# Simple condor job description file
#
#####

Executable      = foo
Log              = foo.log
Queue
```

Example 2

Example 2 queues two copies of the program `mathematica`. The first copy will run in directory `run_1`, and the second will run in directory `run_2`. For both queued copies, `stdin` will be `test.data`, `stdout` will be `loop.out`, and `stderr` will be `loop.error`. There will be two sets of files written, as the files are each written to their own directories. This is a convenient

way to organize data if you have a large group of Condor jobs to run. The example file shows program submission of *mathematica* as a vanilla universe job. This may be necessary if the source and/or object code to program *mathematica* is not available.

```
#####
#
# Example 2: demonstrate use of multiple
# directories for data organization.
#
#####

Executable      = mathematica
Universe = vanilla
input   = test.data
output  = loop.out
error   = loop.error
Log     = loop.log

Initialdir      = run_1
Queue

Initialdir      = run_2
Queue
```

Example 3

The submit description file for Example 3 queues 150 runs of program *foo* which has been compiled and linked for Sun workstations running Solaris 8. This job requires Condor to run the program on machines which have greater than 32 megabytes of physical memory, and expresses a preference to run the program on machines with more than 64 megabytes, if such machines are available. It also advises Condor that it will use up to 28 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. So, files *stdin*, *stdout*, and *stderr* will refer to *in.0*, *out.0*, and *err.0* for the first run of the program, *in.1*, *out.1*, and *err.1* for the second run of the program, and so forth. A log file containing entries about when and where Condor runs, checkpoints, and migrates processes for the 150 queued programs will be written into file *foo.log*.

```
#####
#
# Example 3: Show off some fancy features including
# use of pre-defined macros and logging.
#
#####
```

```

Executable      = foo
Requirements    = Memory >= 32 && OpSys == "SOLARIS28" && Arch == "SUN4u"
Rank            = Memory >= 64
Image_Size      = 28 Meg

Error           = err.$(Process)
Input           = in.$(Process)
Output          = out.$(Process)
Log             = foo.log

Queue 150

```

2.5.2 About Requirements and Rank

The `requirements` and `rank` commands in the submit description file are powerful and flexible. Using them effectively requires care, and this section presents those details.

Both `requirements` and `rank` need to be specified as valid Condor ClassAd expressions, however, default values are set by the `condor_submit` program if these are not defined in the submit description file. From the `condor_submit` manual page and the above examples, you see that writing ClassAd expressions is intuitive, especially if you are familiar with the programming language C. There are some pretty nifty expressions you can write with ClassAds. A complete description of ClassAds and their expressions can be found in section 4.1 on page 430.

All of the commands in the submit description file are case insensitive, *except* for the ClassAd attribute string values. ClassAds attribute names are case insensitive, but ClassAd string values are *case preserving*.

Note that the comparison operators (`<`, `>`, `<=`, `>=`, and `==`) compare strings case insensitively. The special comparison operators `=?=` and `!=` compare strings case sensitively.

A **requirements** or **rank** command in the submit description file may utilize attributes that appear in a machine or a job ClassAd. Within the submit description file (for a job) the prefix `MY.` (on a ClassAd attribute name) causes a reference to the job ClassAd attribute, and the prefix `TARGET.` causes a reference to a potential machine or matched machine ClassAd attribute.

The `condor_status` command displays statistics about machines within the pool. The `-l` option displays the machine ClassAd attributes for all machines in the Condor pool. The job ClassAds, if there are jobs in the queue, can be seen with the `condor_q -l` command. This shows all the defined attributes for current jobs in the queue.

A list of defined ClassAd attributes for job ClassAds is given in the unnumbered Appendix on page 879. A list of defined ClassAd attributes for machine ClassAds is given in the unnumbered Appendix on page 886.

Rank Expression Examples

When considering the match between a job and a machine, rank is used to choose a match from among all machines that satisfy the job's requirements and are available to the user, after accounting for the user's priority and the machine's rank of the job. The rank expressions, simple or complex, define a numerical value that expresses preferences.

The job's rank expression evaluates to one of three values. It can be UNDEFINED, ERROR, or a floating point value. If rank evaluates to a floating point value, the best match will be the one with the largest, positive value. If no rank is given in the submit description file, then Condor substitutes a default value of 0.0 when considering machines to match. If the job's rank of a given machine evaluates to UNDEFINED or ERROR, this same value of 0.0 is used. Therefore, the machine is still considered for a match, but has no rank above any other.

A boolean expression evaluates to the numerical value of 1.0 if true, and 0.0 if false.

The following rank expressions provide examples to follow.

For a job that desires the machine with the most available memory:

```
Rank = memory
```

For a job that prefers to run on a friend's machine on Saturdays and Sundays:

```
Rank = ( (clockday == 0) || (clockday == 6) )
      && (machine == "friend.cs.wisc.edu")
```

For a job that prefers to run on one of three specific machines:

```
Rank = (machine == "friend1.cs.wisc.edu") ||
      (machine == "friend2.cs.wisc.edu") ||
      (machine == "friend3.cs.wisc.edu")
```

For a job that wants the machine with the best floating point performance (on Linpack benchmarks):

```
Rank = kflops
```

This particular example highlights a difficulty with rank expression evaluation as currently defined. While all machines have floating point processing ability, not all machines will have the `kflops` attribute defined. For machines where this attribute is not defined, Rank will evaluate to the value UNDEFINED, and Condor will use a default rank of the machine of 0.0. The rank attribute will only rank machines where the attribute is defined. Therefore, the machine with the highest floating point performance may not be the one given the highest rank.

So, it is wise when writing a rank expression to check if the expression's evaluation will lead to the expected resulting ranking of machines. This can be accomplished using the *condor_status* command with the *-constraint* argument. This allows the user to see a list of machines that fit a constraint. To see which machines in the pool have *kflops* defined, use

```
condor_status -constraint kflops
```

Alternatively, to see a list of machines where *kflops* is not defined, use

```
condor_status -constraint "kflops=?=undefined"
```

For a job that prefers specific machines in a specific order:

```
Rank = ((machine == "friend1.cs.wisc.edu")*3) +  
        ((machine == "friend2.cs.wisc.edu")*2) +  
        (machine == "friend3.cs.wisc.edu")
```

If the machine being ranked is "friend1.cs.wisc.edu", then the expression

```
(machine == "friend1.cs.wisc.edu")
```

is true, and gives the value 1.0. The expressions

```
(machine == "friend2.cs.wisc.edu")
```

and

```
(machine == "friend3.cs.wisc.edu")
```

are false, and give the value 0.0. Therefore, rank evaluates to the value 3.0. In this way, machine "friend1.cs.wisc.edu" is ranked higher than machine "friend2.cs.wisc.edu", machine "friend2.cs.wisc.edu" is ranked higher than machine "friend3.cs.wisc.edu", and all three of these machines are ranked higher than others.

2.5.3 Submitting Jobs Using a Shared File System

If vanilla, java, parallel (or MPI) universe jobs are submitted without using the File Transfer mechanism, Condor must use a shared file system to access input and output files. In this case, the job *must* be able to access the data files from any machine on which it could potentially run.

As an example, suppose a job is submitted from blackbird.cs.wisc.edu, and the job requires a particular data file called /u/p/s/psilord/data.txt. If the job were to run on cardinal.cs.wisc.edu, the file /u/p/s/psilord/data.txt must be available through either NFS or AFS for the job to run correctly.

Condor allows users to ensure their jobs have access to the right shared files by using the `FileSystemDomain` and `UidDomain` machine ClassAd attributes. These attributes specify which machines have access to the same shared file systems. All machines that mount the same shared directories in the same locations are considered to belong to the same file system domain. Similarly, all machines that share the same user information (in particular, the same UID, which is important for file systems like NFS) are considered part of the same UID domain.

The default configuration for Condor places each machine in its own UID domain and file system domain, using the full host name of the machine as the name of the domains. So, if a pool *does* have access to a shared file system, the pool administrator *must* correctly configure Condor such that all the machines mounting the same files have the same `FileSystemDomain` configuration. Similarly, all machines that share common user information must be configured to have the same `UidDomain` configuration.

When a job relies on a shared file system, Condor uses the `requirements` expression to ensure that the job runs on a machine in the correct `UidDomain` and `FileSystemDomain`. In this case, the default `requirements` expression specifies that the job must run on a machine with the same `UidDomain` and `FileSystemDomain` as the machine from which the job is submitted. This default is almost always correct. However, in a pool spanning multiple `UidDomains` and/or `FileSystemDomains`, the user may need to specify a different `requirements` expression to have the job run on the correct machines.

For example, imagine a pool made up of both desktop workstations and a dedicated compute cluster. Most of the pool, including the compute cluster, has access to a shared file system, but some of the desktop machines do not. In this case, the administrators would probably define the `FileSystemDomain` to be `cs.wisc.edu` for all the machines that mounted the shared files, and to the full host name for each machine that did not. An example is `jimi.cs.wisc.edu`.

In this example, a user wants to submit vanilla universe jobs from her own desktop machine (`jimi.cs.wisc.edu`) which does not mount the shared file system (and is therefore in its own file system domain, in its own world). But, she wants the jobs to be able to run on more than just her own machine (in particular, the compute cluster), so she puts the program and input files onto the shared file system. When she submits the jobs, she needs to tell Condor to send them to machines that have access to that shared data, so she specifies a different `requirements` expression than the default:

```
Requirements = TARGET.UidDomain == "cs.wisc.edu" && \  
              TARGET.FileSystemDomain == "cs.wisc.edu"
```

WARNING: If there is *no* shared file system, or the Condor pool administrator does not configure the `FileSystemDomain` setting correctly (the default is that each machine in a pool is in its own file system and UID domain), a user submits a job that cannot use remote system calls (for example,

a vanilla universe job), and the user does not enable Condor's File Transfer mechanism, the job will *only* run on the machine from which it was submitted.

2.5.4 Submitting Jobs Without a Shared File System: Condor's File Transfer Mechanism

Condor works well without a shared file system. The Condor file transfer mechanism is utilized by the user when the user submits jobs. Condor will transfer any files needed by a job from the machine where the job was submitted into a temporary working directory on the machine where the job is to be executed. Condor executes the job and transfers output back to the submitting machine. The user specifies which files to transfer, and at what point the output files should be copied back to the submitting machine. This specification is done within the job's submit description file.

The default behavior of the file transfer mechanism varies across the different Condor universes, and it differs between UNIX and Windows machines.

Default Behavior across Condor Universes and Platforms

For jobs submitted under the standard universe, the existence of a shared file system is not relevant. Access to files (input and output) is handled through Condor's remote system call mechanism. The executable and checkpoint files are transferred automatically, when needed. Therefore, the user does not need to change the submit description file if there is no shared file system.

For the vanilla, java, MPI, and parallel universes, access to files (including the executable) through a shared file system is presumed as a default on UNIX machines. If there is no shared file system, then Condor's file transfer mechanism must be explicitly enabled. When submitting a job from a Windows machine, Condor presumes the opposite: no access to a shared file system. It instead enables the file transfer mechanism by default. Submission of a job might need to specify which files to transfer, and/or when to transfer the output files back.

For the grid universe, jobs are to be executed on remote machines, so there would never be a shared file system between machines. See section 5.3.2 for more details.

For the scheduler universe, Condor is only using the machine from which the job is submitted. Therefore, the existence of a shared file system is not relevant.

Specifying If and When to Transfer Files

To enable the file transfer mechanism, two commands are placed in the job's submit description file: **should_transfer_files** and **when_to_transfer_output**. An example is:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

The **should_transfer_files** command specifies whether Condor should transfer input files from the submit machine to the remote machine where the job executes. It also specifies whether the output files are transferred back to the submit machine. The command takes on one of three possible values:

1. YES: Condor always transfers both input and output files.
2. IF_NEEDED: Condor transfers files if the job is matched with (and to be executed on) a machine in a different `FileSystemDomain` than the one the submit machine belongs to. If the job is matched with a machine in the local `FileSystemDomain`, Condor will not transfer files and relies on a shared file system.
3. NO: Condor's file transfer mechanism is disabled.

The **when_to_transfer_output** command tells Condor when output files are to be transferred back to the submit machine after the job has executed on a remote machine. The command takes on one of two possible values:

1. ON_EXIT: Condor transfers output files back to the submit machine only when the job exits on its own.
2. ON_EXIT_OR_EVICT: Condor will always do the transfer, whether the job completes on its own, is preempted by another job, vacates the machine, or is killed. As the job completes on its own, files are transferred back to the directory where the job was submitted, as expected. For the other cases, *files are transferred back at eviction time*. These files are placed in the directory defined by the configuration variable `SPOOL`, not the directory from which the job was submitted. The transferred files are named using the `ClusterId` and `ProcId` job `ClassAd` attributes. The file name takes the form:

```
cluster<X>.proc<Y>.subproc0
```

where `<X>` is the value of `ClusterId`, and `<Y>` is the value of `ProcId`. As an example, job 735.0 may produce the file

```
$(SPOOL)/cluster735.proc0.subproc0
```

This is only useful if partial runs of the job are valuable. An example of valuable partial runs is when the application produces its own checkpoints.

There is no default value for **when_to_transfer_output**. If using the file transfer mechanism, this command must be defined. If **when_to_transfer_output** is specified in the submit description file, but **should_transfer_files** is not, Condor assumes a value of YES for **should_transfer_files**.

NOTE: The combination of:

```
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT_OR_EVICT
```

would produce undefined file access semantics. Therefore, this combination is prohibited by *condor_submit*.

When submitting from a Unix platform, the file transfer mechanism is unused by default. If neither **when_to_transfer_output** or **should_transfer_files** are defined, Condor assumes `should_transfer_files = NO`.

When submitting from a Windows platform, Condor does not provide any way to use a shared file system for jobs. Therefore, if neither **when_to_transfer_output** or **should_transfer_files** are defined, the file transfer mechanism is enabled by default with the following values:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

Specifying What Files to Transfer

If the file transfer mechanism is enabled, Condor will transfer the following files before the job is run on a remote machine.

1. the executable
2. the input, as defined with the **input** command
3. any jar files (for the Java universe)

If the job requires any other input files, the submit description file should utilize the **transfer_input_files** command. This comma-separated list specifies any other files that Condor is to transfer to a remote site to set up the execution environment for the job before it is run. These files are placed in the same temporary working directory as the job's executable. At this time, directories can not be transferred in this way. For example:

```
transfer_input_files = file1,file2
```

As a default, for jobs other than those submitted to the grid universe, any files that are modified or created by the job in the temporary directory at the remote site are transferred back to the machine from which the job was submitted. Most of the time, this is the best option. To restrict the files that are transferred, specify the exact list of files with **transfer_output_files**. Delimit these file names with a comma. When this list is defined, and any of the files do not exist as the job exits, Condor considers this an error, and re-runs the job.

WARNING: Do not specify **transfer_output_files** (for other than grid universe jobs) unless there is a really good reason – it is best to let Condor figure things out by itself based upon what output the job produces.

For grid universe jobs, files to be transferred (other than standard output and standard error) must be specified using **transfer_output_files** in the submit description file.

File Paths for File Transfer

The file transfer mechanism specifies file names and/or paths on both the file system of the submit machine and on the file system of the execute machine. Care must be taken to know which machine (submit or execute) is utilizing the file name and/or path.

Files in the **transfer_input_files** command are specified as they are accessed on the submit machine. The program (as it executes) accesses files as they are found on the execute machine.

There are three ways to specify files and paths for **transfer_input_files**:

1. Relative to the submit directory, if the submit command **initialdir** is not specified.
2. Relative to the initial directory, if the submit command **initialdir** is specified.
3. Absolute.

Before executing the program, Condor copies the executable, an input file as specified by the submit command **input**, along with any input files specified by **transfer_input_files**. All these files are placed into a temporary directory (on the execute machine) in which the program runs. Therefore, the executing program must access input files *without* paths. Because all transferred files are placed into a single, flat directory, input files must be uniquely named to avoid collision when transferred. A collision causes the last file in the list to overwrite the earlier one.

If the program creates output files during execution, it must create them within the temporary working directory. Condor transfers back all files within the temporary working directory that have been modified or created. To transfer back only a subset of these files, the submit command **transfer_output_files** is defined. Transfer of files that exist, but are not within the temporary working directory is not supported. Condor's behavior in this instance is undefined.

It is okay to create files outside the temporary working directory on the file system of the execute machine, (in a directory such as `/tmp`) if this directory is guaranteed to exist and be accessible on all possible execute machines. However, transferring such a file back after execution completes may not be done.

Here are several examples to illustrate the use of file transfer. The program executable is called *my_program*, and it uses three command-line arguments as it executes: two input file names and an output file name. The program executable and the submit description file for this job are located in directory `/scratch/test`.

The directory tree for all these examples:

```
/scratch/test (directory)
  my_program.condor (the submit description file)
  my_program (the executable)
  files (directory)
    logs2 (directory)
    in1 (file)
```

```

        in2 (file)
    logs (directory)

```

Example 1 This simple example explicitly transfers input files. These input files to be transferred are specified relative to the directory where the job is submitted. The single output file, `out1`, created when the job is executed will be transferred back into the directory `/scratch/test`, *not* the `files` directory.

```

# file name: my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1, files/in2

Arguments      = in1 in2 out1
Queue

```

Example 2 This second example is identical to Example 1, except that absolute paths to the input files are specified, instead of relative paths to the input files.

```

# file name: my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1, /scratch/test/files/in2

Arguments      = in1 in2 out1
Queue

```

Example 3 This third example illustrates the use of the submit command **initialdir**, and its effect on the paths used for the various files. The expected location of the executable is not affected by the **initialdir** command. All other files (specified by **input**, **output**, **transfer_input_files**, as well as files modified or created by the job and automatically transferred back) are located relative to the specified **initialdir**. Therefore, the output file, `out1`, will be placed in the `files` directory. Note that the `logs2` directory exists to make this example work correctly.

```

# file name: my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla

```

```

Error          = logs2/err.%(cluster)
Output         = logs2/out.%(cluster)
Log            = logs2/log.%(cluster)

initialdir     = files

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = in1, in2

Arguments      = in1 in2 out1
Queue

```

Example 4 – Illustrates an Error This example illustrates a job that will fail. The files specified using the **transfer_input_files** command work correctly (see Example 1). However, relative paths to files in the **arguments** command cause the executing program to fail. The file system on the submission side may utilize relative paths to files, however those files are placed into a single, flat, temporary directory on the execute machine.

Note that this specification and submission will cause the job to fail and re-execute.

```

# file name: my_program.condor
# Condor submit description file for my_program
Executable     = my_program
Universe       = vanilla
Error          = logs/err.%(cluster)
Output         = logs/out.%(cluster)
Log            = logs/log.%(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1, files/in2

Arguments      = files/in1 files/in2 files/out1
Queue

```

This example fails with the following error:

```
err: files/out1: No such file or directory.
```

Example 5 – Illustrates an Error As with Example 4, this example illustrates a job that will fail. The executing program's use of absolute paths cannot work.

```

# file name: my_program.condor
# Condor submit description file for my_program
Executable     = my_program
Universe       = vanilla
Error          = logs/err.%(cluster)
Output         = logs/out.%(cluster)
Log            = logs/log.%(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1, /scratch/test/files/in2

Arguments = /scratch/test/files/in1 /scratch/test/files/in2 /scratch/test/files/out1
Queue

```

The job fails with the following error:

```
err: /scratch/test/files/out1: No such file or directory.
```

Example 6 – Illustrates an Error This example illustrates a failure case where the executing program creates an output file in a directory other than within the single, flat, temporary directory that the program executes within. The file creation may or may not cause an error, depending on the existence and permissions of the directories on the remote file system.

Further incorrect usage is seen during the attempt to transfer the output file back using the **transfer_output_files** command. The behavior of Condor for this case is undefined.

```
# file name: my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1, files/in2
transfer_output_files = /tmp/out1

Arguments      = in1 in2 /tmp/out1
Queue
```

Requirements and Rank for File Transfer

The `requirements` expression for a job must depend on the `should_transfer_files` command. The job must specify the correct logic to ensure that the job is matched with a resource that meets the file transfer needs. If no `requirements` expression is in the submit description file, or if the expression specified does not refer to the attributes listed below, *condor_submit* adds an appropriate clause to the `requirements` expression for the job. *condor_submit* appends these clauses with a logical AND, `&&`, to ensure that the proper conditions are met. Here are the default clauses corresponding to the different values of `should_transfer_files`:

1. `should_transfer_files = YES` results in the addition of the clause `(HasFileTransfer)`. If the job is always going to transfer files, it is required to match with a machine that has the capability to transfer files.
2. `should_transfer_files = NO` results in the addition of `(TARGET.FileSystemDomain == MY.FileSystemDomain)`. In addition, Condor automatically adds the `FileSystemDomain` attribute to the job ad, with whatever string is defined for the *condor_schedd* to which the job is submitted. If the job is not using the file transfer mechanism, Condor assumes it will need a shared file system, and therefore, a machine in the same `FileSystemDomain` as the submit machine.
3. `should_transfer_files = IF_NEEDED` results in the addition of

```
(HasFileTransfer || (TARGET.FileSystemDomain == MY.FileSystemDomain))
```

If Condor will optionally transfer files, it must require that the machine is *either* capable of transferring files *or* in the same file system domain.

To ensure that the job is matched to a machine with enough local disk space to hold all the transferred files, Condor automatically adds the `DiskUsage` job attribute. This attribute includes the total size of the job's executable and all input files to be transferred. Condor then adds an additional clause to the `Requirements` expression that states that the remote machine must have at least enough available disk space to hold all these files:

```
&& (Disk >= DiskUsage)
```

If `should_transfer_files = IF_NEEDED` and the job prefers to run on a machine in the local file system domain over transferring files, (but are still willing to allow the job to run remotely and transfer files), the rank expression works well. Use:

```
rank = (TARGET.FileSystemDomain == MY.FileSystemDomain)
```

The rank expression is a floating point number, so if other items are considered in ranking the possible machines this job may run on, add the items:

```
rank = kflops + (TARGET.FileSystemDomain == MY.FileSystemDomain)
```

The value of `kflops` can vary widely among machines, so this rank expression will likely not do as it intends. To place emphasis on the job running in the same file system domain, but still consider `kflops` among the machines in the file system domain, weight the part of the rank expression that is matching the file system domains. For example:

```
rank = kflops + (10000 * (TARGET.FileSystemDomain == MY.FileSystemDomain))
```

2.5.5 Environment Variables

The environment under which a job executes often contains information that is potentially useful to the job. Condor allows a user to both set and reference environment variables for a job or job cluster.

Within a submit description file, the user may define environment variables for the job's environment by using the **environment** command. See the *condor_submit* manual page at section 9 for more details about this command.

The submitter's entire environment can be copied into the job ClassAd for the job at job submission. The **getenv** command within the submit description file does this. See the *condor_submit* manual page at section 9 for more details about this command.

If the environment is set with the **environment** command *and* **getenv** is also set to true, values specified with **environment** override values in the submitter's environment (regardless of the order of the **environment** and **getenv** commands).

Commands within the submit description file may reference the environment variables of the submitter as a job is submitted. Submit description file commands use `$ENV(EnvironmentVariableName)` to reference the value of an environment variable. Again, see the *condor_submit* manual page at section 9 for more details about this usage.

Condor sets several additional environment variables for each executing job that may be useful for the job to reference.

- `_CONDOR_SCRATCH_DIR` gives the directory where the job may place temporary data files. This directory is unique for every job that is run, and its contents are deleted by Condor when the job stops running on a machine, no matter how the job completes.
- `_CONDOR_SLOT` gives the name of the slot (for SMP machines), on which the job is run. On machines with only a single slot, the value of this variable will be 1, just like the `SlotID` attribute in the machine's ClassAd. This setting is available in all universes. See section 3.12.7 for more details about SMP machines and their configuration.
- `CONDOR_VM` equivalent to `_CONDOR_SLOT` described above, except that it is only available in the standard universe. **NOTE:** : As of Condor version 6.9.3, this environment variable is deprecated. It will only be defined if the `ALLOW_VM_CRUFT` configuration setting is set to `TRUE`.
- `X509_USER_PROXY` gives the full path to the X509 user proxy file if one is associated with the job. (Typically a user will specify **x509userproxy** in the submit file.) This setting is currently available in the local, java, and vanilla universes.

2.5.6 Heterogeneous Submit: Execution on Differing Architectures

If executables are available for the different platforms of machines in the Condor pool, Condor can be allowed the choice of a larger number of machines when allocating a machine for a job. Modifications to the submit description file allow this choice of platforms.

A simplified example is a cross submission. An executable is available for one platform, but the submission is done from a different platform. Given the correct executable, the `requirements` command in the submit description file specifies the target architecture. For example, an executable compiled for a Sun 4, submitted from an Intel architecture running Linux would add the requirement

```
requirements = Arch == "SUN4x" && OpSys == "SOLARIS251"
```

Without this requirement, *condor_submit* will assume that the program is to be executed on a machine with the same platform as the machine where the job is submitted.

Cross submission works for all universes except `scheduler` and `local`. See section 5.3.8 for how matchmaking works in the `grid` universe. The burden is on the user to both obtain and specify the correct executable for the target architecture. To list the architecture and operating systems of the machines in a pool, run `condor_status`.

Vanilla Universe Example for Execution on Differing Architectures

A more complex example of a heterogeneous submission occurs when a job may be executed on many different architectures to gain full use of a diverse architecture and operating system pool. If the executables are available for the different architectures, then a modification to the submit description file will allow Condor to choose an executable after an available machine is chosen.

A special-purpose Machine Ad substitution macro can be used in string attributes in the submit description file. The macro has the form

```
$$ (MachineAdAttribute)
```

The `$$()` informs Condor to substitute the requested `MachineAdAttribute` from the machine where the job will be executed.

An example of the heterogeneous job submission has executables available for three platforms: LINUX Intel, Solaris26 Intel, and Solaris 8 Sun. This example uses *povray* to render images using a popular free rendering engine.

The substitution macro chooses a specific executable after a platform for running the job is chosen. These executables must therefore be named based on the machine attributes that describe a platform. The executables named

```
povray.LINUX.INTEL
povray.SOLARIS26.INTEL
povray.SOLARIS28.SUN4u
```

will work correctly for the macro

```
povray.$$ (OpSys) . $$ (Arch)
```

The executables or links to executables with this name are placed into the initial working directory so that they may be found by Condor. A submit description file that queues three jobs for this example:

```
#####
#
# Example of heterogeneous submission
#
```

```
#####

universe      = vanilla
Executable    = povray.$$ (OpSys) .$$ (Arch)
Log           = povray.log
Output        = povray.out.$ (Process)
Error         = povray.err.$ (Process)

Requirements  = (Arch == "INTEL" && OpSys == "LINUX") || \
                (Arch == "INTEL" && OpSys == "SOLARIS26") || \
                (Arch == "SUN4u" && OpSys == "SOLARIS28")

Arguments     = +W1024 +H768 +Iimage1.pov
Queue

Arguments     = +W1024 +H768 +Iimage2.pov
Queue

Arguments     = +W1024 +H768 +Iimage3.pov
Queue
```

These jobs are submitted to the vanilla universe to assure that once a job is started on a specific platform, it will finish running on that platform. Switching platforms in the middle of job execution cannot work correctly.

There are two common errors made with the substitution macro. The first is the use of a non-existent `MachineAdAttribute`. If the specified `MachineAdAttribute` does not exist in the machine's `ClassAd`, then Condor will place the job in the held state until the problem is resolved.

The second common error occurs due to an incomplete job set up. For example, the submit description file given above specifies three available executables. If one is missing, Condor report back that an executable is missing when it happens to match the job with a resource that requires the missing binary.

Standard Universe Example for Execution on Differing Architectures

Jobs submitted to the standard universe may produce checkpoints. A checkpoint can then be used to start up and continue execution of a partially completed job. For a partially completed job, the checkpoint and the job are specific to a platform. If migrated to a different machine, correct execution requires that the platform must remain the same.

In previous versions of Condor, the author of the heterogeneous submission file would need to write extra policy expressions in the `requirements` expression to force Condor to choose the same type of platform when continuing a checkpointed job. However, since it is needed in the common case, this additional policy is now automatically added to the `requirements` expression. The additional expression is added provided the user does not use `CkptArch` in the `requirements`

expression. Condor will remain backward compatible for those users who have explicitly specified `CkptRequirements`—implying use of `CkptArch`, in their requirements expression.

The expression added when the attribute `CkptArch` is not specified will default to

```
# Added by Condor
CkptRequirements = ((CkptArch == Arch) || (CkptArch != UNDEFINED)) && \
    ((CkptOpSys == OpSys) || (CkptOpSys != UNDEFINED))

Requirements = (<user specified policy>) && $(CkptRequirements)
```

The behavior of the `CkptRequirements` expressions and its addition to `requirements` is as follows. The `CkptRequirements` expression guarantees correct operation in the two possible cases for a job. In the first case, the job has not produced a checkpoint. The ClassAd attributes `CkptArch` and `CkptOpSys` will be undefined, and therefore the meta operator (`!=`) evaluates to true. In the second case, the job has produced a checkpoint. The Machine ClassAd is restricted to require further execution only on a machine of the same platform. The attributes `CkptArch` and `CkptOpSys` will be defined, ensuring that the platform chosen for further execution will be the same as the one used just before the checkpoint.

Note that this restriction of platforms also applies to platforms where the executables are binary compatible.

The complete submit description file for this example:

```
#####
#
# Example of heterogeneous submission
#
#####

universe      = standard
Executable    = povray.$$(OpSys).$(Arch)
Log           = povray.log
Output        = povray.out.$(Process)
Error         = povray.err.$(Process)

# Condor automatically adds the correct expressions to insure that the
# checkpointed jobs will restart on the correct platform types.
Requirements = ( (Arch == "INTEL" && OpSys == "LINUX") || \
    (Arch == "INTEL" && OpSys == "SOLARIS26") || \
    (Arch == "SUN4u" && OpSys == "SOLARIS28") )

Arguments     = +W1024 +H768 +Iimage1.pov
Queue

Arguments     = +W1024 +H768 +Iimage2.pov
```

Queue

Arguments = +W1024 +H768 +Iimage3.pov
Queue

2.6 Managing a Job

This section provides a brief summary of what can be done once jobs are submitted. The basic mechanisms for monitoring a job are introduced, but the commands are discussed briefly. You are encouraged to look at the man pages of the commands referred to (located in Chapter 9 beginning on page 672) for more information.

When jobs are submitted, Condor will attempt to find resources to run the jobs. A list of all those with jobs submitted may be obtained through *condor_status* with the *-submitters* option. An example of this would yield output similar to:

```
% condor_status -submitters
```

Name	Machine	Running	IdleJobs	HeldJobs
ballard@cs.wisc.edu	bluebird.c	0	11	0
nice-user.condor@cs.	cardinal.c	6	504	0
wright@cs.wisc.edu	finch.cs.w	1	1	0
jbasney@cs.wisc.edu	perdita.cs	0	0	5

	RunningJobs	IdleJobs	HeldJobs
ballard@cs.wisc.edu	0	11	0
jbasney@cs.wisc.edu	0	0	5
nice-user.condor@cs.	6	504	0
wright@cs.wisc.edu	1	1	0
Total	7	516	5

2.6.1 Checking on the progress of jobs

At any time, you can check on the status of your jobs with the *condor_q* command. This command displays the status of all queued jobs. An example of the output from *condor_q* is

```
% condor_q
```

```
-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
ID      OWNER      SUBMITTED  CPU_USAGE ST PRI SIZE CMD
125.0   jbasney      4/10 15:35 0+00:00:00 I -10 1.2 hello.remote
127.0   raman       4/11 15:35 0+00:00:00 R  0  1.4 hello
128.0   raman       4/11 15:35 0+00:02:33 I  0  1.4 hello

3 jobs; 2 idle, 1 running, 0 held
```

This output contains many columns of information about the queued jobs. The ST column (for status) shows the status of current jobs in the queue. An R in the status column means the the job is currently running. An I stands for idle. The job is not running right now, because it is waiting for a machine to become available. The status H is the hold state. In the hold state, the job will not be scheduled to run until it is released (see the *condor_hold* reference page located on page 730 and the *condor_release* reference page located on page 764). Older versions of Condor used a U in the status column to stand for unexpanded. In this state, a job has never produced a checkpoint, and when the job starts running, it will start running from the beginning. Newer versions of Condor do not use the U state.

The CPU_USAGE time reported for a job is the time that has been committed to the job. It is not updated for a job until the job checkpoints. At that time, the job has made guaranteed forward progress. Depending upon how the site administrator configured the pool, several hours may pass between checkpoints, so do not worry if you do not observe the CPU_USAGE entry changing by the hour. Also note that this is actual CPU time as reported by the operating system; it is not time as measured by a wall clock.

Another useful method of tracking the progress of jobs is through the user log. If you have specified a log command in your submit file, the progress of the job may be followed by viewing the log file. Various events such as execution commencement, checkpoint, eviction and termination are logged in the file. Also logged is the time at which the event occurred.

When your job begins to run, Condor starts up a *condor_shadow* process on the submit machine. The shadow process is the mechanism by which the remotely executing jobs can access the environment from which it was submitted, such as input and output files.

It is normal for a machine which has submitted hundreds of jobs to have hundreds of shadows running on the machine. Since the text segments of all these processes is the same, the load on the submit machine is usually not significant. If, however, you notice degraded performance, you can limit the number of jobs that can run simultaneously through the MAX_JOBS_RUNNING configuration parameter. Please talk to your system administrator for the necessary configuration change.

You can also find all the machines that are running your job through the *condor_status* command. For example, to find all the machines that are running jobs submitted by “breach@cs.wisc.edu,” type:

```
% condor_status -constraint 'RemoteUser == "breach@cs.wisc.edu"'
```

Name	Arch	OpSys	State	Activity	LoadAv	Mem	ActvtyTime
alfred.cs.	INTEL	SOLARIS251	Claimed	Busy	0.980	64	0+07:10:02
biron.cs.w	INTEL	SOLARIS251	Claimed	Busy	1.000	128	0+01:10:00
cambridge.	INTEL	SOLARIS251	Claimed	Busy	0.988	64	0+00:15:00
falcons.cs	INTEL	SOLARIS251	Claimed	Busy	0.996	32	0+02:05:03
happy.cs.w	INTEL	SOLARIS251	Claimed	Busy	0.988	128	0+03:05:00
istat03.st	INTEL	SOLARIS251	Claimed	Busy	0.883	64	0+06:45:01
istat04.st	INTEL	SOLARIS251	Claimed	Busy	0.988	64	0+00:10:00
istat09.st	INTEL	SOLARIS251	Claimed	Busy	0.301	64	0+03:45:00
...							

To find all the machines that are running any job at all, type:

```
% condor_status -run
```

Name	Arch	OpSys	LoadAv	RemoteUser	ClientMachine
adriana.cs	INTEL	SOLARIS251	0.980	hepcon@cs.wisc.edu	chevre.cs.wisc.
alfred.cs.	INTEL	SOLARIS251	0.980	breach@cs.wisc.edu	neufchatel.cs.w
amul.cs.wi	SUN4u	SOLARIS251	1.000	nice-user.condor@cs.	chevre.cs.wisc.
anfrom.cs.	SUN4x	SOLARIS251	1.023	ashoks@jules.ncsa.ui	jules.ncsa.uiuc
anthrax.cs	INTEL	SOLARIS251	0.285	hepcon@cs.wisc.edu	chevre.cs.wisc.
astro.cs.w	INTEL	SOLARIS251	1.000	nice-user.condor@cs.	chevre.cs.wisc.
aura.cs.wi	SUN4u	SOLARIS251	0.996	nice-user.condor@cs.	chevre.cs.wisc.
balder.cs.	INTEL	SOLARIS251	1.000	nice-user.condor@cs.	chevre.cs.wisc.
bamba.cs.w	INTEL	SOLARIS251	1.574	dmarino@cs.wisc.edu	riola.cs.wisc.e
bardolph.c	INTEL	SOLARIS251	1.000	nice-user.condor@cs.	chevre.cs.wisc.
...					

2.6.2 Removing a job from the queue

A job can be removed from the queue at any time by using the *condor_rm* command. If the job that is being removed is currently running, the job is killed without a checkpoint, and its queue entry is removed. The following example shows the queue of jobs before and after a job is removed.

```
% condor_q
```

```
-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
ID      OWNER      SUBMITTED    CPU_USAGE ST PRI SIZE CMD
125.0   jbasney        4/10 15:35   0+00:00:00 I -10 1.2 hello.remote
132.0   raman          4/11 16:57   0+00:00:00 R  0  1.4 hello
```

```
2 jobs; 1 idle, 1 running, 0 held
```

```
% condor_rm 132.0
```

```
Job 132.0 removed.
```

```
% condor_q
```

```
-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
ID      OWNER      SUBMITTED    CPU_USAGE ST PRI SIZE CMD
125.0   jbasney        4/10 15:35   0+00:00:00 I -10 1.2 hello.remote
```

```
1 jobs; 1 idle, 0 running, 0 held
```

2.6.3 Placing a job on hold

A job in the queue may be placed on hold by running the command *condor_hold*. A job in the hold state remains in the hold state until later released for execution by the command *condor_release*.

Use of the *condor_hold* command causes a hard kill signal to be sent to a currently running job (one in the running state). For a standard universe job, this means that no checkpoint is generated before the job stops running and enters the hold state. When released, this standard universe job continues its execution using the most recent checkpoint available.

Jobs in universes other than the standard universe that are running when placed on hold will start over from the beginning when released.

The manual page for *condor_hold* on page 730 and the manual page for *condor_release* on page 764 contain usage details.

2.6.4 Changing the priority of jobs

In addition to the priorities assigned to each user, Condor also provides each user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue and can be any integer value, with higher values meaning better priority.

The default priority of a job is 0, but can be changed using the *condor_prio* command. For example, to change the priority of a job to -15,

```
% condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
ID      OWNER      SUBMITTED  CPU_USAGE ST PRI  SIZE CMD
126.0   raman         4/11 15:06  0+00:00:00 I  0   0.3  hello

1 jobs; 1 idle, 0 running, 0 held

% condor_prio -p -15 126.0

% condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
ID      OWNER      SUBMITTED  CPU_USAGE ST PRI  SIZE CMD
126.0   raman         4/11 15:06  0+00:00:00 I -15 0.3  hello

1 jobs; 1 idle, 0 running, 0 held
```

It is important to note that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities do not impact user priorities. They are only a mechanism for the user to identify the relative importance of jobs among all the jobs submitted by the user to that specific queue.

2.6.5 Why does the job not run?

Users sometimes find that their jobs do not run. There are several reasons why a specific job does not run. These reasons include failed job or machine constraints, bias due to preferences, insufficient priority, and the preemption throttle that is implemented by the *condor_negotiator* to prevent thrashing. Many of these reasons can be diagnosed by using the *-analyze* option of *condor_q*. For example, a job (assigned the cluster.process value of 331228.2359) submitted to the local pool at UW-Madison is not running. Running *condor_q*'s analyzer provided the following information:

```
% condor_q -pool condor -name beak -analyze 331228.2359
```

```
-- Schedd: beak.cs.wisc.edu : <128.105.146.14:30918>
ID      OWNER      SUBMITTED  RUN_TIME ST PRI SIZE CMD
---
```

331228.2359: Run analysis summary. Of 819 machines,
 159 are rejected by your job's requirements
 137 reject your job because of their own requirements
 488 match but are serving users with a better priority in the pool
 11 match but reject the job for unknown reasons
 24 match but will not currently preempt their existing job
 0 are available to run your job

A second example shows a job that does not run because the job does not have a high enough priority to cause other running jobs to be preempted.

```
% condor_q -pool condor -name beak -analyze 207525.0

-- Schedd: beak.cs.wisc.edu : <128.105.146.14:30918>
ID      OWNER      SUBMITTED  RUN_TIME ST PRI SIZE CMD
---
```

207525.000: Run analysis summary. Of 818 machines,
 317 are rejected by your job's requirements
 419 reject your job because of their own requirements
 79 match but are serving users with a better priority in the pool
 3 match but reject the job for unknown reasons
 0 match but will not currently preempt their existing job
 0 are available to run your job

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect due to the instantaneous and local nature of the information it uses to detect the problem. Thus, it may be that the analyzer reports that resources are available to service the request, but the job still does not run. In most of these situations, the delay is transient, and the job will run during the next negotiation cycle.

If the problem persists and the analyzer is unable to detect the situation, it may be that the job begins to run but immediately terminates due to some problem. Viewing the job's error and log files (specified in the submit command file) and Condor's `SHADOW_LOG` file may assist in tracking down the problem. If the cause is still unclear, please contact your system administrator.

2.6.6 In the log file

In a job's log file are a log of events (a listing of events in chronological order) that occurred during the life of the job. The formatting of the events is always the same, so that they may be machine readable. Four fields are always present, and they will most often be followed by other fields that give further information that is specific to the type of event.

The first field in an event is the numeric value assigned as the event type in a 3-digit format. The second field identifies the job which generated the event. Within parentheses are the ClassAd job attributes of `ClusterId` value, `ProcId` value, and the MPI-specific rank for MPI universe jobs

or a set of zeros (for jobs run under universes other than MPI), separated by periods. The third field is the date and time of the event logging. The fourth field is a string that briefly describes the event. Fields that follow the fourth field give further information for the specific event type.

These are all of the events that can show up in a job log file:

Event Number: 000

Event Name: Job submitted

Event Description: This event occurs when a user submits a job. It is the first event you will see for a job, and it should only occur once.

Event Number: 001

Event Name: Job executing

Event Description: This shows up when a job is running. It might occur more than once.

Event Number: 002

Event Name: Error in executable

Event Description: The job couldn't be run because the executable was bad.

Event Number: 003

Event Name: Job was checkpointed

Event Description: The job's complete state was written to a checkpoint file. This might happen without the job being removed from a machine, because the checkpointing can happen periodically.

Event Number: 004

Event Name: Job evicted from machine

Event Description: A job was removed from a machine before it finished, usually for a policy reason: perhaps an interactive user has claimed the computer, or perhaps another job is higher priority.

Event Number: 005

Event Name: Job terminated

Event Description: The job has completed.

Event Number: 006

Event Name: Image size of job updated

Event Description: This is informational. It is referring to the memory that the job is using while running. It does not reflect the state of the job.

Event Number: 007

Event Name: Shadow exception

Event Description: The *condor_shadow*, a program on the submit computer that watches over the job and performs some services for the job, failed for some catastrophic reason. The job will leave the machine and go back into the queue.

Event Number: 008

Event Name: Generic log event

Event Description: Not used.

Event Number: 009

Event Name: Job aborted

Event Description: The user canceled the job.

Event Number: 010

Event Name: Job was suspended

Event Description: The job is still on the computer, but it is no longer executing. This is usually for a policy reason, like an interactive user using the computer.

Event Number: 011

Event Name: Job was unsuspended

Event Description: The job has resumed execution, after being suspended earlier.

Event Number: 012

Event Name: Job was held

Event Description: The user has paused the job, perhaps with the *condor_hold* command. It was stopped, and will go back into the queue again until it is aborted or released.

Event Number: 013

Event Name: Job was released

Event Description: The user is requesting that a job on hold be re-run.

Event Number: 014

Event Name: Parallel node executed

Event Description: A parallel (MPI) program is running on a node.

Event Number: 015

Event Name: Parallel node terminated

Event Description: A parallel (MPI) program has completed on a node.

Event Number: 016

Event Name: POST script terminated

Event Description: A node in a DAGMan work flow has a script that should be run after a job. The script is run on the submit host. This event signals that the post script has completed.

Event Number: 017

Event Name: Job submitted to Globus

Event Description: A grid job has been delegated to Globus (version 2, 3, or 4).

Event Number: 018

Event Name: Globus submit failed

Event Description: The attempt to delegate a job to Globus failed.

Event Number: 019

Event Name: Globus resource up

Event Description: The Globus resource that a job wants to run on was unavailable, but is now available.

Event Number: 020

Event Name: Detected Down Globus Resource

Event Description: The Globus resource that a job wants to run on has become unavailable.

Event Number: 021

Event Name: Remote error

Event Description: The *condor_starter* (which monitors the job on the execution machine) has failed.

Event Number: 022

Event Name: Remote system call socket lost

Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) have lost contact.

Event Number: 023

Event Name: Remote system call socket reestablished

Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) have been able to resume contact before the job lease expired.

Event Number: 024

Event Name: Remote system call reconnect failure

Event Description: The *condor_shadow* and *condor_starter* (which communicate while the job runs) were unable to resume contact before the job lease expired.

Event Number: 025

Event Name: Grid Resource Back Up

Event Description: A grid resource that was previously unavailable is now available.

Event Number: 026

Event Name: Detected Down Grid Resource

Event Description: The grid resource that a job is to run on is unavailable.

Event Number: 027

Event Name: Job submitted to grid resource

Event Description: A job has been submitted, and is under the auspices of the grid resource.

Event Number: 028

Event Name: Job ad information event triggered.

Event Description: Extra job ad attributes. This event is written as a supplement to other events when the configuration parameter `EVENT_LOG_JOB_AD_INFORMATION_ATTRS` is set.

2.6.7 Job Completion

When your Condor job completes (either through normal means or abnormal termination by signal), Condor will remove it from the job queue (i.e., it will no longer appear in the output of *condor_q*) and insert it into the job history file. You can examine the job history file with the *condor_history* command. If you specified a log file in your submit description file, then the job exit status will be recorded there as well.

By default, Condor will send you an email message when your job completes. You can modify this behavior with the *condor_submit* “notification” command. The message will include the exit status of your job (i.e., the argument your job passed to the exit system call when it completed) or notification that your job was killed by a signal. It will also include the following statistics (as appropriate) about your job:

Submitted at: when the job was submitted with *condor_submit*

Completed at: when the job completed

Real Time: elapsed time between when the job was submitted and when it completed (days hours:minutes:seconds)

Run Time: total time the job was running (i.e., real time minus queuing time)

Committed Time: total run time that contributed to job completion (i.e., run time minus the run time that was lost because the job was evicted without performing a checkpoint)

Remote User Time: total amount of committed time the job spent executing in user mode

Remote System Time: total amount of committed time the job spent executing in system mode

Total Remote Time: total committed CPU time for the job

Local User Time: total amount of time this job’s *condor_shadow* (remote system call server) spent executing in user mode

Local System Time: total amount of time this job’s *condor_shadow* spent executing in system mode

Total Local Time: total CPU usage for this job’s *condor_shadow*

Leveraging Factor: the ratio of total remote time to total system time (a factor below 1.0 indicates that the job ran inefficiently, spending more CPU time performing remote system calls than actually executing on the remote machine)

Virtual Image Size: memory size of the job, computed when the job checkpoints

Checkpoints written: number of successful checkpoints performed by the job

Checkpoint restarts: number of times the job successfully restarted from a checkpoint

Network: total network usage by the job for checkpointing and remote system calls

Buffer Configuration: configuration of remote system call I/O buffers

Total I/O: total file I/O detected by the remote system call library

I/O by File: I/O statistics per file produced by the remote system call library

Remote System Calls: listing of all remote system calls performed (both Condor-specific and Unix system calls) with a count of the number of times each was performed

2.7 Priorities and Preemption

Condor has two independent priority controls: *job* priorities and *user* priorities.

2.7.1 Job Priority

Job priorities allow the assignment of a priority level to each submitted Condor job in order to control order of execution. To set a job priority, use the *condor_prio* command — see the example in section 2.6.4, or the command reference page on page 748. Job priorities do not impact user priorities in any fashion. A job priority can be any integer, and higher values are “better”.

2.7.2 User priority

Machines are allocated to users based upon a user’s priority. A lower numerical value for user priority means higher priority, so a user with priority 5 will get more resources than a user with priority 50. User priorities in Condor can be examined with the *condor_userprio* command (see page 838). Condor administrators can set and change individual user priorities with the same utility.

Condor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ratio between user priorities. For example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources the individual is using. Each user starts out with the best possible priority: 0.5. If the number of machines a user currently has is greater than the user priority, the user priority will worsen by numerically increasing over time. If the number of machines is less than the priority, the priority will improve by numerically decreasing over time. The long-term result is fair-share access across all users. The speed at which Condor adjusts the priorities is controlled with the configuration macro `PRIORITY_HALFLIFE`, an exponential half-life value. The default is one day. If a user that has user priority of 100 and is utilizing 100 machines removes all his/her jobs, one day later that user’s priority will be 50, and two days later the priority will be 25.

Condor enforces that each user gets his/her fair share of machines according to user priority both when allocating machines which become available and by priority preemption of currently allocated machines. For instance, if a low priority user is utilizing all available machines and suddenly a higher priority user submits jobs, Condor will immediately checkpoint and vacate jobs belonging to the lower priority user. This will free up machines that Condor will then give over to the higher priority user. Condor will not starve the lower priority user; it will preempt only enough jobs so that the higher priority user’s fair share can be realized (based upon the ratio between user priorities). To prevent thrashing of the system due to priority preemption, the Condor site administrator can define a `PREEMPTION_REQUIREMENTS` expression in Condor’s configuration. The default expression that ships with Condor is configured to only preempt lower priority jobs that have run for at least one hour. So in the previous example, in the worse case it could take up to a maximum of one hour until the higher priority user receives his fair share of machines. For a general discussion of limiting

preemption, please see section 3.5.9 of the Administrator's manual.

User priorities are keyed on "username@domain", for example "johndoe@cs.wisc.edu". The domain name to use, if any, is configured by the Condor site administrator. Thus, user priority and therefore resource allocation is not impacted by which machine the user submits from or even if the user submits jobs from multiple machines.

An extra feature is the ability to submit a job as a *nice* job (see page 817). Nice jobs artificially boost the user priority by one million just for the nice job. This effectively means that nice jobs will only run on machines that no other Condor job (that is, non-niced job) wants. In a similar fashion, a Condor administrator could set the user priority of any specific Condor user very high. If done, for example, with a guest account, the guest could only use cycles not wanted by other users of the system.

2.7.3 Details About How Condor Jobs Vacate Machines

When Condor needs a job to vacate a machine for whatever reason, it sends the job an asynchronous signal specified in the `KillSig` attribute of the job's `ClassAd`. The value of this attribute can be specified by the user at submit time by placing the **kill_sig** option in the Condor submit description file.

If a program wanted to do some special work when required to vacate a machine, the program may set up a signal handler to use a trappable signal as an indication to clean up. When submitting this job, this clean up signal is specified to be used with **kill_sig**. Note that the clean up work needs to be quick. If the job takes too long to go away, Condor follows up with a `SIGKILL` signal which immediately terminates the process.

A job that is linked using *condor_compile* and is subsequently submitted into the standard universe, will checkpoint and exit upon receipt of a `SIGTSTP` signal. Thus, `SIGTSTP` is the default value for `KillSig` when submitting to the standard universe. The user's code may still checkpoint itself at any time by calling one of the following functions exported by the Condor libraries:

`ckpt()` Performs a checkpoint and then returns.

`ckpt_and_exit()` Checkpoints and exits; Condor will then restart the process again later, potentially on a different machine.

For jobs submitted into the vanilla universe, the default value for `KillSig` is `SIGTERM`, the usual method to nicely terminate a Unix program.

2.8 Java Applications

Condor allows users to access a wide variety of machines distributed around the world. The Java Virtual Machine (JVM) provides a uniform platform on any machine, regardless of the machine's

architecture or operating system. The Condor Java universe brings together these two features to create a distributed, homogeneous computing environment.

Compiled Java programs can be submitted to Condor, and Condor can execute the programs on any machine in the pool that will run the Java Virtual Machine.

The *condor_status* command can be used to see a list of machines in the pool for which Condor can use the Java Virtual Machine.

```
% condor_status -java
```

Name	JavaVendor	Ver	State	Activity	LoadAv	Mem	ActvtyTime
coral.cs.wisc	Sun	Microsy 1.2.2	Unclaimed	Idle	0.000	511	0+02:28:04
doc.cs.wisc.e	Sun	Microsy 1.2.2	Unclaimed	Idle	0.000	511	0+01:05:04
dsonokwa.cs.w	Sun	Microsy 1.2.2	Unclaimed	Idle	0.000	511	0+01:05:04
...							

If there is no output from the *condor_status* command, then Condor does not know the location details of the Java Virtual Machine on machines in the pool, or no machines have Java correctly installed. In this case, contact your system administrator or see section 3.13 for more information on getting Condor to work together with Java.

2.8.1 A Simple Example Java Application

Here is a complete, if simple, example. Start with a simple Java program, *Hello.java*:

```
public class Hello {
    public static void main( String [] args ) {
        System.out.println("Hello, world!\n");
    }
}
```

Build this program using your Java compiler. On most platforms, this is accomplished with the command

```
javac Hello.java
```

Submission to Condor requires a submit description file. If submitting where files are accessible using a shared file system, this simple submit description file works:

```
#####
#
# Example 1
```

```
# Execute a single Java class
#
#####

universe      = java
executable    = Hello.class
arguments     = Hello
output        = Hello.output
error         = Hello.error
queue
```

The Java universe must be explicitly selected.

The main class of the program is given in the **executable** statement. This is a file name which contains the entry point of the program. The name of the main class (not a file name) must be specified as the first argument to the program.

If submitting the job where a shared file system is *not* accessible, the submit description file becomes:

```
#####
#
# Example 1
# Execute a single Java class,
# not on a shared file system
#
#####

universe      = java
executable    = Hello.class
arguments     = Hello
output        = Hello.output
error         = Hello.error
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
queue
```

For more information about using Condor's file transfer mechanisms, see section 2.5.4.

To submit the job, where the submit description file is named `Hello.cmd`, execute

```
condor_submit Hello.cmd
```

To monitor the job, the commands `condor_q` and `condor_rm` are used as with all jobs.

2.8.2 Less Simple Java Specifications

Specifying more than 1 class file. For programs that consist of more than one `.class` file, identify the files in the submit description file:

```
executable = Stooges.class
transfer_input_files = Larry.class,Curly.class,Moe.class
```

The **executable** command does not change. It still identifies the class file that contains the program's entry point.

JAR files. If the program consists of a large number of class files, it may be easier to collect them all together into a single Java Archive (JAR) file. A JAR can be created with:

```
% jar cvf Library.jar Larry.class Curly.class Moe.class Stooges.class
```

Condor must then be told where to find the JAR as well as to use the JAR. The JAR file that contains the entry point is specified with the **executable** command. All JAR files are specified with the **jar_files** command. For this example that collected all the class files into a single JAR file, the submit description file contains:

```
executable = Library.jar
jar_files = Library.jar
```

Note that the JVM must know whether it is receiving JAR files or class files. Therefore, Condor must also be informed, in order to pass the information on to the JVM. That is why there is a difference in submit description file commands for the two ways of specifying files (**transfer_input_files** and **jar_files**).

If there are multiple JAR files, the **executable** command specifies the JAR file that contains the program's entry point. This file is also listed with the **jar_files** command:

```
executable = sortmerge.jar
jar_files = sortmerge.jar,statemap.jar
```

Using a third-party JAR file. As Condor requires that all JAR files (third-party or not) be available, specification of a third-party JAR file is no different than other JAR files. If the sortmerge example above also relies on version 2.1 from <http://jakarta.apache.org/commons/lang/>, and this JAR file has been placed in the same directory with the other JAR files, then the submit description file contains

```
executable = sortmerge.jar
jar_files = sortmerge.jar,statemap.jar,commons-lang-2.1.jar
```

An executable JAR file. When the JAR file is an executable, specify the program's entry point in the **arguments** command:

```
executable = anexecutable.jar
jar_files  = anexecutable.jar
arguments  = some.main.ClassFile
```

Packages. An example of a Java class that is declared in a non-default package is

```
package hpc;

public class CondorDriver
{
    // class definition here
}
```

The JVM needs to know the location of this package. It is passed as a command-line argument, implying the use of the naming convention and directory structure.

Therefore, the submit description file for this example will contain

```
arguments = hpc.CondorDriver
```

JVM-version specific features. If the program uses Java features found only in certain JVMs, then the Java application submitted to Condor must only run on those machines within the pool that run the needed JVM. Inform Condor by adding a `requirements` statement to the submit description file. For example, to require version 3.2, add to the submit description file:

```
requirements = (JavaVersion=="3.2")
```

Benchmark speeds. Each machine with Java capability in a Condor pool will execute a benchmark to determine its speed. The benchmark is taken when Condor is started on the machine, and it uses the SciMark2 (<http://math.nist.gov/scimark2>) benchmark. The result of the benchmark is held as an attribute within the machine ClassAd. The attribute is called `JavaMFlops`. Jobs that are run under the Java universe (as all other Condor jobs) may prefer or require a machine of a specific speed by setting `rank` or `requirements` in the submit description file. As an example, to execute only on machines of a minimum speed:

```
requirements = (JavaMFlops>4.5)
```

JVM options. Options to the JVM itself are specified in the submit description file:

```
java_vm_args = -DMyProperty=Value -verbose:gc
```

These options are those which go after the java command, but before the user's main class. Do not use this to set the classpath, as Condor handles that itself. Setting these options is useful for setting system properties, system assertions and debugging certain kinds of problems.

2.8.3 Chirp I/O

If a job has more sophisticated I/O requirements that cannot be met by Condor's file transfer mechanism, then the Chirp facility may provide a solution. Chirp has two advantages over simple, whole-file transfers. First, it permits the input files to be decided upon at run-time rather than submit time, and second, it permits partial-file I/O with results than can be seen as the program executes. However, small changes to the program are required in order to take advantage of Chirp. Depending on the style of the program, use either Chirp I/O streams or UNIX-like I/O functions.

Chirp I/O streams are the easiest way to get started. Modify the program to use the objects `ChirpInputStream` and `ChirpOutputStream` instead of `FileInputStream` and `FileOutputStream`. These classes are completely documented in the Condor Software Developer's Kit (SDK). Here is a simple code example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

    public static void main( String args[] ) {

        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    new ChirpInputStream("input")));

            PrintWriter out = new PrintWriter(
                new OutputStreamWriter(
                    new ChirpOutputStream("output")));

            while(true) {
                String line = in.readLine();
                if(line==null) break;
                out.println(line);
            }
            out.close();
        } catch( IOException e ) {
            System.out.println(e);
        }
    }
}
```

To perform UNIX-like I/O with Chirp, create a `ChirpClient` object. This object supports familiar operations such as `open`, `read`, `write`, and `close`. Exhaustive detail of the methods may be found in the Condor SDK, but here is a brief example:

```

import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

    public static void main( String args[] ) {

        try {
            ChirpClient client = new ChirpClient();
            String message = "Hello, world!\n";
            byte [] buffer = message.getBytes();

            // Note that we should check that actual==length.
            // However, skip it for clarity.

            int fd = client.open("output","wct",0777);
            int actual = client.write(fd,buffer,0,buffer.length);
            client.close(fd);

            client.rename("output","output.new");
            client.unlink("output.new");

        } catch( IOException e ) {
            System.out.println(e);
        }
    }
}

```

Regardless of which I/O style, the Chirp library must be specified and included with the job. The Chirp JAR (Chirp.jar) is found in the lib directory of the Condor installation. Copy it into your working directory in order to compile the program after modification to use Chirp I/O.

```

% condor_config_val LIB
/usr/local/condor/lib
% cp /usr/local/condor/lib/Chirp.jar .

```

Rebuild the program with the Chirp JAR file in the class path.

```

% javac -classpath Chirp.jar:. TestChirp.java

```

The Chirp JAR file must be specified in the submit description file. Here is an example submit description file that works for both of the given test programs:

```

universe = java

```

```
executable = TestChirp.class
arguments = TestChirp
jar_files = Chirp.jar
queue
```

2.9 Parallel Applications (Including MPI Applications)

Condor's Parallel universe supports a wide variety of parallel programming environments, and it encompasses the execution of MPI jobs. It supports jobs which need to be co-scheduled. A co-scheduled job has more than one process that must be running at the same time on different machines to work correctly. The parallel universe supersedes the mpi universe. The mpi universe eventually will be removed from Condor.

2.9.1 Prerequisites to Running Parallel Jobs

Condor must be configured such that resources (machines) running parallel jobs are dedicated. Note that *dedicated* has a very specific meaning in Condor: dedicated machines never vacate their executing Condor jobs, should the machine's interactive owner return. This is implemented by running a single dedicated scheduler process on a machine in the pool, which becomes the single machine from which parallel universe jobs are submitted. Once the dedicated scheduler claims a dedicated machine for use, the dedicated scheduler will try to use that machine to satisfy the requirements of the queue of parallel universe or MPI universe jobs. If the dedicated scheduler cannot use a machine for a configurable amount of time, it will release its claim on the machine, making it available again for the opportunistic scheduler.

Since Condor does not ordinarily run this way, (Condor usually uses opportunistic scheduling), dedicated machines must be specially configured. Section 3.12.8 of the Administrator's Manual describes the necessary configuration and provides detailed examples.

To simplify the scheduling of dedicated resources, a single machine becomes the scheduler of dedicated resources. This leads to a further restriction that jobs submitted to execute under the parallel universe must be submitted from the machine acting as the dedicated scheduler.

2.9.2 Parallel Job Submission

Given correct configuration, parallel universe jobs may be submitted from the machine running the dedicated scheduler. The dedicated scheduler claims machines for the parallel universe job, and invokes the job when the correct number of machines of the correct platform (architecture and operating system) are claimed. Note that the job likely consists of more than one process, each to be executed on a separate machine. The first process (machine) invoked is treated different than the others. When this first process exits, Condor shuts down all the others, even if they have not yet completed their execution.

An overly simplified submit description file for a parallel universe job appears as

```
#####
##  submit description file for a parallel program
#####
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
queue
```

This job specifies the **universe** as **parallel**, letting Condor know that dedicated resources are required. The **machine_count** command identifies the number of machines required by the job.

When submitted, the dedicated scheduler allocates eight machines with the same architecture and operating system as the submit machine. It waits until all eight machines are available before starting the job. When all the machines are ready, it invokes the */bin/sleep* command, with a command line argument of 30 on all eight machines more or less simultaneously.

A more realistic example of a parallel job utilizes other features.

```
#####
## Parallel example submit description file
#####
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
queue
```

The specification of the **input**, **output**, and **error** files utilize the predefined macro `$(NODE)`. See the *condor_submit* manual page on page 795 for further description of predefined macros. The `$(NODE)` macro is given a unique value as processes are assigned to machines. The `$(NODE)` value is fixed for the entire length of the job. It can therefore be used to identify individual aspects of the computation. In this example, it is used to utilize and assign unique names to input and output files.

This example presumes a shared file system across all the machines claimed for the parallel universe job. Where no shared file system is either available or guaranteed, use Condor's file transfer mechanism, as described in section 2.5.4 on page 26. This example uses the file transfer mechanism.

```
#####
```

```
## Parallel example submit description file
## without using a shared file system
#####
universe = parallel
executable = /bin/cat
log = logfile
input = infile.%(NODE)
output = outfile.%(NODE)
error = errfile.%(NODE)
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
queue
```

The job requires exactly four machines, and queues four processes. Each of these processes requires a correctly named input file, and produces an output file.

2.9.3 Parallel Jobs with Separate Requirements

The different machines executing for a parallel universe job may specify different machine requirements. A common example requires that the head node execute on a specific machine. It may be also useful for debugging purposes.

Consider the following example.

```
#####
## Example submit description file
## with multiple procs
#####
universe = parallel
executable = example
machine_count = 1
requirements = ( machine == "machine1" )
queue

requirements = ( machine != "machine1" )
machine_count = 3
queue
```

The dedicated scheduler allocates four machines. All four executing jobs have the same value for `$(Cluster)` macro. The `$(Process)` macro takes on two values; the value 0 will be assigned for the single executable that must be executed on machine1, and the value 1 will be assigned for the other three that must be executed anywhere but on machine1.

Carefully consider the ordering and nature of multiple sets of requirements in the same submit description file. The scheduler matches jobs to machines based on the ordering within the submit description file. Mutually exclusive requirements eliminate the dependence on ordering within the submit description file. Without mutually exclusive requirements, the scheduler may be unable to schedule the job. The ordering within the submit description file may preclude the scheduler considering the specific allocation that could satisfy the requirements.

2.9.4 MPI Applications Within Condor's Parallel Universe

MPI applications utilize a single executable that is invoked in order to execute in parallel on one or more machines. Condor's parallel universe provides the environment within which this executable is executed in parallel. However, the various implementations of MPI (for example, LAM or MPICH) require further framework items within a system-wide environment. Condor supports this necessary framework through user visible and modifiable scripts. An MPI implementation-dependent script becomes the Condor job. The script sets up the extra, necessary framework, and then invokes the MPI application's executable.

Condor provides these scripts in the `$(RELEASE_DIR)/etc/examples` directory. The script for the LAM implementation is `lamscrip`t. The script for the MPICH implementation is `mpiscrip`t. Therefore, a Condor submit description file for these implementations would appear similar to:

```
#####
## Example submit description file
## for MPICH 1 MPI
## works with MPICH 1.2.4, 1.2.5 and 1.2.6
#####
universe = parallel
executable = mpiscript
arguments = my_mpich_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_mpich_linked_executable
queue
```

or

```
#####
## Example submit description file
## for LAM MPI
#####
universe = parallel
executable = lamscrip
```

```
arguments = my_lam_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_lam_linked_executable
queue
```

The **executable** is the MPI implementation-dependent script. The first argument to the script is the MPI application's executable. Further arguments to the script are the MPI application's arguments. Condor must transfer this executable; do this with the **transfer_input_files** command.

For other implementations of MPI, copy and modify one of the given scripts. Most MPI implementations require two system-wide prerequisites. The first prerequisite is the ability to run a command on a remote machine without being prompted for a password. *ssh* is commonly used, but other command may be used. The second prerequisite is an ASCII file containing the list of machines that may utilize *ssh*. These common prerequisites are implemented in a further script called *sshd.sh*. *sshd.sh* generates *ssh* keys (to enable password-less remote execution), and starts an *sshd* daemon. The machine name and MPI rank are given to the submit machine.

The *sshd.sh* script requires the definition of two Condor configuration variables. Configuration variable `CONDOR_SSHD` is an absolute path to an implementation of *sshd*. *sshd.sh* has been tested with *openssh* version 3.9, but should work with more recent versions. Configuration variable `CONDOR_SSH_KEYGEN` points to the corresponding *ssh-keygen* executable.

Scripts *lamscript* and *mpi1script* each have their own idiosyncrasies. In *mpi1script*, the `PATH` to the MPICH installation must be set. The shell variable `MPDIR` indicates its proper value. This directory contains the MPICH *mpirun* executable. For LAM, there is a similar path setting, but it is called `LAMDIR` in the *lamscript* script. In addition, this path must be part of the path set in the user's `.cshrc` script. As of this writing, the LAM implementation does not work if the user's login shell is the Bourne or compatible shell.

These MPI jobs operate as all parallel universe jobs do. The default policy is that when the first node exits, the whole job is considered done, and Condor kills all other running nodes in that parallel job. Alternatively, a parallel universe job that sets the attribute

```
+ParallelShutdownPolicy = "WAIT_FOR_ALL"
```

in its submit description file changes the policy, such that Condor will wait until every node in the parallel job has completed to consider the job finished.

2.9.5 Outdated Documentation of the MPI Universe

The following sections on implementing MPI applications utilizing the MPI universe are superseded by the sections describing MPI applications utilizing the parallel universe. These sections are included in the manual as reference, until the time when the MPI universe is no longer supported within Condor.

MPI stands for Message Passing Interface. It provides an environment under which parallel programs may synchronize, by providing communication support. Running the MPI-based parallel programs within Condor eases the programmer's effort. Condor dedicates machines for running the programs, and it does so using the same interface used when submitting non-MPI jobs.

The MPI universe in Condor currently supports MPICH versions 1.2.2, 1.2.3, and 1.2.4 using the `ch_p4` device. The MPI universe does not support MPICH version 1.2.5. These supported implementations are offered by Argonne National Labs without charge by download. See the web page at <http://www-unix.mcs.anl.gov/mpi/mpich/> for details and availability. Programs to be submitted for execution under Condor will have been compiled using *mpicc*. No further compilation or linking is necessary to run jobs under Condor.

The Parallel universe 2.9 is now the preferred way to run MPI jobs. Support for the MPI universe will be removed from Condor at a future date.

MPI Details of Set Up

Administratively, Condor must be configured such that resources (machines) running MPI jobs are dedicated. Dedicated machines never vacate their running condor jobs should the machine's interactive owner return. Once the dedicated scheduler claims a dedicated machine for use, it will try to use that machine to satisfy the requirements of the queue of MPI jobs.

Since Condor is not ordinarily used in this manner (Condor uses opportunistic scheduling), machines that are to be used as dedicated resources must be configured as such. Section 3.12.8 of Administrator's Manual describes the necessary configuration and provides detailed examples.

To simplify the dedicated scheduling of resources, a single machine becomes the scheduler of dedicated resources. This leads to a further restriction that jobs submitted to execute under the MPI universe (with dedicated machines) must be submitted from the machine running as the dedicated scheduler.

MPI Job Submission

Once the programs are written and compiled, and Condor resources are correctly configured, jobs may be submitted. Each Condor job requires a submit description file. The simplest submit description file for an MPI job:

```
#####
##  submit description file for mpi_program
#####
universe = MPI
executable = mpi_program
machine_count = 4
queue
```

This job specifies the `universe` as `mpi`, letting Condor know that dedicated resources will be required. The `machine_count` command identifies the number of machines required by the job. The four machines that run the program will default to be of the same architecture and operating system as the machine on which the job is submitted, since a platform is not specified as a requirement.

The simplest example does not specify an input or output, meaning that the computation completed is useless, since both input comes from and the output goes to `/dev/null`. A more complex example of a submit description file utilizes other features.

```
#####
## MPI example submit description file
#####
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
queue
```

The specification of the input, output, and error files utilize a predefined macro that is only relevant to `mpi` universe jobs. See the *condor_submit* manual page on page 795 for further description of predefined macros. The `$(NODE)` macro is given a unique value as programs are assigned to machines. This value is what the MPICH version `ch_p4` implementation terms the rank of a program. Note that this term is unrelated and independent of the Condor term rank. The `$(NODE)` value is fixed for the entire length of the job. It can therefore be used to identify individual aspects of the computation. In this example, it is used to give unique names to input and output files.

If your site does NOT have a shared file system across all the nodes where your MPI computation will execute, you can use Condor's file transfer mechanism. You can find out more details about these settings by reading the *condor_submit* man page or section 2.5.4 on page 26. Assuming your job only reads input from STDIN, here is an example submit file for a site without a shared file system:

```
#####
## MPI example submit description file
## without using a shared file system
#####
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
```

```

machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
queue

```

Consider the following C program that uses this example submit description file.

```

/*****
 * simplempi.c
 *****/
#include <stdio.h>
#include "mpi.h"

int main(argc,argv)
    int argc;
    char *argv[];
{
    int myid;
    char line[128];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    fprintf ( stdout, "Printing to stdout...%d\n", myid );
    fprintf ( stderr, "Printing to stderr...%d\n", myid );
    fgets ( line, 128, stdin );
    fprintf ( stdout, "From stdin: %s", line );

    MPI_Finalize();
    return 0;
}

```

Here is a makefile that works with the example. It would build the MPI executable, using the MPICH version ch_p4 implementation.

```

#####
## This is a very basic Makefile                                     ##
#####

# the location of the MPICH compiler
CC          = /usr/local/bin/mpicc
CLINKER     = $(CC)

CFLAGS      = -g
EXECS       = simplempi

```

```
all: $(EXECS)

simplempi: simplempi.o
        $(CLINKER) -o simplempi simplempi.o -lm

.c.o:
        $(CC) $(CFLAGS) -c $.c
```

The submission to Condor requires exactly four machines, and queues four programs. Each of these programs requires an input file (correctly named) and produces an output file.

If input file for $$(NODE) = 0$ (called `infile.0`) contains

```
Hello number zero.
```

and the input file for $$(NODE) = 1$ (called `infile.1`) contains

```
Hello number one.
```

then after the job is submitted to Condor, there will be eight files created: `errfile.[0-3]` and `outfile.[0-3]`. `outfile.0` will contain

```
Printing to stdout...0
From stdin: Hello number zero.
```

and `errfile.0` will contain

```
Printing to stderr...0
```

Different nodes for an MPI job can have different machine requirements. For example, often the first node, sometimes called the head node, needs to run on a specific machine. This can be also useful for debugging. Condor accomodates this by supporting multiple queue statements in the submit file, much like with the other universes. For example:

```
#####
## MPI example submit description file
## with multiple procs
#####
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
```

```
error = errfile.$(NODE)
machine_count = 1
should_transfer_files = yes
when_to_transfer_output = on_exit
requirements = ( machine == "machine1" )
queue

requirements = ( machine != "machine1" )
machine_count = 3
queue
```

The dedicated scheduler will allocate four machines (nodes) total in two procs for this job. The first proc has one node, (rank 0 in MPI terms) and will run on the machine named machine1. The other three nodes, in the second proc, will run on other machines. Like in the other condor universes, the second requirements command overwrites the first, but the other commands are inherited from the first proc.

When submitting jobs with multiple requirements, it is best to write the requirements to be mutually exclusive, or to have the most selective requirement first in the submit file. This is because the scheduler tries to match jobs to machine in submit file order. If the requirements are not mutually exclusive, it can happen that the scheduler may be unable to schedule the job, even if all needed resources are available.

2.10 DAGMan Applications

A directed acyclic graph (DAG) can be used to represent a set of computations where the input, output, or execution of one or more computations is dependent on one or more other computations. The computations are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Condor finds machines for the execution of programs, but it does not schedule programs based on dependencies. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for the execution of programs (computations). DAGMan submits the programs to Condor in an order represented by a DAG and processes the results. A DAG input file describes the DAG, and further submit description file(s) are used by DAGMan when submitting programs to run under Condor.

DAGMan is itself executed as a scheduler universe job within Condor. As DAGMan submits programs, it monitors log file(s) to enforce the ordering required within the DAG. DAGMan is also responsible for scheduling, recovery, and reporting on the set of programs submitted to Condor.

2.10.1 DAGMan Terminology

To DAGMan, a node in a DAG may encompass more than a single program submitted to run under Condor. Figure 2.2 illustrates the elements of a node.

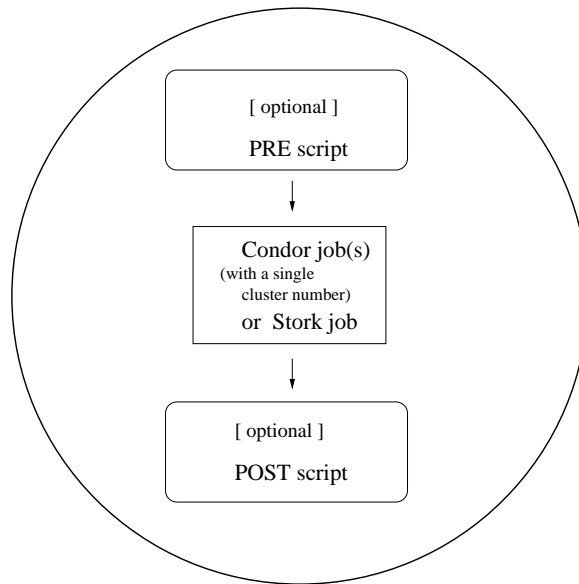


Figure 2.2: One Node within a DAG

At one time, the number of Condor jobs per node was restricted to one. This restriction is now relaxed such that all Condor jobs within a node must share a single cluster number. See the *condor_submit* manual page for a further definition of a cluster. A limitation exists such that all jobs within the single cluster must use the same log file.

As DAGMan schedules and submits jobs within nodes to Condor, these jobs are defined to succeed or fail based on their return values. This success or failure is propagated in well-defined ways to the level of a node within a DAG. Further progression of computation (towards completing the DAG) may be defined based upon the success or failure of one or more nodes.

The failure of a single job within a cluster of multiple jobs (within a single node) causes the entire cluster of jobs to fail. Any other jobs within the failed cluster of jobs are immediately removed. Each node within a DAG is further defined to succeed or fail, based upon the return values of a PRE script, the job(s) within the cluster, and/or a POST script.

2.10.2 Input File Describing the DAG

The input file used by DAGMan is called a DAG input file. All items are optional, but there must be at least one *JOB* or *DATA* item.

Comments may be placed in the DAG input file. The pound character (#) as the first character on a line identifies the line as a comment. Comments do not span lines.

A simple diamond-shaped DAG, as shown in Figure 2.3 is presented as a starting point for

examples. This DAG contains 4 nodes.

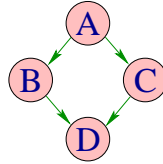


Figure 2.3: Diamond DAG

A very simple DAG input file for this diamond-shaped DAG is

```
# File name: diamond.dag
#
JOB A A.condor
JOB B B.condor
JOB C C.condor
JOB D D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

A set of basic key words appearing in a DAG input file is described below.

- **JOB**

The *JOB* key word specifies a job to be managed by Condor. The syntax used for each *JOB* entry is

JOB *JobName* *SubmitDescriptionFileName* [**DIR** *directory*] [**DONE**]

A *JOB* entry maps a *JobName* to a Condor submit description file. The *JobName* uniquely identifies nodes within the DAGMan input file and in output messages. Note that the name for each node within the DAG must be unique.

The key words *JOB* and *DONE* are not case sensitive. Therefore, *DONE*, *Done*, and *done* are all equivalent. The values defined for *JobName* and *SubmitDescriptionFileName* are case sensitive, as file names in the Unix file system are case sensitive. The *JobName* can be any string that contains no white space, except for the strings *PARENT* and *CHILD* (in upper, lower, or mixed case).

The *DIR* option specifies a working directory for this node, from which the Condor job will be submitted, and from which a *PRE* and/or *POST* script will be run. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*. A rescue DAG generated by a DAG run with the *-usedagdir* argument will contain *DIR* specifications, so the rescue DAG must be run *without* the *-usedagdir* argument.

The optional *DONE* identifies a job as being already completed. This is useful in situations where the user wishes to verify results, but does not need all programs within the dependency graph to be executed. The *DONE* feature is also utilized when an error occurs causing the

DAG to be aborted without completion. DAGMan generates a Rescue DAG, a DAG input file that can be used to restart and complete a DAG without re-executing completed nodes.

- **DATA**

The *DATA* key word specifies a job to be managed by the Stork data placement server. The syntax used for each *DATA* entry is

DATA *JobName SubmitDescriptionFileName* [**DIR** *directory*] [**DONE**]

A *DATA* entry maps a *JobName* to a Stork submit description file. In all other respects, the *DATA* key word is identical to the *JOB* key word.

Here is an example of a simple DAG that stages in data using Stork, processes the data using Condor, and stages the processed data out using Stork. Depending upon the implementation, multiple data jobs to stage in data or to stage out data may be run in parallel.

```
DATA    STAGE_IN1  stage_in1.stork
DATA    STAGE_IN2  stage_in2.stork
JOB     PROCESS    process.condor
DATA    STAGE_OUT1 stage_out1.stork
DATA    STAGE_OUT2 stage_out2.stork
PARENT  STAGE_IN1  STAGE_IN2 CHILD PROCESS
PARENT  PROCESS    CHILD STAGE_OUT1 STAGE_OUT2
```

- **SCRIPT**

The *SCRIPT* key word specifies processing that is done either before a job within the DAG is submitted to Condor or Stork for execution or after a job within the DAG completes its execution. Processing done before a job is submitted to Condor or Stork is called a *PRE* script. Processing done after a job completes its execution under Condor or Stork is called a *POST* script. A node in the DAG is comprised of the job together with *PRE* and/or *POST* scripts.

PRE and *POST* script lines within the DAG input file use the syntax:

SCRIPT PRE *JobName ExecutableName* [*arguments*]

SCRIPT POST *JobName ExecutableName* [*arguments*]

The *SCRIPT* key word identifies the type of line within the DAG input file. The *PRE* or *POST* key word specifies the relative timing of when the script is to be run. The *JobName* specifies the node to which the script is attached. The *ExecutableName* specifies the script to be executed, and it may be followed by any command line arguments to that script. The *ExecutableName* and optional *arguments* are case sensitive; they have their case preserved.

Scripts are optional for each job, and any scripts are executed on the machine from which the DAG is submitted; this is not necessarily the same machine upon which the node's Condor or Stork job is run. Further, a single cluster of Condor jobs may be spread across several machines.

A *PRE* script is commonly used to place files in a staging area for the cluster of jobs to use. A *POST* script is commonly used to clean up or remove files once the cluster of jobs is finished running. An example uses *PRE* and *POST* scripts to stage files that are stored on tape. The *PRE* script reads compressed input files from the tape drive, and it uncompresses them, placing

the input files in the current directory. The cluster of Condor jobs reads these input files, and produces output files. The POST script compresses the output files, writes them out to the tape, and then removes both the staged input files and the output files.

DAGMan takes note of the exit value of the scripts as well as the job. A script with an exit value not equal to 0 fails. If the PRE script fails, then neither the job nor the POST script runs, and the node fails.

If the PRE script succeeds, the Condor or Stork job is submitted. If the job fails and there is no POST script, the DAG node is marked as failed. An exit value not equal to 0 indicates program failure. It is therefore important that a successful program return the exit value 0.

If the job fails and there is a POST script, node failure is determined by the exit value of the POST script. A failing value from the POST script marks the node as failed. A succeeding value from the POST script (even with a failed job) marks the node as successful. Therefore, the POST script may need to consider the return value from the job.

By default, the POST script is run regardless of the job's return value.

A node not marked as failed at any point is successful. Table 2.1 summarizes the success or failure of an entire node for all possibilities. An *S* stands for success, an *F* stands for failure, and the dash character (-) identifies that there is no script.

PRE	-	-	F	S	S	-	-	-	-	S	S	S	S
JOB	S	F	not run	S	F	S	S	F	F	S	F	F	S
POST	-	-	not run	-	-	S	F	S	F	S	S	F	F
node	S	F	F	S	F	S	F	S	F	S	S	F	F

Table 2.1: Node success or failure definition

Two variables may be used within the DAG input file, and may ease script writing. The variables are often utilized in the arguments passed to a PRE or POST script. The variable `$JOB` evaluates to the (case sensitive) string defined for *JobName*. For use as an argument to POST scripts, the `$RETURN` variable evaluates to the return value of the Condor or Stork job. A job that dies due to a signal is reported with a `$RETURN` value representing the negative signal number. For example, SIGKILL (signal 9) is reported as -9. A job whose batch system submission fails is reported as -1001. A job that is externally removed from the batch system queue (by something other than *condor_dagman*) is reported as -1002.

As an example, consider the diamond-shaped DAG example. Suppose the PRE script expands a compressed file needed as input to nodes B and C. The file is named of the form *JobName.gz*. The DAG input file becomes

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
```

```
JOB D D.condor
SCRIPT PRE B pre.csh $JOB .gz
SCRIPT PRE C pre.csh $JOB .gz
PARENT A CHILD B C
PARENT B C CHILD D
```

The script `pre.csh` uses the arguments to form the file name of the compressed file:

```
#!/bin/csh
gunzip $argv[1]$argv[2]
```

• PARENT ...CHILD

The *PARENT* and *CHILD* key words specify the dependencies within the DAG. Nodes are parents and/or children within the DAG. A parent node must be completed successfully before any of its children may be started. A child node may only be started once all its parents have successfully completed.

The syntax of a dependency line within the DAG input file:

PARENT *ParentJobName*. . . **CHILD** *ChildJobName*. . .

The *PARENT* key word is followed by one or more *ParentJobNames*. The *CHILD* key word is followed by one or more *ChildJobNames*. Each child job depends on every parent job within the line. A single line in the input file can specify the dependencies from one or more parents to one or more children. As an example, the line

```
PARENT p1 p2 CHILD c1 c2
```

produces four dependencies:

1. p1 to c1
2. p1 to c2
3. p2 to c1
4. p2 to c2

2.10.3 Submit Description File

Each node in a DAG may use a unique submit description file. One key limitation is that each Condor submit description file must submit jobs described by a single cluster number. At the present time DAGMan cannot deal with a submit file producing multiple job clusters.

At one time, DAGMan required that all jobs within all nodes specify the same, single log file. This is no longer the case. However, if the DAG utilizes a large number of separate log files, performance may suffer. Therefore, it is better to have fewer, or even only a single log file. Unfortunately, each Stork job currently requires a separate log file. DAGMan enforces the dependencies within a DAG using the events recorded in the log file(s) produced by job submission to Condor.

Here is a modified version of the DAG input file for the diamond-shaped DAG. The modification has each node use the same submit description file.

```
# File name: diamond.dag
#
JOB A diamond_job.condor
JOB B diamond_job.condor
JOB C diamond_job.condor
JOB D diamond_job.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Here is the single Condor submit description file for this DAG:

```
# File name: diamond_job.condor
#
executable = /path/diamond.exe
output      = diamond.out.$(cluster)
error       = diamond.err.$(cluster)
log         = diamond_condor.log
universe    = vanilla
notification = NEVER
queue
```

This example uses the same Condor submit description file for all the jobs in the DAG. This implies that each node within the DAG runs the same job. The `$(cluster)` macro produces unique file names for each job's output. As the Condor job within each node causes a separate job submission, each has a unique cluster number.

Notification is set to `NEVER` in this example. This tells Condor not to send e-mail about the completion of a job submitted to Condor. For DAGs with many nodes, this reduces or eliminates excessive numbers of e-mails.

2.10.4 Job Submission

A DAG is submitted using the program *condor_submit_dag*. See the manual page 825 for complete details. A simple submission has the syntax

```
condor_submit_dag DAGInputFileName
```

The diamond-shaped DAG example may be submitted with

```
condor_submit_dag diamond.dag
```

In order to guarantee recoverability, the DAGMan program itself is run as a Condor job. As such, it needs a submit description file. *condor_submit_dag* produces this needed submit description file, naming it by appending `.condor.sub` to the *DAGInputFileName*. This submit description file may be edited if the DAG is submitted with

```
condor_submit_dag -no_submit diamond.dag
```

causing *condor_submit_dag* to generate the submit description file, but not submit DAGMan to Condor. To submit the DAG, once the submit description file is edited, use

```
condor_submit diamond.dag.condor.sub
```

An optional argument to *condor_submit_dag*, *-maxjobs*, is used to specify the maximum number of batch jobs that DAGMan may submit at one time. It is commonly used when there is a limited amount of input file staging capacity. As a specific example, consider a case where each job will require 4 Mbytes of input files, and the jobs will run in a directory with a volume of 100 Mbytes of free space. Using the argument *-maxjobs 25* guarantees that a maximum of 25 jobs, using a maximum of 100 Mbytes of space, will be submitted to Condor and/or Stork at one time.

While the *-maxjobs* argument is used to limit the number of batch system jobs submitted at one time, it may be desirable to limit the number of scripts running at one time. The optional *-maxpre* argument limits the number of PRE scripts that may be running at one time, while the optional *-maxpost* argument limits the number of POST scripts that may be running at one time.

An optional argument to *condor_submit_dag*, *-maxidle*, is used to limit the number of idle jobs within a given DAG. When the number of idle node jobs in the DAG reaches the specified value, *condor_dagman* will stop submitting jobs, even if there are ready nodes in the DAG. Once some of the idle jobs start to run, *condor_dagman* will resume submitting jobs. Note that this parameter only limits the number of idle jobs submitted by a given instance of *condor_dagman*. Idle jobs submitted by other sources (including other *condor_dagman* runs) are ignored.

2.10.5 Job Monitoring, Job Failure, and Job Removal

After submission, the progress of the DAG can be monitored by looking at the log file(s), observing the e-mail that job submission to Condor causes, or by using *condor_q -dag*. There is a large amount of information in an extra file. The name of this extra file is produced by appending *.dagman.out* to *DAGInputFileName*; for example, if the DAG file is *diamond.dag*, this extra file is *diamond.dag.dagman.out*. If this extra file grows too large, limit its size with the *MAX_DAGMAN_LOG* configuration macro (see section 3.3.4).

If you have some kind of problem in your DAGMan run, please save the corresponding *dagman.out* file; it is the most important debugging tool for DAGMan. As of version 6.8.2, the *dagman.out* is appended to, rather than overwritten, with each new DAGMan run.

condor_submit_dag attempts to check the DAG input file. If a problem is detected, *condor_submit_dag* prints out an error message and aborts.

To remove an entire DAG, consisting of DAGMan plus any jobs submitted to Condor or Stork, remove the DAGMan job running under Condor. *condor_q* will list the job number. Use the job number to remove the job, for example

```
% condor_q
-- Submitter: turunmaa.cs.wisc.edu : <128.105.175.125:36165> : turunmaa.cs.wisc.edu
ID      OWNER      SUBMITTED  RUN_TIME ST PRI SIZE CMD
  9.0    smoler      10/12 11:47  0+00:01:32 R  0   8.7  condor_dagman -f -
 11.0    smoler      10/12 11:48  0+00:00:00 I  0   3.6  B.out
 12.0    smoler      10/12 11:48  0+00:00:00 I  0   3.6  C.out

    3 jobs; 2 idle, 1 running, 0 held

% condor_rm 9.0
```

Before the DAGMan job stops running, it uses *condor_rm* and/or *stork_rm* to remove any jobs within the DAG that are running.

In the case where a machine is scheduled to go down, DAGMan will clean up memory and exit. However, it will leave any submitted jobs in Condor's queue.

2.10.6 Advanced Features of DAGMan

Retrying Failed Nodes

The *RETRY* key word provides a way to retry failed nodes. The use of retry is optional. The syntax for retry is

RETRY *JobName NumberOfRetries [UNLESS-EXIT value]*

where *JobName* identifies the node. *NumberOfRetries* is an integer number of times to retry the node after failure. The implied number of retries for any node is 0, the same as not having a retry line in the file. Retry is implemented on nodes, not parts of a node.

The diamond-shaped DAG example may be modified to retry node C:

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
```

If node C is marked as failed (for any reason), then it is started over as a first retry. The node will be tried a second and third time, if it continues to fail. If the node is marked as successful, then further retries do not occur.

Retry of a node may be short circuited using the optional key word *UNLESS-EXIT* (followed by an integer exit value). If the node exits with the specified integer exit value, then no further processing will be done on the node.

The *ABORT-DAG-ON* key word provides a way to abort the entire DAG if a given node returns a specific exit code. The syntax for *ABORT-DAG-ON* is

ABORT-DAG-ON *JobName AbortExitValue* [**RETURN** *DAGReturnValue*]

If the node specified by *JobName* returns the specified *AbortExitValue*, the DAG is immediately aborted. A DAG abort differs from a node failure, in that a DAG abort causes all nodes within the DAG to be stopped immediately. This includes removing the jobs in nodes that are currently running. A node failure allows the DAG to continue running, until no more progress can be made due to dependencies.

An abort overrides node retries. If a node returns the abort exit value, the DAG is aborted, even if the node has retry specified.

When a DAG aborts, by default it exits with the node return value that caused the abort. This can be changed by using the optional *RETURN* key word along with specifying the desired *DAGReturnValue*. The DAG abort return value can be used for DAGs within DAGs, allowing an inner DAG to cause an abort of an outer DAG.

Adding *ABORT-DAG-ON* for node C in the diamond-shaped DAG

```
# File name: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry C 3
ABORT-DAG-ON C 10 RETURN 1
```

causes the DAG to be aborted, if node C exits with a return value of 10. Any other currently running nodes (only node B is a possibility for this particular example) are stopped and removed. If this abort occurs, the return value for the DAG is 1.

Variable Values Associated with Nodes

The *VAR*s key word provides a method for defining a macro that can be referenced in the node's submit description file. These macros are defined on a per-node basis, using the following syntax:

VARs *JobName macroname*= "*string*" [**macroname**= "*string*". . .]

The macro may be used within the submit description file of the relevant node. A *macroname* consists of alphanumeric characters (a..Z and 0..9), as well as the underscore character. The space character delimits macros, when there is more than one macro defined for a node.

Correct syntax requires that the *string* must be enclosed in double quotes. To use a double quote inside *string*, escape it with the backslash character (\). To add the backslash character itself, use

two backslashes (\). The string \$(JOB) maybe used in *string* and will expand to *JobName*. If the **VARs** line appears in a DAG file used as a splice file, then \$(JOB) will be the fully scoped name of the node.

Note that macro names cannot begin with the string "queue" (in any combination of upper and lower case).

If the DAG input file contains

```
# File name: diamond.dag
#
JOB A A.condor
JOB B B.condor
JOB C C.condor
JOB D D.condor
VARs A state="Wisconsin"
PARENT A CHILD B C
PARENT B C CHILD D
```

then file A.condor may use the macro state. This example submit description file for the Condor job in node A passes the value of the macro as a command-line argument to the job.

```
# file name: A.condor
executable = A.exe
log         = A.log
error       = A.err
arguments   = $(state)
queue
```

This Condor job's command line will be

```
A.exe Wisconsin
```

The use of macros may allow a reduction in the necessary number of unique submit description files.

A separate example shows an intended use of a *VARs* entry in the DAG input file. This use may dramatically reduce the number of Condor submit description files needed for a DAG. In the case where the submit description file for each node varies only in file naming, the use of a substitution macro within the submit description file reduces the need to a single submit description file. Note that the user log file for a job currently cannot be specified using a macro passed from the DAG.

The example uses a single submit description file in the DAG input file, and uses the *Vars* entry to name output files.

The relevant portion of the DAG input file appears as

```
JOB A theonefile.sub
JOB B theonefile.sub
```

```
JOB C theonefile.sub
```

```

VARS A outfilename="A"
VARS B outfilename="B"
VARS C outfilename="C"

```

The submit description file appears as

```

# submit description file called:  theonefile.sub
executable   = progX
universe     = standard
output       = $(outfilename)
error        = error.$(outfilename)
log          = progX.log
queue

```

For a DAG such as this one, but with thousands of nodes, being able to write and maintain a single submit description file and a single, yet more complex, DAG input file is preferable.

Setting Priorities for Nodes

The *PRIORITY* key word assigns a priority to a DAG node. The syntax for *PRIORITY* is

PRIORITY *JobName PriorityValue*

The node priority affects the order in which nodes that are ready at the same time will be submitted. Note that node priority does *not* override the DAG dependencies.

Node priority is mainly relevant if node submission is throttled via the *-maxjobs* or *-maxidle* command-line flags or the *DAGMAN_MAX_JOBS_SUBMITTED* or *DAGMAN_MAX_JOBS_IDLE* configuration macros. Note that PRE scripts can affect the order in which jobs run, so DAGs containing PRE scripts may not run the nodes in exact priority order, even if doing so would satisfy the DAG dependencies.

The priority value is an integer (which can be negative). A larger numerical priority is better (will be run before a smaller numerical value). The default priority is 0.

Adding *PRIORITY* for node C in the diamond-shaped DAG

```

# File name: diamond.dag
#
JOB A A.condor
JOB B B.condor
JOB C C.condor
JOB D D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry C 3
PRIORITY C 1

```

This will cause node C to be submitted before node B (normally, node B would be submitted first).

Limiting the Number of Submitted Job Clusters within a DAG

In order to limit the number of submitted job clusters within a DAG, the nodes may be placed into categories by assignment of a name. Then, a maximum number of submitted clusters may be specified for each category.

The *CATEGORY* key word assigns a category name to a DAG node. The syntax for *CATEGORY* is

CATEGORY *JobName CategoryName*

Category names cannot contain white space.

The *MAXJOBS* key word limits the number of submitted job clusters on a per category basis. The syntax for *MAXJOBS* is

MAXJOBS *CategoryName MaxJobsValue*

If the number of submitted job clusters for a given category reaches the limit, no further job clusters in that category will be submitted until other job clusters within the category terminate. If *MAXJOBS* is not set for a defined category, then there is no limit placed on the number of submissions within that category.

Note that a single invocation of *condor_submit* results in one job cluster. The number of Condor jobs within a cluster may be greater than 1.

The configuration variable *DAGMAN_MAX_JOBS_SUBMITTED* and the *condor_submit_dag -maxjobs* command-line option are still enforced if these *CATEGORY* and *MAXJOBS* throttles are used.

Configuration Specific to a DAG

The *CONFIG* keyword specifies a configuration file to be used to set *condor_dagman* configuration options when running this DAG. The syntax for *CONFIG* is

CONFIG *ConfigFileName*

If the DAG file contains a line like this:

```
CONFIG dagman.config
```

the configuration values in the file *dagman.config* will be used for this DAG.

Configuration macros for *condor_dagman* can be specified in several ways, as given within the ordered list:

1. In a Condor configuration file.
2. With an environment variable. Prepend the string "`_CONDOR_`" to the macro name.
3. In a *condor_dagman*-specific configuration file specified in the DAG file or on the *condor_submit_dag* command line.
4. For some configuration macros, there is a corresponding *condor_submit_dag* command line flag (for example, `DAGMAN_MAX_JOBS_SUBMITTED`/`-maxjobs`).

In the above list, configuration values specified later in the list override ones specified earlier (e.g., a value specified on the *condor_submit_dag* command line overrides corresponding values in any configuration file; a value specified in a DAGMan-specific configuration file overrides values specified in a general Condor configuration file).

Non-*condor_dagman*, non-daemoncore configuration macros in a *condor_dagman*-specific configuration file are ignored.

Only a single configuration file can be specified for a given *condor_dagman* run. For example, if one file is specified in a DAG, and a different file is specified on the *condor_submit_dag* command line, this is a fatal error at submit time. The same is true if different configuration files are specified in multiple DAG files referenced in a single *condor_submit_dag* command.

If multiple DAGs are run in a single *condor_dagman* run, the configuration options specified in the *condor_dagman* configuration file, if any, apply to all DAGs, even if some of the DAGs specify no configuration file.

Configuration variables relating to DAGMan may be found in section 3.3.25.

Single Submission of Multiple, Independent DAGs

A single use of *condor_submit_dag* may execute multiple, independent DAGs. Each independent DAG has its own DAG input file. These DAG input files are command-line arguments to *condor_submit_dag* (see the *condor_submit_dag* manual page at 9).

Internally, all of the independent DAGs are combined into a single, larger DAG, with no dependencies between the original independent DAGs. As a result, any generated rescue DAG file represents all of the input DAGs as a single DAG. The file name of this rescue DAG is based on the DAG input file listed first within the command-line arguments to *condor_submit_dag* (unlike a single-DAG rescue DAG file, however, the file name will be `<whatever>.dagmulti.rescue` or `<whatever>.dagmulti.rescueNNN`, as opposed to just `<whatever>.dag.rescue` or `<whatever>.dag.rescueNNN`). Other files such as `dagman.out` and the lock file also have names based on this first DAG input file.

The success or failure of the independent DAGs is well defined. When multiple, independent DAGs are submitted with a single command, the success of the composite DAG is defined as the logical AND of the success of each independent DAG. This implies that failure is defined as the logical OR of the failure of any of the independent DAGs.

By default, DAGMan internally renames the nodes to avoid node name collisions. If all node names are unique, the renaming of nodes may be disabled by setting the configuration variable `DAGMAN_MUNGE_NODE_NAMES` to `False` (see 3.3.25).

A DAG Within a DAG Is a SUBDAG

The organization and dependencies of the jobs within a DAG are the keys to its utility. There are cases when a DAG is easier to visualize and construct hierarchically, in other words when a node within a DAG is also a DAG. Condor DAGMan handles this situation quite easily. (Note that DAGs can be nested to any depth.)

Since more than one DAG is being discussed, terminology is introduced to clarify which DAG is which. Reuse the example diamond-shaped DAG as given in Figure 2.3. Assume that node B of this diamond-shaped DAG will itself be a DAG. The DAG of node B is called the inner DAG, and the diamond-shaped DAG is called the outer DAG.

Work on the inner DAG first. Here is a very simple linear DAG input file used as an example of the inner DAG.

```
# File name: inner.dag
#
JOB  X  X.submit
JOB  Y  Y.submit
JOB  Z  Z.submit
PARENT X CHILD Y
PARENT Y CHILD Z
```

The Condor submit file corresponding to this DAG will be named `inner.dag.condor.sub`. (The DAGMan submit file is always named `<DAG file name>.condor.sub`.)

A simple example of a DAG input file for the outer DAG is

```
# File name: diamond.dag
#
JOB  A  A.submit
SUBDAG EXTERNAL B  inner.dag
JOB  C  C.submit
JOB  D  D.submit
PARENT A CHILD B C
PARENT B C CHILD D
```

This is equivalent, but the version above is now preferred:

```
# File name: diamond.dag
#
JOB   A   A.submit
JOB   B   inner.dag.condor.sub
JOB   C   C.submit
JOB   D   D.submit
PARENT A CHILD B C
PARENT B C CHILD D
```

Within the DAG input file, the *SUBDAG* keyword specifies a special case of a **JOB** node, where the job is a DAG.

SUBDAG EXTERNAL *JobName DagFileName* [**DIR** *directory*] [**DONE**]

A **SUBDAG** node is essentially the same as a "normal" node, except that the nested DAG file is specified instead of the Condor submit file. ("SUBDAG EXTERNAL A foo.dag" is functionally equivalent to "JOB A foo.dag.condor.sub", but SUBDAG EXTERNAL is now the preferred syntax for specifying such a node.) "EXTERNAL" means that the SUBDAG is run in its own instance of *condor_dagman*.

- *condor_submit_dag* recursion

The outer DAG is then submitted as before, with the command

```
condor_submit_dag diamond.dag
```

In Condor 7.1.4 and later, when you run *condor_submit_dag* on the outer DAG file, *condor_submit_dag -no_submit -update_submit* is automatically run on the inner DAG file before the outer DAG is actually run. (If you want to disable this feature, you can do so by passing the **-no_recurse** command-line flag to *condor_submit_dag*.)

The following command-line flags are passed to the lower-level *condor_submit_dag*:

- **-verbose**
- **-force**
- **-notification**
- **-dagman**
- **-debug**
- **-usedagdir**
- **-outfile_dir**
- **-oldrescue**
- **-autorescue**
- **-dorescuefrom**

- **-allowversionmismatch**

The following command-line flags are preserved in existing lower-level DAG submit files (if any exist):

- **-maxjobs**
- **-maxidle**
- **-maxpre**
- **-maxpost**

Note that the **-force** option will cause existing DAG submit files to be overwritten without preserving any existing values.

Because of the automatic recursion in *condor_submit_dag*, normally you only need to run *condor_submit_dag* on your outermost DAG. But you can manually run *condor_submit_dag* on an inner DAG or DAGs to set **-maxjobs** or other values. For instance, using the example in the previous section, you could do the following:

```
condor_submit_dag -no_submit -maxjobs 1 inner.dag
condor_submit_dag diamond.dag
```

This would set maxjobs to 1 for the inner DAG, and then run the entire work flow.

- Interaction with Rescue DAGs

When using nested DAGs, it is strongly recommended that you use "new-style" rescue DAGs (this is the default). Using "new-style" rescue DAGs will automatically run the proper rescue DAG(s) if there is a failure in your work flow. For example, if one of the nodes in *inner.dag* fails, this will produce a rescue DAG for *inner.dag* (*inner.dag.rescue.001*, etc.). Then, since *inner.dag* failed, node B of *diamond.dag* will fail, producing a rescue DAG for *diamond.dag* (*diamond.dag.rescue.001*, etc.). If you re-run *condor_submit_dag diamond.dag* the most recent outer rescue DAG will be run, and this will re-run the inner DAG, which will actually run the most recent inner rescue DAG. If you use "old-style" rescue DAGs, you would have to either rename the inner rescue DAG or run it manually.

- File Paths

Remember that, unless you use the DIR keyword in your outer DAG, the inner DAG will be submitted from the directory in which you run the outer DAG. Therefore, all paths in the inner DAG file (to submit files, etc.) must be specified accordingly.

DAG Splicing

A weakness in scalability exists when submitting a DAG within a DAG. Each executing independent DAG requires its own invocation of *condor_dagman* to be running. The scaling issue presents itself when the same semantic DAG is reused hundreds or thousands of times in a larger DAG. Further, there may be many rescue DAGs created if a problem occurs. To alleviate these concerns, the DAGMan language introduces the concept of graph splicing.

A splice is a named instance of a subgraph which is specified in a separate DAG file. The splice is treated as a whole entity during dependency specification in the including DAG. The same DAG file may be reused as differently named splices, each one incorporating a copy of the dependency graph (and nodes therein) into the including DAG. Any splice in an including DAG may have dependencies between the sets of initial and final nodes. A splice may be incorporated into an including DAG without any dependencies; it is considered a disjoint DAG within the including DAG. The nodes within a splice are scoped according to a hierarchy of names associated with the splices, as the splices are parsed from the top level DAG file. The scoping character to describe the inclusion hierarchy of nodes into the top level dag is ' + '. This character is chosen due to a restriction in the allowable characters which may be in a file name across the variety of ports that Condor supports. In any DAG file, all splices must have unique names, but the same splice name may be reused in different DAG files.

Condor does not detect nor support splices that form a cycle within the DAG. A DAGMan job that causes a cyclic inclusion of splices will eventually exhaust available memory and crash.

The *SPLICE* keyword in a DAG input file creates a named instance of a DAG as specified in another file as an entity which may have *PARENT* and *CHILD* dependencies associated with other splice names or node names in the including DAG file. The syntax for *SPLICE* is

SPLICE *SpliceName DagFileName* [**DIR** *directory*]

After parsing incorporates a splice, all nodes within the splice become nodes within the including DAG.

The following series of examples illustrate potential uses of splicing. To simplify the examples, presume that each and every job uses the same, simple Condor submit description file:

```
# BEGIN SUBMIT FILE submit.condor
executable    = /bin/echo
arguments     = OK
universe      = vanilla
output        = $(jobname).out
error         = $(jobname).err
log           = submit.log
notification  = NEVER
queue
# END SUBMIT FILE submit.condor
```

This first simple example splices a diamond-shaped DAG in between the two nodes of a top level DAG. Here is the DAG input file for the diamond-shaped DAG:

```
# BEGIN DAG FILE diamond.dag
JOB A submit.condor
VARS A jobname="$(JOB) "

JOB B submit.condor
```

```

VARS B jobname="$(JOB) "

JOB C submit.condor
VARS C jobname="$(JOB) "

JOB D submit.condor
VARS D jobname="$(JOB) "

PARENT A CHILD B C
PARENT B C CHILD D
# END DAG FILE diamond.dag

```

The top level DAG incorporates the diamond-shaped splice:

```

# BEGIN DAG FILE toplevel.dag
JOB X submit.condor
VARS X jobname="$(JOB) "

JOB Y submit.condor
VARS Y jobname="$(JOB) "

# This is an instance of diamond.dag, given the symbolic name DIAMOND
SPLICE DIAMOND diamond.dag

# Set up a relationship between the nodes in this dag and the splice

PARENT X CHILD DIAMOND
PARENT DIAMOND CHILD Y

# END DAG FILE toplevel.dag

```

Figure 2.4 illustrates the resulting top level DAG and the dependencies produced. Notice the naming of nodes scoped with the splice name. This hierarchy of splice names assures unique names associated with all nodes.

Figure 2.5 illustrates the starting point for a more complex example. The DAG input file `x.dag` describes this X-shaped DAG. The completed example displays more of the spatial constructs provided by splices. Pay particular attention to the notion that each named splice creates a new graph, even when the same DAG input file is specified.

```

# BEGIN DAG FILE x.dag

JOB A submit.condor
VARS A jobname="$(JOB) "

```

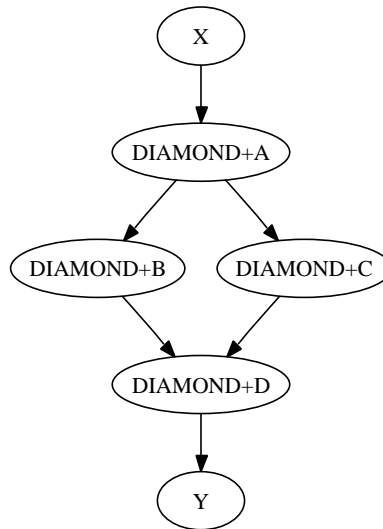


Figure 2.4: The diamond-shaped DAG spliced between two nodes.

```

JOB B submit.condor
VARS B jobname="$(JOB) "

JOB C submit.condor
VARS C jobname="$(JOB) "

JOB D submit.condor
VARS D jobname="$(JOB) "

JOB E submit.condor
VARS E jobname="$(JOB) "

JOB F submit.condor
VARS F jobname="$(JOB) "

JOB G submit.condor
VARS G jobname="$(JOB) "

# Make an X-shaped dependency graph
PARENT A B C CHILD D
PARENT D CHILD E F G

# END DAG FILE X.dag

```

File `s1.dag` continues the example, presenting the DAG input file that incorporates two sepa-

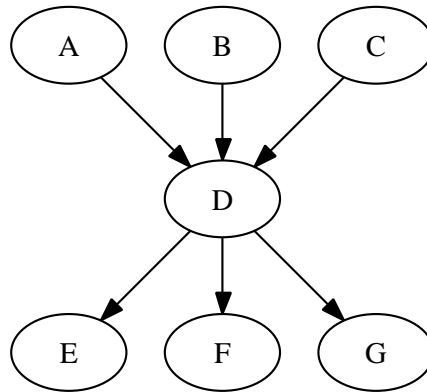


Figure 2.5: The X-shaped DAG.

rate splices of the X-shaped DAG. Figure 2.6 illustrates the resulting DAG.

```

# BEGIN DAG FILE s1.dag

JOB A submit.condor
VARS A jobname="$(JOB)"

JOB B submit.condor
VARS B jobname="$(JOB)"

# name two individual splices of the X-shaped DAG
SPLICE X1 X.dag
SPLICE X2 X.dag

# Define dependencies
# A must complete before the initial nodes in X1 can start
PARENT A CHILD X1
# All final nodes in X1 must finish before the initial nodes in X2 can begin
PARENT X1 CHILD X2
# All final nodes in X2 must finish before B may begin.
PARENT X2 CHILD B

# END DAG FILE s1.dag

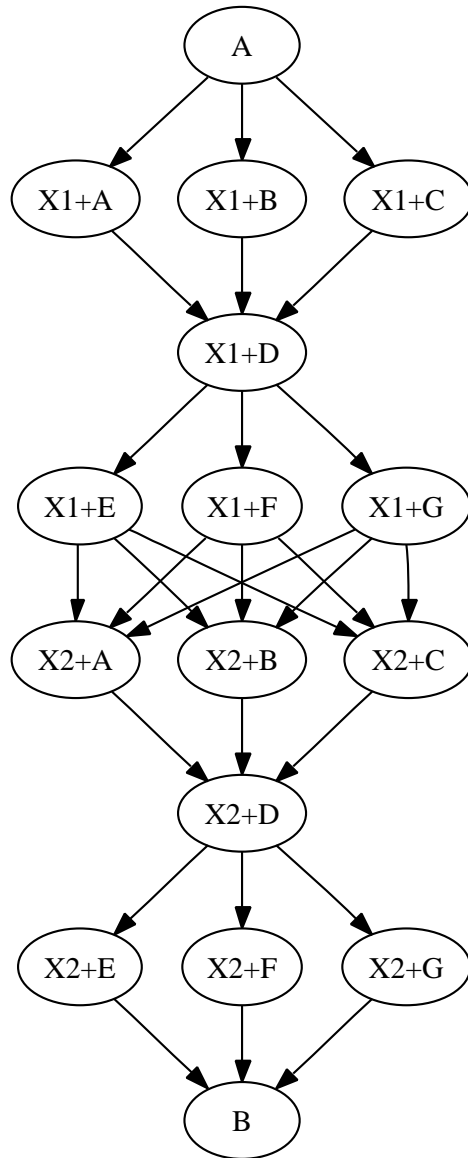
```

The top level DAG in the hierarchy of this complex example is described by the DAG input file `toplevel.dag`. Figure 2.7 illustrates the final DAG. Notice that the DAG has two disjoint graphs in it as a result of splice S3 not having any dependencies associated with it in this top level DAG.

```

# BEGIN DAG FILE topLevel.dag

```

Figure 2.6: The DAG described by `s1.dag`.

```

JOB A submit.condor
VARS A jobname="$(JOB)"

```

```

JOB B submit.condor
VARS B jobname="$(JOB)"

```

```
JOB C submit.condor
VARS C jobname="$(JOB) "

JOB D submit.condor
VARS D jobname="$(JOB) "

# a diamond-shaped DAG
PARENT A CHILD B C
PARENT B C CHILD D

# This splice of the X-shaped DAG can only run after
# the diamond dag finishes
SPLICE S2 X.dag
PARENT D CHILD S2

# Since there are no dependencies for S3,
# the following splice is disjoint
SPLICE S3 s1.dag

# END DAG FILE toplevel.dag
```

The *DIR* option specifies a working directory for a splice, from which the splice will be parsed and the containing jobs submitted. The directory associated with the splices' *DIR* specification will be propagated as a prefix to all nodes in the splice and any included splices. If a node already has a *DIR* specification, then the splice's *DIR* specification will be a prefix to the nodes and separated by a directory separator character. Jobs in included splices with an absolute path for their *DIR* specification will have their *DIR* specification untouched. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*. A rescue DAG generated by a DAG run with the *-usedagdir* argument will contain *DIR* specifications, so the rescue DAG must be run *without* the *-usedagdir* argument.

2.10.7 Job Recovery: The Rescue DAG

DAGMan can help with the resubmission of uncompleted portions of a DAG, when one or more nodes result in failure. If any node in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG.

The Rescue DAG is a DAG input file, functionally the same as the original DAG file. The Rescue DAG additionally contains an indication of successfully completed nodes by appending the *DONE* key word to the node's *JOB* or *DATA* lines. If the DAG is resubmitted utilizing the Rescue DAG, the successfully completed nodes will not be re-executed.

Note that if multiple DAG files are specified on the *condor_submit_dag* command line, a single rescue DAG encompassing all of the input DAGs is generated.

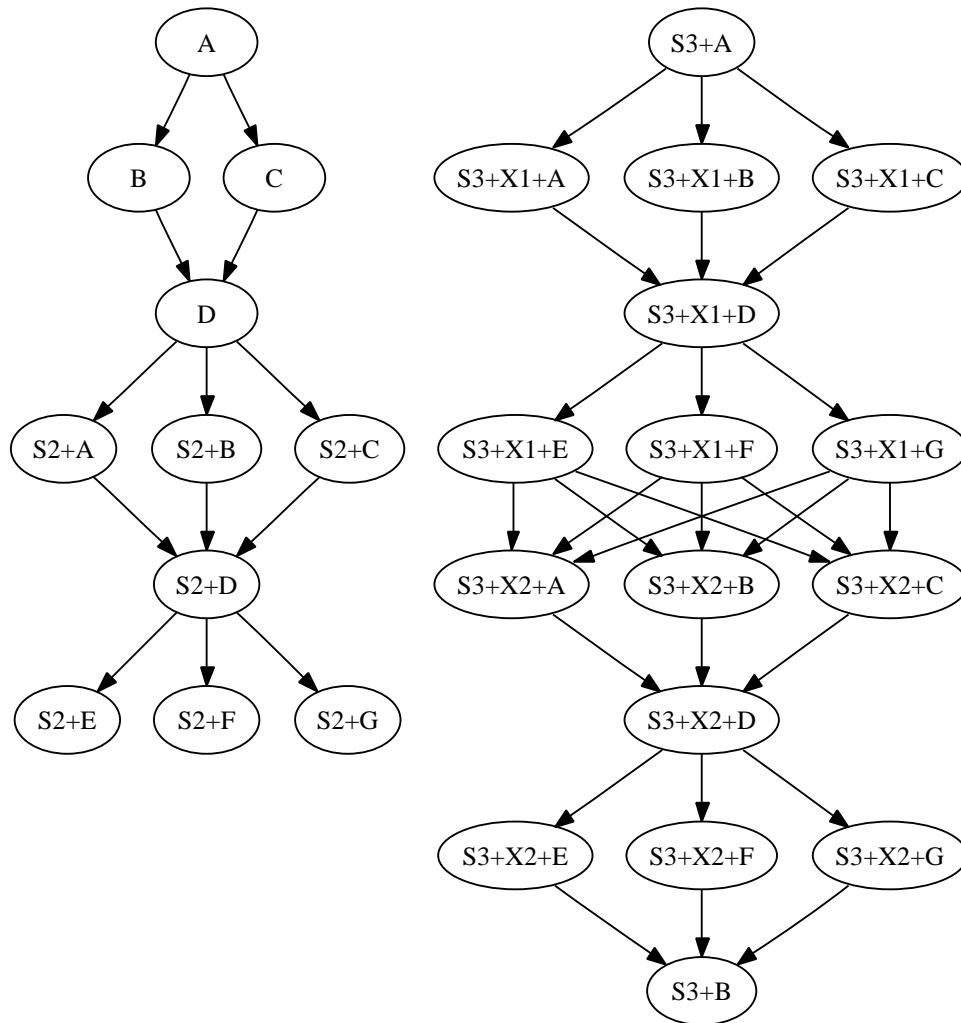


Figure 2.7: The complex splice example DAG.

If the Rescue DAG file is generated before all retries of a node are completed, then the Rescue DAG file will also contain *Retry* entries. The number of retries will be set to the appropriate remaining number of retries.

The granularity defining success or failure in the Rescue DAG is the node. For a node that fails, all parts of the node will be re-run, even if some parts were successful the first time. For example, if a node's PRE script succeeds, but then the node's Condor job cluster fails, the entire node, which includes the PRE script will be re-run. A job cluster may result in the submission of multiple Condor jobs. If one of the multiple jobs fails, the node fails. Therefore, a the Rescue DAG will re-run the entire node, implying the submission of the entire cluster of jobs, not just the one(s) that failed.

Statistics about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

The Rescue DAG is automatically generated by DAGMan when a node within the DAG fails or when *condor_dagman* itself is removed with *condor_rm*. The file name of the Rescue DAG, and usage of the Rescue DAG changed from explicit specification to implicit usage beginning with Condor version 7.1.0.

Current naming of the Rescue DAG appends the string `.rescue<XXX>` to the original DAG input file. Values for `<XXX>` start at 001 and continue to 002, 003, and beyond. If a Rescue DAG exists, the Rescue DAG with the largest magnitude value for `<XXX>` will be used, and its usage is implied.

Here is an example showing file naming and DAG submission for the case of a failed DAG. The initial DAG is submitted with

```
condor_submit_dag my.dag
```

A failure of this DAG results in the Rescue DAG called `my.dag.rescue001`. The DAG is resubmitted using the same command:

```
condor_submit_dag my.dag
```

This resubmission of the DAG uses the Rescue DAG file `my.dag.rescue001`, because it exists. Failure of this Rescue DAG results in another Rescue DAG called `my.dag.rescue002`. If the DAG is again submitted, using the same command as with the first two submissions, but not repeated here, then this third submission uses the Rescue DAG file `my.dag.rescue002`, because it exists, and because the value 002 is larger in magnitude than 001.

To explicitly specify a particular Rescue DAG, use the optional command-line argument *-dorescuefrom* with *condor_submit_dag*. Note that this will have the side effect of renaming existing Rescue DAG files with larger magnitude values of `<XXX>`. Each renamed file has its existing name appended with the string `.old`. For example, assume that `my.dag` has failed 4 times, resulting in the Rescue DAGs named `my.dag.rescue001`, `my.dag.rescue002`, `my.dag.rescue003`, and `my.dag.rescue004`. A decision is made to re-run using `my.dag.rescue002`. The submit command is

```
condor_submit_dag -dorescuefrom 2 my.dag
```

The DAG specified by the DAG input file `my.dag.rescue002` is submitted. And, the existing Rescue DAG `my.dag.rescue003` is renamed to be `my.dag.rescue003.old`, while the existing Rescue DAG `my.dag.rescue004` is renamed to be `my.dag.rescue004.old`.

A maximum value for XXX may be configured by the `DAGMAN_MAX_RESCUE_NUM` configuration macro (see 228).

Prior to Condor version 7.1.0, the naming of a Rescue DAG appended the string `.rescue` to the existing DAG input file name. And, the Rescue DAG file would be explicitly placed in the command line that submitted it. For example a first submission is

```
condor_submit_dag my.dag
```

Assuming that this DAG failed, the file `my.dag.rescue` would be created. To run this Rescue DAG, the submission command is

```
condor_submit_dag my.dag.rescue
```

If this Rescue DAG also failed, a new Rescue DAG named `my.dag.rescue.rescue` would be created.

The behavior of DAGMan with respect to Rescue DAGs can be forced to the old behavior by setting the configuration variables `DAGMAN_OLD_RESCUE` (see 228) to `True` and `DAGMAN_AUTO_RESCUE` (see 228) to `False`.

2.10.8 File Paths in DAGs

By default, *condor_dagman* assumes that all relative paths in a DAG input file and the associated Condor submit description files are relative to the current working directory when *condor_submit_dag* is run. Note that relative paths in submit description files can be modified by the submit command **initialdir**; see the *condor_submit* manual page at 9 for more details. The rest of this discussion ignores **initialdir**.

In most cases, path names relative to the current working directory is the desired behavior. However, if running multiple DAGs with a single *condor_dagman*, and each DAG is in its own directory, this will cause problems. In this case, use the *-usedagdir* command-line argument to *condor_submit_dag* (see the *condor_submit_dag* manual page at 9 for more details). This tells *condor_dagman* to run each DAG as if *condor_submit_dag* had been run in the directory in which the relevant DAG file exists.

For example, assume that a directory called `parent` contains two subdirectories called `dag1` and `dag2`, and that `dag1` contains the DAG input file `one.dag` and `dag2` contains the DAG input file `two.dag`. Further, assume that each DAG is set up to be run from its own directory with the following command:

```
cd dag1; condor_submit_dag one.dag
```

This will correctly run `one.dag`.

The goal is to run the two, independent DAGs located within `dag1` and `dag2` while the current working directory is `parent`. To do so, run the following command:

```
condor_submit_dag -usedagdir dag1/one.dag dag2/two.dag
```

Of course, if all paths in the DAG input file(s) and the relevant submit description files are absolute, the *-usedagdir* argument is not needed; however, using absolute paths is NOT generally a good idea.

If you *do not* use *-usedagdir*, relative paths can still work for multiple DAGs, if all file paths are given relative to the current working directory as *condor_submit_dag* is executed. However, this means that, if the DAGs are in separate directories, they cannot be submitted from their own directories, only from the parent directory the paths are set up for.

Note that if you use the *-usedagdir* argument, and your run results in a rescue DAG, the rescue DAG file will be written to the current working directory, and should be run from that directory. The rescue DAG includes all the path information necessary to run each node job in the proper directory.

2.10.9 Visualizing DAGs with *dot*

It can be helpful to see a picture of a DAG. DAGMan can assist you in visualizing a DAG by creating the input files used by the AT&T Research Labs *graphviz* package. *dot* is a program within this package, available from <http://www.graphviz.org/>, and it is used to draw pictures of DAGs.

DAGMan produces one or more dot files as the result of an extra line in a DAGMan input file. The line appears as

```
DOT dag.dot
```

This creates a file called *dag.dot*, which contains a specification of the DAG before any jobs within the DAG are submitted to Condor. The *dag.dot* file is used to create a visualization of the DAG by using this file as input to *dot*. This example creates a Postscript file, with a visualization of the DAG:

```
dot -Tps dag.dot -o dag.ps
```

Within the DAGMan input file, the DOT command can take several optional parameters:

- **UPDATE** This will update the dot file every time a significant update happens.
- **DONT-UPDATE** Creates a single dot file, when the DAGMan begins executing. This is the default if the parameter **UPDATE** is not used.
- **OVERWRITE** Overwrites the dot file each time it is created. This is the default, unless **DONT-OVERWRITE** is specified.
- **DONT-OVERWRITE** Used to create multiple dot files, instead of overwriting the single one specified. To create file names, DAGMan uses the name of the file concatenated a period and an integer. For example, the DAGMan input file line

```
DOT dag.dot DONT-OVERWRITE
```

causes files `dag.dot.0`, `dag.dot.1`, `dag.dot.2`, etc. to be created. This option is most useful combined with the **UPDATE** option to visualize the history of the DAG after it has finished executing.

- **INCLUDE** *path-to-filename* Includes the contents of a file given by *path-to-filename* in the file produced by the **DOT** command. The include file contents are always placed after the line of the form `label=`. This may be useful if further editing of the created files would be necessary, perhaps because you are automatically visualizing the DAG as it progresses.

If conflicting parameters are used in a DOT command, the last one listed is used.

2.11 Virtual Machine Applications

The **vm** universe facilitates a Condor job that matches and then lands a disk image on an execute machine within a Condor pool. This disk image is intended to be a virtual machine.

This section describes this Condor job. See section 3.3.28 for details of configuration variables.

2.11.1 The Submit Description File

Different than all other universe jobs, the **vm** universe job specifies a disk image, not an executable. Therefore, the submit commands **input**, **output**, and **error** do not apply. If specified, `condor_submit` rejects the job with an error. The **executable** command changes definition within a **vm** universe job. It no longer specifies an executable file, but instead provides a string that identifies the job for tools such as `condor_q`.

Use of the **args** command creates a file named `condor.args`, which is added to the set of CD-ROM files. The contents of this file are the arguments specified.

VMware and Xen virtual machine software are supported. As the two differ from each other, the submit description file specifies either

```
vm_type = vmware
```

or

```
vm_type = xen
```

The job specifies its memory needs for the disk image with **vm.memory**, which is given in Mbytes. Condor uses this number to assure a match with a machine that can provide the needed memory space.

A CD-ROM for the virtual machine is composed of a set of files. These files are specified in the submit description file with a comma-separated list of file names.

```
vm_cdrom_files = a.txt,b.txt,c.txt
```

Condor must also be told to transfer these files from the submit machine to the machine that will run the **vm** universe job with

```
vm_should_transfer_cdrom_files = YES
```

Creating a checkpoint is straightforward for a virtual machine, as a checkpoint is a set of files that represent a snapshot of both disk image and memory. The checkpoint is created and all files are transferred back to the \$(SPOOL) directory on the machine from which the job was submitted. **vm** universe jobs can not use a checkpoint server. The submit command to create checkpoints is

```
vm_checkpoint = true
```

Without this command, no checkpoints are created (by default).

Virtual machine networking is enabled with the command

```
vm_networking = true
```

And, when networking is enabled, a definition of **vm_networking_type** as **bridge** matches the job only with a machine that is configured to use bridge networking. A definition of **vm_networking_type** as **nat** matches the job only with a machine that is configured to use NAT networking. When no definition of **vm_networking_type** is given, Condor may match the job with a machine that enables networking, and further, the choice of bridge or NAT networking is determined by the machine's configuration.

A current limitation restricts the use of networking to **vm** universe jobs that do not create checkpoints such that the job may migrate to another machine.

When both checkpoints and networking are enabled, the job further specifies

```
when_to_transfer_output = ON_EXIT_OR_EVICT
```

Modified disk images are transferred back to the machine from which the job was submitted as the **vm** universe job completes. Job completion for a **vm** universe job occurs when the virtual machine is shut down, and Condor notices (as the result of a periodic check on the state of the virtual machine). Should the job not want any files transferred back (modified or not), for example because the job explicitly transferred its own files, the submit command to prevent the transfer is

```
vm_no_output_vm = true
```

Further commands specify information that is specific to the virtual machine type targeted.

VMware-Specific Submit Commands

Specific to VMware, the submit description file command **vmware_dir** gives the path and directory (on the machine from which the job is submitted) where VMware-specific files and applications reside. Examples of these VMware-specific applications are VMDK and VMX.

Condor must be told whether or not the contents of the **vmware_dir** directory must be transferred to the machine where the job is to be executed. This required information is given with the submit command **vmware_should_transfer_files**. With a value of `True`, Condor does transfer the contents of the directory. With a value of `False`, Condor does not transfer the contents of the directory, and instead presumes that access to this directory is available through a shared file system.

By default, Condor uses a snapshot disk for new and modified files. They may also be utilized for checkpoints. The snapshot disk is initially quite small, growing only as new files are created or files are modified. When **vmware_should_transfer_files** is `True`, a job may specify that a snapshot disk is *not* to be used with the command

```
vmware_snapshot_disk = False
```

In this case, Condor will utilize original disk files in producing checkpoints. Note that *condor_submit* issues an error message and does not submit the job if both **vmware_should_transfer_files** and **vmware_snapshot_disk** are `False`.

Note that if snapshot disks are requested and file transfer is not being used, the **vmware_dir** setting given in the submit file should not contain any symlink path components. This is to work around the issue discussed in the FAQ entry in section 7.3.

Xen-Specific Submit Commands

The required disk image must be identified for a Xen virtual machine. This **xen_disk** command specifies a list of comma-separated files. Each disk file is specified by 3 colon separated fields. The first field is the path and file name of the disk file. The second field specifies the device, and the third field specifies permissions. Here is an example that identifies two files:

```
xen_disk = /myxen/diskfile.img:sda1:w,/myxen/swap.img:sda2:w
```

If any files need to be transferred from the submit machine to the machine where the **vm** universe job will execute, Condor must be explicitly told to do so with the **xen_transfer_files** command:

```
xen_transfer_files = /myxen/diskfile.img,/myxen/swap.img
```

Any and all needed files on a system without a shared file system (between the submit machine and the machine where the job will execute) must be listed.

A Xen **vm** universe job requires specification of the guest kernel. The **xen_kernel** command accomplishes this, utilizing one of the following definitions.

1. **xen_kernel = any** tells Condor that the kernel is pre-staged, and its location is specified by configuration of the *condor_vm-gahp*.
2. **xen_kernel = included** implies that the kernel is to be found in disk image given by the definition of the single file specified in **xen_disk**.
3. **xen_kernel = path-to-kernel** gives a full path and file name of the required kernel. If this kernel must be transferred to machine on which the **vm** universe job will execute, it must also be included in the **xen_transfer_files** command.

This form of the **xen_kernel** command also requires further definition of the **xen_root** command. **xen_root** defines the device containing files needed by **root**.

Transfer of CD-ROM files under Xen requires the definition of the associated device in addition to the specification of the files. The submit description file contains

```
vm_cdrom_files = a.txt,b.txt,c.txt
vm_should_transfer_cdrom_files = YES
xen_cdrom_device = device-name
```

where the last line of this example defines the device.

2.11.2 Checkpoints

This section has not yet been written

2.11.3 Disk Images

VMware on Windows and Linux

Following the platform-specific guest OS installation instructions found at <http://pubs.vmware.com/guestnotes>, creates a VMware disk image.

Xen

This section has not yet been written

2.11.4 Job Completion in the vm Universe

Job completion for a **vm** universe job occurs when the virtual machine is shut down, and Condor notices (as the result of a periodic check on the state of the virtual machine). This is different from jobs executed under the environment of other universes.

Shut down of a virtual machine occurs from within the virtual machine environment. Under a Windows 2000, Windows XP, or Vista virtual machine, an administrator issues the command

```
shutdown -s -t 01
```

For older versions of Windows operating systems, directions are given at <http://www.aumha.org/win4/a/shutcut.php>.

Under a Linux virtual machine, the `root` user executes

```
/sbin/poweroff
```

The command `/sbin/halt` will not completely shut down some Linux distributions, and instead causes the job to hang.

Since the successful completion of the **vm** universe job requires the successful shut down of the virtual machine, it is good advice to try the shut down procedure outside of Condor, before a **vm** universe job is submitted.

2.12 Time Scheduling for Job Execution

Jobs may be scheduled to begin execution at a specified time in the future with Condor's job deferral functionality. All specifications are in a job's submit description file. Job deferral functionality is expanded to provide for the periodic execution of a job, known as the CronTab scheduling.

2.12.1 Job Deferral

Job deferral allows the specification of the exact date and time at which a job is to begin executing. Condor attempts to match the job to an execution machine just like any other job, however, the job will wait until the exact time to begin execution. A user can specify Condor to allow some flexibility to execute jobs that miss their execution time.

Deferred Execution Time

A job's deferral time is the exact time that Condor should attempt to execute the job. The deferral time attribute is defined as an expression that evaluates to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). This is the time that Condor will begin to execute the job.

After a job is matched and all of its files have been transferred to an execution machine, Condor checks to see if the job's ad contains a deferral time. If it does, Condor calculates the number

of seconds between the execution machine's current system time to the job's deferral time. If the deferral time is in the future, the job waits to begin execution. While a job waits, its job ClassAd attribute `JobStatus` indicates the job is running. As the deferral time arrives, the job begins to execute. If a job misses its execution time, that is, if the deferral time is in the past, the job is evicted from the execution machine and put on hold in the queue.

The specification of a deferral time does not interfere with Condor's behavior. For example, if a job is waiting to begin execution when a *condor_hold* command is issued, the job is removed from the execution machine and is put on hold. If a job is waiting to begin execution when a *condor_suspend* command is issued, the job continues to wait. When the deferral time arrives, Condor begins execution for the job, but immediately suspends it.

Missed Execution Window

If a job arrives at its execution machine after the deferral time passes, the job is evicted from the machine and put on hold in the job queue. This may occur, for example, because the transfer of needed files took too long due to a slow network connection. A deferral window permits the execution of a job that misses its deferral time by specifying a window of time within which the job may begin.

The deferral window is the number of seconds after the deferral time, within which the job may begin. When a job arrives too late, Condor calculates the difference in seconds between the execution machine's current time and the job's deferral time. If this difference is less than or equal to the deferral window, the job immediately begins execution. If this difference is greater than the deferral window, the job is evicted from the execution machine and is put on hold in the job queue.

Preparation Time

When a job defines a deferral time far in the future and then is matched to an execution machine, potential computation cycles are lost because the deferred job has claimed the machine, but is not actually executing. Other jobs could execute during the interval when the job waits for its deferral time. To make use of the wasted time, a job defines a **deferral_prep_time** with an integer expression that evaluates to a number of seconds. At this number of seconds before the deferral time, the job may be matched with a machine.

Usage Examples

Here are examples of how the job deferral time, deferral window, and the preparation time may be used.

The job's submit description file specifies that the job is to begin execution on January 1st, 2006 at 12:00 pm:

```
deferral_time = 1136138400
```

The Unix *date* program may be used to calculate a Unix epoch time. The syntax of the command to do this appears as

```
% date --date "MM/DD/YYYY HH:MM:SS" +%s
```

MM is a 2-digit month number, DD is a 2-digit day of the month number, and YYYY is a 4-digit year. HH is the 2-digit hour of the day, MM is the 2-digit minute of the hour, and SS are the 2-digit seconds within the minute. The characters +%s tell the *date* program to give the output as a Unix epoch time.

The job always waits 60 seconds before beginning execution:

```
deferral_time = (CurrentTime + 60)
```

In this example, assume that the deferral time is 45 seconds in the past as the job is available. The job begins execution, because 75 seconds remain in the deferral window:

```
deferral_window = 120
```

In this example, a job is scheduled to execute far in the future, on January 1st, 2010 at 12:00 pm. The **deferral_prep_time** attribute delays the job from being matched until 60 seconds before the job is to begin execution.

```
deferral_time      = 1262368800
deferral_prep_time = 60
```

Limitations

There are some limitations to Condor's job deferral feature.

- Job deferral is not available for scheduler universe jobs. A scheduler universe job defining the `deferral_time` produces a fatal error when submitted.
- The time that the job begins to execute is based on the execution machine's system clock, and not the submission machine's system clock. Be mindful of the ramifications when the two clocks show dramatically different times.
- A job's `JobStatus` attribute is always in the running state when job deferral is used. There is currently no way to distinguish between a job that is executing and a job that is waiting for its deferral time.

Submit Command	Allowed Values
cron_minute	0 - 59
cron_hour	0 - 23
cron_day_of_Month	1 - 31
cron_month	1 - 12
cron_day_of_week	0 - 7 (Sunday is 0 or 7)

Table 2.2: The list of submit commands and their value ranges.

2.12.2 CronTab Scheduling

Condor's CronTab scheduling functionality allows jobs to be scheduled to executed periodically. A job's execution schedule is defined by commands within the submit description file. The notation is much like that used by the Unix *cron* daemon. The scheduling of jobs using Condor's CronTab feature calculates and utilizes the `DeferralTime` ClassAd attribute.

Also, unlike the Unix *cron* daemon, Condor never runs more than one instance of a job at the same time.

The capability for repetitive or periodic execution of the job is enabled by specifying an **on_exit_remove** command for the job, such that the job does not leave the queue until desired.

Semantics for CronTab Specification

A job's execution schedule is defined by a set specifications within the submit description file. Condor uses these to calculate a `DeferralTime` for the job.

Table 2.2 lists the submit commands and acceptable values for these commands. At least one of these must be defined in order for Condor to calculate a `DeferralTime` for the job. Once one CronTab value is defined, the default for all the others uses all the values in the allowed values ranges.

The day of a job's execution can be specified by both the **cron_day_of_month** and the **cron_day_of_week** attributes. The day will be the logical or of both.

The semantics allow more than one value to be specified by using the `*` operator, ranges, lists, and steps (strides) within ranges.

The asterisk operator The `*` (asterisk) operator specifies that all of the allowed values are used for scheduling. For example,

```
cron_month = *
```

becomes any and all of the list of possible months: (1,2,3,4,5,6,7,8,9,10,11,12). Thus, a job runs any month in the year.

Ranges A range creates a set of integers from all the allowed values between two integers separated by a hyphen. The specified range is inclusive, and the integer to the left of the hyphen must be less than the right hand integer. For example,

```
cron_hour = 0-4
```

represents the set of hours from 12:00 am (midnight) to 4:00 am, or (0,1,2,3,4).

Lists A list is the union of the values or ranges separated by commas. Multiple entries of the same value are ignored. For example,

```
cron_minute = 15,20,25,30
cron_hour   = 0-3,9-12,15
```

cron_minute represents (15,20,25,30) and **cron_hour** represents (0,1,2,3,9,10,11,12,15).

Steps Steps select specific numbers from a range, based on an interval. A step is specified by appending a range or the asterisk operator with a slash character (/), followed by an integer value. For example,

```
cron_minute = 10-30/5
cron_hour   = */3
```

cron_minute specifies every five minutes within the specified range to represent (10,15,20,25,30). **cron_hour** specifies every three hours of the day to represent (0,3,6,9,12,15,18,21).

Preparation Time and Execution Window

The **cron_prep_time** command is analogous to the deferral time's **deferral_prep_time** command. It specifies the number of seconds before the deferral time that the job is to be matched and sent to the execution machine. This permits Condor to make necessary preparations before the deferral time occurs.

Consider the submit description file example that includes

```
cron_hour = *
cron_prep_time = 300
```

The job is scheduled to begin execution at the top of every hour. The job will be matched and sent to an execution machine no more than five minutes before the next deferral time. For example, if a job is submitted at 9:30am, then the next deferral time will be calculated to be 10:00am. Condor may attempt to match the job to a machine and send the job once it is 9:55am.

As the CronTab scheduling calculates and uses deferral time, jobs may also make use of the deferral window. The submit command **cron_window** is analogous to the submit command **deferral_window**. Consider the submit description file example that includes

```
cron_hour = *
cron_window = 360
```

As the previous example, the job is scheduled to begin execution at the top of every hour. Yet with no preparation time, the job is likely to miss its deferral time. The 6-minute window allows the job to begin execution, as long as it arrives and can begin within 6 minutes of the deferral time, as seen by the time kept on the execution machine.

Scheduling

When a job using the CronTab functionality is submitted to Condor, use of at least one of the submit description file commands beginning with `cron_` causes Condor to calculate and set a deferral time for when the job should run. A deferral time is determined based on the current time rounded later in time to the next minute. The deferral time is the job's `DeferralTime` attribute. A new deferral time is calculated when the job first enters the job queue, when the job is re-queued, or when the job is released from the hold state. New deferral times for *all* jobs in the job queue using the CronTab functionality are recalculated when a *condor_reconfig* or a *condor_restart* command that affects the job queue is issued.

A job's deferral time is not always the same time that a job will receive a match and be sent to the execution machine. This is because Condor operates on the job queue at times that are independent of job events, such as when job execution completes. Therefore, Condor may operate on the job queue just after a job's deferral time states that it is to begin execution. Condor attempts to start a job when the following pseudo-code boolean expression evaluates to True:

```
( CurrentTime + SCHEDD_INTERVAL ) >= ( DeferralTime - CronPrepTime )
```

If the `CurrentTime` plus the number of seconds until the next time Condor checks the job queue is greater than or equal to the time that the job should be submitted to the execution machine, then the job is to be matched and sent now.

Jobs using the CronTab functionality are not automatically re-queued by Condor after their execution is complete. The submit description file for a job must specify an appropriate **on_exit_remove** command to ensure that a job remains in the queue. This job maintains its original `ClusterId` and `ProcId`.

Usage Examples

Here are some examples of the submit commands necessary to schedule jobs to run at multifarious times. Please note that it is not necessary to explicitly define each attribute; the default value is `*`.

Run 23 minutes after every two hours, every day of the week:

```
on_exit_remove = false
cron_minute = 23
cron_hour = 0-23/2
cron_day_of_month = *
cron_month = *
cron_day_of_week = *
```

Run at 10:30pm on each of May 10th to May 20th, as well as every remaining Monday within the month of May:

```
on_exit_remove = false
cron_minute = 30
cron_hour = 20
cron_day_of_month = 10-20
cron_month = 5
cron_day_of_week = 2
```

Run on every 10 minutes and every 6 minutes before noon on January 18th with a 2-minute preparation time:

```
on_exit_remove = false
cron_minute = */10,*/6
cron_hour = 0-11
cron_day_of_month = 18
cron_month = 1
cron_day_of_week = *
cron_prep_time = 120
```

Limitations

The use of the CronTab functionality has all of the same limitations of deferral times, because the mechanism is based upon deferral times.

- It is impossible to schedule vanilla and standard universe jobs at intervals that are smaller than the interval at which Condor evaluates jobs. This interval is determined by the configuration variable `SCHEDD_INTERVAL`. As a vanilla or standard universe job completes execution and is placed back into the job queue, it may not be placed in the idle state in time. This problem does not afflict local universe jobs.
- Condor cannot guarantee that a job will be matched in order to make its scheduled deferral time. A job must be matched with an execution machine just as any other Condor job; if Condor is unable to find a match, then the job will miss its chance for executing and must wait for the next execution time specified by the CronTab schedule.

2.13 Stork Applications

Today's scientific applications have huge data requirements, which continue to increase drastically every year. These data are generally accessed by many users from all across the globe. This requires moving huge amounts of data around wide area networks to complete the computation cycle, which brings with it the problem of efficient and reliable data placement.

Stork is a scheduler for data placement. With Stork, *data placement jobs* have been elevated to the same level as Condor's computational jobs; data placements are queued, managed, queried and autonomously restarted upon error. Stork understands the semantics and protocols of data placement.

The underlying data placement jobs are performed by Stork *modules*, typically installed in the Condor `libexec` directory. The module name is encoded from the data placement type and functions. For example, the `stork.transfer.file-file` module transfers data from the `file:/` (local file system) to the `file:/` protocol. The `stork.transfer.file-file` module is the only module bundled with Condor/Stork. Additionally, contributed modules may be downloaded for these data transfer protocols:

<code>ftp://</code>	FTP File Transfer Protocol
<code>http://</code>	HTTP Hypertext Transfer Protocol
<code>gsiftp://</code>	Globus Grid FTP
<code>nest://</code>	Condor NeST network storage appliance (see http://www.cs.wisc.edu/condor/nest/)
<code>srb://</code>	SDSC Storage Resource Broker (SRB) (see http://www.sdsc.edu/srb/)
<code>srm://</code>	Storage Resource Manager (SRM) (see http://sdm.lbl.gov/srm-wg/)
<code>csmr://</code>	Castor Storage Resource Manager (Castor SRM) (see http://castor.web.cern.ch/castor/)
<code>unitree://</code>	NCSA UniTree (see http://www.ncsa.uiuc.edu/Divisions/CC/HPDM/unitree/)

The Stork module API is simple and extensible, enabling users to create and use their own modules.

Stork includes high level features for managing data transfers. By configuration, the number of active jobs running from a Stork server may be limited. Stork includes built in fault tolerance, with capabilities for retrying failed jobs, together with the specification of alternate protocols. Stork users also have access to a higher level job manager, Condor DAGMan (section 2.10), which can manage both Stork data placement jobs and traditional Condor jobs at the same time.

2.13.1 Submitting Stork Jobs

As with Condor jobs, Stork jobs are specified with a submit description file. It is important to note the syntax of the submit description file for a Stork job is different than that used by Condor jobs. Specifically, Stork submit description files are written in the ClassAd language. See the ClassAd Language Reference Manual for complete details. Please note that while most of Condor uses ClassAds, Stork utilizes the most recent version of this language, which has evolved over time. Stork defines keywords. When present in the job submit file, keywords define the function of the

job.

Here is sample Stork job submit description file, showing file syntax and keywords. A job specifies a 1-to-1 mapping of a data source URL to destination URL.

```
// This is a comment line.
[
    dap_type = transfer;
    src_url = "file:/etc/termcap";
    dest_url = "file:/tmp/stork/file-termcap";
]
```

This example shows the ClassAd pairs that form the heart of a Stork job specification. The minimum keywords required to specify a Stork job are:

dap_type Currently, the data type is constrained to **transfer**.

src_url Specify the data protocol and URL of the source.

dest_url Specify the data protocol and URL of the destination.

Additionally, the following keywords may be used in a Stork submit description file:

x509proxy Specifies the location of the X.509 proxy file for protocols that use GSI authentication, such as **gsiftp://**. The special value of *"default"* (quotes are required) invokes GSI libraries to search for the user credential in the standard locations.

alt_protocols A comma separated list of alternative protocol pairs (for source and destination protocols), used in a round robin fashion when transfers fail. See section 2.13.3 for a further discussion and examples.

Stork places no restriction on the submit file name or extension, and will accept any valid file name for a Stork submit description file.

Submit data placement jobs to Stork using the *stork_submit* tool. For example, after creating the submit description file *sample.stork* with an editor, submit the data transfer job with the command:

```
stork_submit sample.stork
```

Stork then returns the associated job id, which is used by other Stork job control tools.

Only the first ClassAd (a record expression within brackets) within a Stork submit description file becomes a data placement job upon submission. Other ClassAds within the file are ignored.

2.13.2 Managing Stork Jobs

Stork provides a set of command-line user tools for job management, including submitting, querying, and removing data placement jobs.

Querying Stork Jobs

Use *stork_status* to check the status of any active or completed Stork job. *stork_status* takes a single argument: the job id. For example, to check the status of the Stork job with job id 3:

```
stork_status 3
```

Use *stork_q* to query all active Stork jobs. *stork_q* does not report on completed Stork jobs.

For example, to check the status all active Stork jobs:

```
stork_q
```

Removing Stork Jobs

Active jobs may be removed from the job queue with the *stork_rm* tool. *stork_rm* takes a single argument: the job id of the job to remove. All jobs may be removed, provided they have not completed.

For example, to remove the queued job with job id 4:

```
stork_rm 4
```

2.13.3 Fault Tolerance

In an ideal world, all data transfers succeed on the first attempt. However, data transfers do fail for various reasons. Stork is designed with data transfer fault tolerance. Based on configuration, Stork retries failed data transfer jobs using specified protocols.

If a transfer fails, Stork attempts the transfer again, until the number of attempts reaches the limit, as defined by the configuration variable `STORK_MAX_RETRY` (section 3.3.33).

For each attempt at transfer, the transfer protocols to be used at both source and destination are defined. These transfer protocols may vary, when defined by an **alt_protocols** entry in the submit description file. The location of the data at the source and destination is unchanged by the **alt_protocols** entry. **alt_protocols** defines an ordered list of alternative translation protocols to be used. Each entry in the list is a pair. The first of the pair defines the protocol to be used at the

source of the transfer. The second of the pair defines the protocol to be used at the destination of the transfer.

The syntax is a comma-separated list of pairs. A dash character separated the pairs. The protocol name is given in all lower case letters, without colons or slash characters. Stork uses these strings to identify the protocol translation and transfer module to be used.

The initial translation protocol (specified in the **src_url** and **dest_url** entries) together with the list defined by an **alt_protocols** entry form the ordered list of protocols to be utilized in a round robin fashion.

For example, if STORK_MAX_RETRY has the value 4, and the Stork job submit description file contains

```
[
  dap_type = transfer;
  src_url = "gsiftp://serverA/dirA/fileA";
  dest_url = "http://serverB/dirB/fileB";
]
```

then Stork will attempt up to 4 transfers, with each using the same translation protocol. **gsiftp://** is used at the source, and **http://** is used at the destination. The Stork job fails if it has not been completed after 4 attempts.

A second example shows the transfer protocols used for each attempted transfer, when **alt_protocols** is used. For this example, assume that STORK_MAX_RETRY has the value 7.

```
[
  dap_type = transfer;
  src_url = "gsiftp://no-such-server/dir/file";
  dest_url = "file:/dir/file";
  alt_protocols = "ftp-file, http-file";
]
```

Stork attempts the following transfers, in the given order, stopping when the transfer succeeds.

1. from gsiftp://no-such-server/dir/file to file:/dir/file
2. from ftp://no-such-server/dir/file to file:/dir/file
3. from http://no-such-server/dir/file to file:/dir/file
4. from gsiftp://no-such-server/dir/file to file:/dir/file
5. from ftp://no-such-server/dir/file to file:/dir/file
6. from http://no-such-server/dir/file to file:/dir/file
7. from gsiftp://no-such-server/dir/file to file:/dir/file

2.13.4 Running Stork Jobs Under DAGMan

Condor DAGMan (section 2.10) provides high level management of both traditional CPU jobs and Stork data placement jobs. Using DAGMan, users can specify data placement using the *DATA* keyword. DAGMan can mix Stork data transfer jobs and Condor jobs. This capability lends itself well to grid computing, as data is often staged in (transferred) before processing the data. After processing, output is often staged out (transferred).

Here is a sample DAGMan input file that stages in input files using Stork transfers, processes the data as a Condor job, and stages out the result using a Stork transfer.

```
# Transfer input files using Stork
DATA INPUT1 transfer_input_data1.stork
DATA INPUT1 transfer_input_data2.stork

DATA INPUT2 transfer_data
#
# Process the data using Condor
JOB PROCESS process.condor
#
# Transfer output file using Stork
DATA RESULT transfer_result_data.stork
#
# Specify job dependencies
PARENT INPUT1 INPUT2 CHILD PROCESS
PARENT PROCESS CHILD RESULT
```

2.13.5 The Lease Manager

The Lease Manager provides a mechanism for managing leases to resources, as described by Condor's ClassAd mechanism. The resources and leases are persistent, so that state may be restored after a shutdown or crash.

Resources are advertised to the Lease Manager by publishing a Condor ClassAd with a *condor_collector*. These leases describe the number of resources available (the number of leases), and they can also specify a requirements expression using Condor's ClassAd mechanism. The resource may also specify the maximum duration of the leases it will allow.

Similarly, leases are requested by through the Condor ClassAd mechanism; a Condor daemon client API provides an interface through which to request a lease. This request ClassAd may specify the number of and the duration of the resource leases that are being requested. It may also specify a requirements expression.

With both the resource and request able to specify requirements expressions, the Lease Manager performs 2-way match making, providing a great deal of flexibility.

2.14 Job Monitor

The Condor Job Monitor is a Java application designed to allow users to view user log files.

To view a user log file, select it using the open file command in the File menu. After the file is parsed, it will be visually represented. Each horizontal line represents an individual job. The x-axis is time. Whether a job is running at a particular time is represented by its color at that time – white for running, black for idle. For example, a job which appears predominantly white has made efficient progress, whereas a job which appears predominantly black has received an inordinately small proportion of computational time.

2.14.1 Transition States

A transition state is the state of a job at any time. It is called a "transition" because it is defined by the two events which bookmark it. There are two basic transition states: running and idle. An idle job typically is a job which has just been submitted into the Condor pool and is waiting to be matched with an appropriate machine or a job which has vacated from a machine and has been returned to the pool. A running job, by contrast, is a job which is making active progress.

Advanced users may want a visual distinction between two types of running transitions: "goodput" or "badput". Goodput is the transition state preceding an eventual job completion or checkpoint. Badput is the transition state preceding a non-checkpointed eviction event. Note that "badput" is potentially a misleading nomenclature; a job which is not checkpointed by the Condor program may checkpoint itself or make progress in some other way. To view these two transition as distinct transitions, select the appropriate option from the "View" menu.

2.14.2 Events

There are two basic kinds of events: checkpoint events and error events. Plus advanced users can ask to see more events.

2.14.3 Selecting Jobs

To view any arbitrary selection of jobs in a job file, use the job selector tool. Jobs appear visually by order of appearance within the actual text log file. For example, the log file might contain jobs 775.1, 775.2, 775.3, 775.4, and 775.5, which appear in that order. A user who wishes to see only jobs 775.2 and 775.5 can select only these two jobs in the job selector tool and click the "Ok" or "Apply" button. The job selector supports double clicking; double click on any single job to see it drawn in isolation.

2.14.4 Zooming

To view a small area of the log file, zoom in on the area which you would like to see in greater detail. You can zoom in, out and do a full zoom. A full zoom redraws the log file in its entirety. For example, if you have zoomed in very close and would like to go all the way back out, you could do so with a succession of zoom outs or with one full zoom.

There is a difference between using the menu driven zooming and the mouse driven zooming. The menu driven zooming will recenter itself around the current center, whereas mouse driven zooming will recenter itself (as much as possible) around the mouse click. To help you re-find the clicked area, a box will flash after the zoom. This is called the "zoom finder" and it can be turned off in the zoom menu if you prefer.

2.14.5 Keyboard and Mouse Shortcuts

1. The Keyboard shortcuts:

- Arrows - an approximate ten percent scrollbar movement
- PageUp and PageDown - an approximate one hundred percent scrollbar movement
- Control + Left or Right - approximate one hundred percent scrollbar movement
- End and Home - scrollbar movement to the vertical extreme
- Others - as seen beside menu items

2. The mouse shortcuts:

- Control + Left click - zoom in
- Control + Right click - zoom out
- Shift + left click - re-center

2.15 Special Environment Considerations

2.15.1 AFS

The Condor daemons do not run authenticated to AFS; they do not possess AFS tokens. Therefore, no child process of Condor will be AFS authenticated. The implication of this is that you must set file permissions so that your job can access any necessary files residing on an AFS volume without relying on having your AFS permissions.

If a job you submit to Condor needs to access files residing in AFS, you have the following choices:

1. Copy the needed files from AFS to either a local hard disk where Condor can access them using remote system calls (if this is a standard universe job), or copy them to an NFS volume.
2. If you must keep the files on AFS, then set a host ACL (using the AFS *fs setacl* command) on the subdirectory to serve as the current working directory for the job. If a standard universe job, then the host ACL needs to give read/write permission to any process on the submit machine. If vanilla universe job, then you need to set the ACL such that any host in the pool can access the files without being authenticated. If you do not know how to use an AFS host ACL, ask the person at your site responsible for the AFS configuration.

The Condor Team hopes to improve upon how Condor deals with AFS authentication in a subsequent release.

Please see section 3.12.1 on page 390 in the Administrators Manual for further discussion of this problem.

2.15.2 NFS Automounter

If your current working directory when you run *condor_submit* is accessed via an NFS automounter, Condor may have problems if the automounter later decides to unmount the volume before your job has completed. This is because *condor_submit* likely has stored the dynamic mount point as the job's initial current working directory, and this mount point could become automatically unmounted by the automounter.

There is a simple work around: When submitting your job, use the *initialdir* command in your submit description file to point to the stable access point. For example, suppose the NFS automounter is configured to mount a volume at mount point */a/myserver.company.com/vol1/johndoe* whenever the directory */home/johndoe* is accessed. Adding the following line to the submit description file solves the problem.

```
initialdir = /home/johndoe
```

2.15.3 Condor Daemons That Do Not Run as root

Condor is normally installed such that the Condor daemons have root permission. This allows Condor to run the *condor_shadow* process and your job with your UID and file access rights. When Condor is started as root, your Condor jobs can access whatever files you can.

However, it is possible that whomever installed Condor did not have root access, or decided not to run the daemons as root. That is unfortunate, since Condor is designed to be run as the Unix user root. To see if Condor is running as root on a specific machine, enter the command

```
condor_status -master -l <machine-name>
```

where `machine-name` is the name of the specified machine. This command displays a `condor_master` ClassAd; if the attribute `RealUid` equals zero, then the Condor daemons are indeed running with root access. If the `RealUid` attribute is not zero, then the Condor daemons do not have root access.

NOTE: The Unix program *ps* is *not* an effective method of determining if Condor is running with root access. When using *ps*, it may often appear that the daemons are running as the `condor` user instead of root. However, note that the *ps* command shows the current *effective* owner of the process, not the *real* owner. (See the *getuid(2)* and *geteuid(2)* Unix man pages for details.) In Unix, a process running under the real UID of root may switch its effective UID. (See the *seteuid(2)* man page.) For security reasons, the daemons only set the effective UID to root when absolutely necessary (to perform a privileged operation).

If they are not running with root access, you need to make any/all files and/or directories that your job will touch readable and/or writable by the UID (user id) specified by the `RealUid` attribute. Often this may mean using the Unix command `chmod 777` on the directory where you submit your Condor job.

2.15.4 Job Leases

A job lease specifies how long a given job will attempt to run on a remote resource, even if that resource loses contact with the submitting machine. Similarly, it is the length of time the submitting machine will spend trying to reconnect to the (now disconnected) execution host, before the submitting machine gives up and tries to claim another resource to run the job. The goal aims at run only once semantics, so that the *condor_schedd* daemon does not allow the same job to run on multiple sites simultaneously.

If the submitting machine is alive, it periodically renews the job lease, and all is well. If the submitting machine is dead, or the network goes down, the job lease will no longer be renewed. Eventually the lease expires. While the lease has not expired, the execute host continues to try to run the job, in the hope that the submit machine will come back to life and reconnect. If the job completes and the lease has not expired, yet the submitting machine is still dead, the *condor_starter* daemon will wait for a *condor_shadow* daemon to reconnect, before sending final information on the job, and its output files. Should the lease expire, the *condor_startd* daemon kills off the *condor_starter* daemon and user job.

A default value equal to 20 minutes exists for a job's ClassAd attribute `job_lease_duration`, or this attribute may be set in the submit description file to keep a job running in the case that the submit side no longer renews the lease. There is a trade off in setting the value of `job_lease_duration`. Too small a value, and the job might get killed before the submitting machine has a chance to recover. Forward progress on the job will be lost. Too large a value, and an execute resource will be tied up waiting for the job lease to expire. The value should be chosen based on how long is the user willing to tie up the execute machines, how quickly submit machines come back up, and how much work would be lost if the lease expires, the job is killed, and the job must start over from its beginning.

As a special case, a submit description file setting of

```
job_lease_duration = 0
```

as well as utilizing submission other than *condor_submit* that do not set `JobLeaseDuration` (such as using the web services interface) results in the corresponding job ClassAd attribute to be explicitly undefined. This has the further effect of changing the duration of a claim lease, the amount of time that the execution machine waits before dropping a claim due to missing keep alive messages.

2.16 Potential Problems

2.16.1 Renaming of argv[0]

When Condor starts up your job, it renames `argv[0]` (which usually contains the name of the program) to `condor_exec`. This is convenient when examining a machine's processes with the Unix command *ps*; the process is easily identified as a Condor job.

Unfortunately, some programs read `argv[0]` expecting their own program name and get confused if they find something unexpected like `condor_exec`.

Administrators' Manual

3.1 Introduction

This is the Condor Administrator's Manual for Unix. Its purpose is to aid in the installation and administration of a Condor pool. For help on using Condor, see the Condor User's Manual.

A Condor pool is comprised of a single machine which serves as the *central manager*, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. Periodically, the central manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

Each resource has an owner, the user who works at the machine. This person has absolute power over their own resource and Condor goes out of its way to minimize the impact on this owner caused by Condor. It is up to the resource owner to define a policy for when Condor requests will be serviced and when they will be denied.

Each resource request has an owner as well: the user who submitted the job. These people want Condor to provide as many CPU cycles as possible for their work. Often the interests of the resource owners are in conflict with the interests of the resource requesters.

The job of the Condor administrator is to configure the Condor pool to find the happy medium that keeps both resource owners and users of resources satisfied. The purpose of this manual is to help you understand the mechanisms that Condor provides to enable you to find this happy medium for your particular set of users and resource owners.

3.1.1 The Different Roles a Machine Can Play

Every machine in a Condor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by single machines in your pool. The following list describes what these roles are and what resources are required on the machine that is providing that service:

Central Manager There can be only one central manager for your pool. The machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the central manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, choose for central manager a machine that is likely to be up and running all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in your pool, since they all send updates over the network to the central manager. All queries go to the central manager.

Execute Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

Submit Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server (described below), however the binaries of the jobs you submit are still stored on the submit machine.

Checkpoint Server One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool.

This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

Now that you know the various roles a machine can play in a Condor pool, we will describe the actual daemons within Condor that implement these functions.

3.1.2 The Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

condor_master This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

condor_startd This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the *condor_starter*, described below.

condor_starter This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

condor_schedd This daemon represents resource requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running. When users submit jobs, they go to the schedd, where they are stored in the *job queue*, which the schedd manages. Various tools to view and manipulate the job queue (such as *condor_submit*, *condor_q*, or *condor_rm*) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work.

The schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a *condor_shadow* (described below) to serve that particular request.

condor_shadow This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the *condor_shadow*. Any system call performed on the remote execute machine is sent over the network, back to the *condor_shadow* which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

condor_collector This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The *condor_status* command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

condor_negotiator This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a *negotiation cycle*, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

NOTE: A higher numerical value of the user priority in Condor translate into worse priority for that user. The best priority you can have is 0.5, the lowest numerical value, and your priority gets worse as this number grows.

condor_kbdd This daemon is only needed on Digital Unix. On that platforms, the *condor_startd* cannot determine console (keyboard or mouse) activity directly from the system. The *condor_kbdd* connects to the X Server and periodically checks to see if there has been any activity. If there has, the kbdd sends a command to the startd. That way, the startd knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

condor_ckpt_server This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

condor_quill This daemon builds and manages a database that represents a copy of the Condor job queue. The *condor_q* and *condor_history* tools can then query the database.

condor_dbmsd This daemon assists the *condor_quill* daemon.

condor_gridmanager This daemon handles management and execution of all **grid** universe jobs. The *condor_schedd* invokes the *condor_gridmanager* when there are **grid** universe jobs in the queue, and the *condor_gridmanager* exits when there are no more **grid** universe jobs in the queue.

condor_credd This daemon runs on Windows platforms to manage password storage in a secure manner.

condor_had This daemon implements the high availability of a pool's central manager through monitoring the communication of necessary daemons. If the current, functioning, central manager machine stops working, then this daemon ensures that another machine takes its place, and becomes the central manager of the pool.

condor_replication This daemon assists the *condor_had* daemon by keeping an updated copy of the pool's state. This state provides a better transition from one machine to the next, in the event that the central manager machine stops working.

condor_procd This daemon controls and monitors process families within Condor. Its use is optional in general but it must be used if privilege separation (see Section 3.6.12) or group-ID based tracking (see Section 3.12.10) is enabled.

condor_job_router This daemon transforms **vanilla** universe jobs into **grid** universe jobs, such that the transformed jobs are capable of running elsewhere, as appropriate.

condor_lease_manager This daemon manages leases in a persistent manner. Leases are represented by ClassAds.

stork_server This daemon handles requests for Stork data placement jobs.

See figure 3.1 for a graphical representation of the pool architecture.

3.2 Installation

This section contains the instructions for installing Condor at your Unix site.

The installation will have a default configuration that can be customized. Sections of the manual that follow this one explain customization.

Read this entire section before starting installation.

Please read the copyright and disclaimer information in section ?? on page ?? of the manual, or in the file `LICENSE.TXT`, before proceeding. Installation and use of Condor is acknowledgment that you have read and agree to the terms.

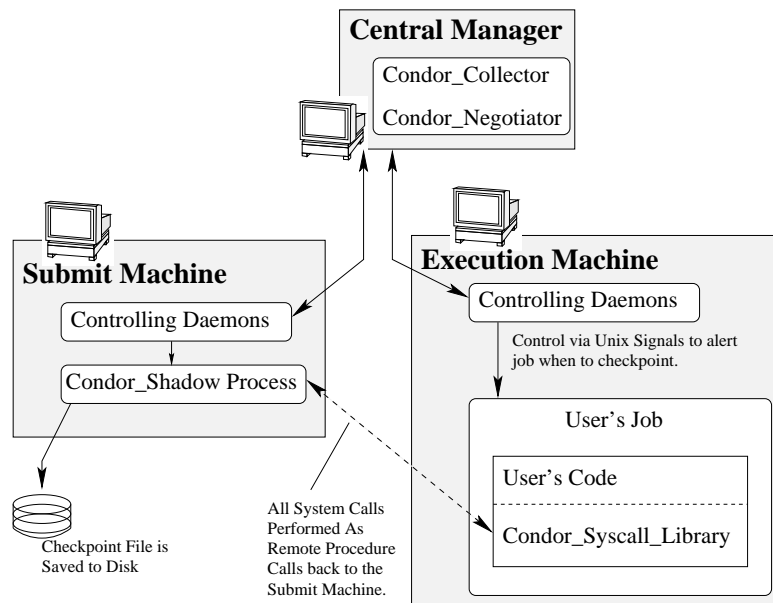


Figure 3.1: Pool Architecture

3.2.1 Obtaining Condor

The first step to installing Condor is to download it from the Condor web site, <http://www.cs.wisc.edu/condor>. The downloads are available from the downloads page, at <http://www.cs.wisc.edu/condor/downloads/>.

The platform-dependent Condor files are currently available from two sites. The main site is at the University of Wisconsin–Madison, Madison, Wisconsin, USA. A second site is the Istituto Nazionale di Fisica Nucleare Sezione di Bologna, Bologna, Italy. Please choose the site nearest to you.

Make note of the location of where you download the binary into.

The Condor binary distribution is packaged in the following 5 files and 2 directories:

DOC directions on where to find Condor documentation

INSTALL these installation directions

LICENSE.TXT the licensing agreement. By installing Condor, you agree to the contents of this file

README general information

condor.install the Perl script used to install and configure Condor

examples directory containing C, Fortran and C++ example programs to run with Condor

bin directory which contains the distribution Condor user programs.

sbin directory which contains the distribution Condor system programs.

etc directory which contains the distribution Condor configuration data.

lib directory which contains the distribution Condor libraries.

libexec directory which contains the distribution Condor programs that are only used internally by Condor.

man directory which contains the distribution Condor manual pages.

sql directory which contains the distribution Condor files used for SQL operations.

src directory which contains the distribution Condor source code for CHIRP and DRMAA.

Before you install, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, send a message to majordomo@cs.wisc.edu with the body:

```
subscribe condor-world
```

3.2.2 Preparation

Before installation, make a few important decisions about the basic layout of your pool. The decisions answer the questions:

1. What machine will be the central manager?
2. What machines should be allowed to submit jobs?
3. Will Condor run as root or not?
4. Who will be administering Condor on the machines in your pool?
5. Will you have a Unix user named condor and will its home directory be shared?
6. Where should the machine-specific directories for Condor go?
7. Where should the parts of the Condor system be installed?
 - Configuration files
 - Release directory
 - user binaries
 - system binaries

- lib directory
- etc directory
- Documentation

8. Am I using AFS?
9. Do I have enough disk space for Condor?

1. What machine will be the central manager? One machine in your pool must be the central manager. Install Condor on this machine first. This is the centralized information repository for the Condor pool, and it is also the machine that does match-making between available machines and submitted jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, install the central manager on a machine that is likely to stay up all the time, or on one that will be rebooted quickly if it does crash.

Also consider network traffic and your network layout when choosing your central manager. All the daemons send updates (by default, every 5 minutes) to this machine. Memory requirements for the central manager differ by the number of machines in the pool. A pool with up to about 100 machines will require approximately 25 Mbytes of memory for the central manager's tasks. A pool with about 1000 machines will require approximately 100 Mbytes of memory for the central manager's tasks.

A faster CPU will improve the time to do matchmaking.

2. Which machines should be allowed to submit jobs? Condor can restrict the machines allowed to submit jobs. Alternatively, it can allow any machine the network allows to connect to a submit machine to submit jobs. If the Condor pool is behind a firewall, and all machines inside the firewall are trusted, the `HOSTALLOW_WRITE` configuration entry can be set to *. Otherwise, it should be set to reflect the set of machines permitted to submit jobs to this pool. Condor tries to be secure by default, so out of the box, the configuration file ships with an invalid definition for this configuration variable. This invalid value allows no machine to connect and submit jobs, so after installation, change this entry. Look for the entry defined with the value `YOU_MUST_CHANGE_THIS_INVALID_CONDOR_CONFIGURATION_VALUE`.

3. Will Condor run as root or not? Start up the Condor daemons as the Unix user root. Without this, Condor can do very little to enforce security and policy decisions. You can install Condor as any user, however there are both serious security and performance consequences. Please see section 3.6.11 on page 316 in the manual for the details and ramifications of running Condor as a Unix user other than root.

4. Who will administer Condor? Either root will be administering Condor directly, or someone else would be acting as the Condor administrator. If root has delegated the responsibility to another person, keep in mind that as long as Condor is started up as root, it should be clearly understood that whoever has the ability to edit the condor configuration files can effectively run arbitrary programs as root.

5. Will you have a Unix user named condor, and will its home directory be shared? To simplify installation of Condor, create a Unix user named condor on all machines in the pool. The Condor daemons will create files (such as the log files) owned by this user, and the home directory can be used to specify the location of files and directories needed by Condor. The home directory of this user can either be shared among all machines in your pool, or could be a separate home directory on the local partition of each machine. Both approaches have advantages and disadvantages. Having the directories centralized can make administration easier, but also concentrates the resource usage such that you potentially need a lot of space for a single shared home directory. See the section below on machine-specific directories for more details.

If you choose not to create a user named condor, then you must specify either via the `CONDOR_IDS` environment variable or the `CONDOR_IDS` config file setting which uid.gid pair should be used for the ownership of various Condor files. See section 3.6.11 on UIDs in Condor on page 315 in the Administrator's Manual for details.

6. Where should the machine-specific directories for Condor go? Condor needs a few directories that are unique on every machine in your pool. These are `spool`, `log`, and `execute`. Generally, all three are subdirectories of a single machine specific directory called the local directory (specified by the `LOCAL_DIR` macro in the configuration file). Each should be owned by the user that Condor is to be run as.

If you have a Unix user named condor with a local home directory on each machine, the `LOCAL_DIR` could just be user condor's home directory (`LOCAL_DIR = $(TILDE)` in the configuration file). If this user's home directory is shared among all machines in your pool, you would want to create a directory for each host (named by host name) for the local directory (for example, `LOCAL_DIR = $(TILDE)/hosts/$(HOSTNAME)`). If you do not have a condor account on your machines, you can put these directories wherever you'd like. However, where to place them will require some thought, as each one has its own resource needs:

execute This is the directory that acts as the current working directory for any Condor jobs that run on a given execute machine. The binary for the remote job is copied into this directory, so there must be enough space for it. (Condor will not send a job to a machine that does not have enough disk space to hold the initial binary). In addition, if the remote job dumps core for some reason, it is first dumped to the execute directory before it is sent back to the submit machine. So, put the execute directory on a partition with enough space to hold a possible core file from the jobs submitted to your pool.

spool The `spool` directory holds the job queue and history files, and the checkpoint files for all jobs submitted from a given machine. As a result, disk space requirements for the `spool` directory can be quite large, particularly if users are submitting jobs with very large executables or image sizes. By using a checkpoint server (see section 3.8 on Installing a Checkpoint Server on page 346 for details), you can ease the disk space requirements, since all checkpoint files are stored on the server instead of the spool directories for each machine. However, the initial checkpoint files (the executables for all the clusters you submit) are still stored in the spool directory, so you will need some space, even with a checkpoint server.

log Each Condor daemon writes its own log file, and each log file is placed in the `log` directory. You can specify what size you want these files to grow to before they are

rotated, so the disk space requirements of the directory are configurable. The larger the log files, the more historical information they will hold if there is a problem, but the more disk space they use up. If you have a network file system installed at your pool, you might want to place the log directories in a shared location (such as `/usr/local/condor/logs/$(HOSTNAME)`), so that you can view the log files from all your machines in a single location. However, if you take this approach, you will have to specify a local partition for the `lock` directory (see below).

lock Condor uses a small number of lock files to synchronize access to certain files that are shared between multiple daemons. Because of problems encountered with file locking and network file systems (particularly NFS), these lock files should be placed on a local partition on each machine. By default, they are placed in the `log` directory. If you place your `log` directory on a network file system partition, specify a local partition for the lock files with the `LOCK` parameter in the configuration file (such as `/var/lock/condor`).

Generally speaking, it is recommended that you do not put these directories (except `lock`) on the same partition as `/var`, since if the partition fills up, you will fill up `/var` as well. This will cause lots of problems for your machines. Ideally, you will have a separate partition for the Condor directories. Then, the only consequence of filling up the directories will be Condor's malfunction, not your whole machine.

7. Where should the parts of the Condor system be installed? • Configuration Files

- Release directory
 - User Binaries
 - System Binaries
 - `lib` Directory
 - `etc` Directory
- Documentation

Configuration Files There are a number of configuration files that allow you different levels of control over how Condor is configured at each machine in your pool. The global configuration file is shared by all machines in the pool. For ease of administration, this file should be located on a shared file system, if possible. In addition, there is a local configuration file for each machine, where you can override settings in the global file. This allows you to have different daemons running, different policies for when to start and stop Condor jobs, and so on. You can also have configuration files specific to each platform in your pool. See section 3.12.2 on page 391 about Configuring Condor for Multiple Platforms for details.

In general, there are a number of places that Condor will look to find its configuration files. The first file it looks for is the global configuration file. These locations are searched in order until a configuration file is found. If none contain a valid configuration file, Condor will print an error message and exit:

1. File specified in the `CONDOR_CONFIG` environment variable
2. `/etc/condor/condor.config`

3. `/usr/local/etc/condor_config`
4. `~condor/condor_config`
5. `$(GLOBUS_LOCATION)/etc/condor_config`

If you specify a file in the `CONDOR_CONFIG` environment variable and there's a problem reading that file, Condor will print an error message and exit right away, instead of continuing to search the other options. However, if no `CONDOR_CONFIG` environment variable is set, Condor will search through the other options.

Next, Condor tries to load the local configuration file(s). The only way to specify the local configuration file(s) is in the global configuration file, with the `LOCAL_CONFIG_FILE` macro. If that macro is not set, no local configuration file is used. This macro can be a list of files or a single file.

Release Directory Every binary distribution contains a contains five subdirectories: `bin`, `etc`, `lib`, `sbin`, and `libexec`. Wherever you choose to install these five directories we call the release directory (specified by the `RELEASE_DIR` macro in the configuration file). Each release directory contains platform-dependent binaries and libraries, so you will need to install a separate one for each kind of machine in your pool. For ease of administration, these directories should be located on a shared file system, if possible.

- **User Binaries:**

All of the files in the `bin` directory are programs the end Condor users should expect to have in their path. You could either put them in a well known location (such as `/usr/local/condor/bin`) which you have Condor users add to their `PATH` environment variable, or copy those files directly into a well known place already in the user's `PATHs` (such as `/usr/local/bin`). With the above examples, you could also leave the binaries in `/usr/local/condor/bin` and put in soft links from `/usr/local/bin` to point to each program.

- **System Binaries:**

All of the files in the `sbin` directory are Condor daemons and agents, or programs that only the Condor administrator would need to run. Therefore, add these programs only to the `PATH` of the Condor administrator.

- **Private Condor Binaries:**

All of the files in the `libexec` directory are Condor programs that should never be run by hand, but are only used internally by Condor.

- **lib Directory:**

The files in the `lib` directory are the Condor libraries that must be linked in with user jobs for all of Condor's checkpointing and migration features to be used. `lib` also contains scripts used by the `condor_compile` program to help re-link jobs with the Condor libraries. These files should be placed in a location that is world-readable, but they do not need to be placed in anyone's `PATH`. The `condor_compile` script checks the configuration file for the location of the `lib` directory.

- **etc Directory:**

`etc` contains an `examples` subdirectory which holds various example configuration files and other files used for installing Condor. `etc` is the recommended location to keep the master copy of your configuration files. You can put in soft links from one of the places mentioned above that Condor checks automatically to find its global configuration file.

Documentation The documentation provided with Condor is currently available in HTML, Postscript and PDF (Adobe Acrobat). It can be locally installed wherever is customary at your site. You can also find the Condor documentation on the web at: <http://www.cs.wisc.edu/condor/manual>.

7. Am I using AFS? If you are using AFS at your site, be sure to read the section 3.12.1 on page 389 in the manual. Condor does not currently have a way to authenticate itself to AFS. A solution is not ready for Version 7.2.3. This implies that you are probably not going to want to have the `LOCAL_DIR` for Condor on AFS. However, you can (and probably should) have the Condor `RELEASE_DIR` on AFS, so that you can share one copy of those files and upgrade them in a centralized location. You will also have to do something special if you submit jobs to Condor from a directory on AFS. Again, read manual section 3.12.1 for all the details.

8. Do I have enough disk space for Condor? Condor takes up a fair amount of space. This is another reason why it is a good idea to have it on a shared file system. The size requirements for the downloads are given on the downloads page. They currently vary from about 20 Mbytes (statically linked HP Unix on a PA RISC) to more than 50 Mbytes (dynamically linked Irix on an SGI).

In addition, you will need a lot of disk space in the local directory of any machines that are submitting jobs to Condor. See question 6 above for details on this.

3.2.3 Newer Unix Installation Procedure

The Perl script *condor_configure* installs Condor. Command-line arguments specify all needed information to this script. The script can be executed multiple times, to modify or further set the configuration. *condor_configure* has been tested using Perl 5.003. Use this or a more recent version of Perl.

After download, all the files are in a compressed, tar format. They need to be untarred, as

```
tar xzf completename.tar.gz
```

After untarring, the directory will have the Perl scripts *condor_configure* and *condor_install*, as well as a “bin”, “etc”, “examples”, “include”, “lib”, “libexec”, “man”, “sbin”, “sql” and “src” subdirectories.

condor_configure and *condor_install* are the same program, but have different default behaviors. *condor_install* is identical to running “*condor_configure* –install=.”. *condor_configure* and *condor_install* work on above directories (“sbin”, etc.). As the names imply, *condor_install* is used to install Condor, whereas *condor_configure* is used to modify the configuration of an existing Condor install.

condor_configure and *condor_install* are completely command-line driven; it is not interactive. Several command-line arguments are always needed with *condor_configure* and *condor_install*. The argument

```
--install=/path/to/release.
```

specifies the path to the Condor release directories (see above). The default for *condor_install* is “--install=.”. The argument

```
--install-dir=directory
```

or

```
--prefix=directory
```

specifies the path to the install directory.

The argument

```
--local-dir=directory
```

specifies the path to the local directory.

The **–type** option to *condor_configure* specifies one or more of the roles that a machine may take on within the Condor pool: central manager, submit or execute. These options are given in a comma separated list. So, if a machine is both a submit and execute machine, the proper command-line option is

```
--type=manager,execute
```

Install Condor on the central manager machine first. If Condor will run as root in this pool (Item 3 above), run *condor_install* as root, and it will install and set the file permissions correctly. On the central manager machine, run *condor_install* as follows.

```
% condor_install --prefix=~condor \
--local-dir=/scratch/condor --type=manager
```

To update the above Condor installation, for example, to also be submit machine:

```
% condor_configure --prefix=~condor \
--local-dir=/scratch/condor --type=manager,submit
```

As in the above example, the central manager can also be a submit point or and execute machine, but this is only recommended for very small pools. If this is the case, the **–type** option changes to *manager,execute* or *manager,submit* or *manager,submit,execute*.

After the central manager is installed, the execute and submit machines should then be configured. Decisions about whether to run Condor as root should be consistent throughout the pool. For each machine in the pool, run

```
% condor_install --prefix=~condor \  
--local-dir=/scratch/condor --type=execute,submit
```

See the *condor_configure* manual page in section 9 on page 704 for details.

3.2.4 Condor is installed Under Unix ... now what?

Now that Condor has been installed on your machine(s), there are a few things you should check before you start up Condor.

1. Read through the `<release_dir>/etc/condor_config` file. There are a lot of possible settings and you should at least take a look at the first two main sections to make sure everything looks okay. In particular, you might want to set up security for Condor. See the section 3.6.1 on page 282 to learn how to do this.
2. Condor can monitor the activity of your mouse and keyboard, provided that you tell it where to look. You do this with the `CONSOLE_DEVICES` entry in the `condor_startd` section of the configuration file. On most platforms, reasonable defaults are provided. For example, the default device for the mouse on Linux is 'mouse', since most Linux installations have a soft link from `/dev/mouse` that points to the right device (such as `tty00` if you have a serial mouse, `psaux` if you have a PS/2 bus mouse, etc). If you do not have a `/dev/mouse` link, you should either create one (you will be glad you did), or change the `CONSOLE_DEVICES` entry in Condor's configuration file. This entry is a comma separated list, so you can have any devices in `/dev` count as 'console devices' and activity will be reported in the `condor_startd`'s ClassAd as `ConsoleIdleTime`.
3. (Linux only) Condor needs to be able to find the `utmp` file. According to the Linux File System Standard, this file should be `/var/run/utmp`. If Condor cannot find it there, it looks in `/var/adm/utmp`. If it still cannot find it, it gives up. So, if your Linux distribution places this file somewhere else, be sure to put a soft link from `/var/run/utmp` to point to the real location.

To start up the Condor daemons, execute `<release_dir>/sbin/condor_master`. This is the Condor master, whose only job in life is to make sure the other Condor daemons are running. The master keeps track of the daemons, restarts them if they crash, and periodically checks to see if you have installed new binaries (and if so, restarts the affected daemons).

If you are setting up your own pool, you should start Condor on your central manager machine first. If you have done a submit-only installation and are adding machines to an existing pool, the start order does not matter.

To ensure that Condor is running, you can run either:

```
ps -ef | egrep condor_
```

or

```
ps -aux | egrep condor_
```

depending on your flavor of Unix. On a central manager machine that can submit jobs as well as execute them, there will be processes for:

- `condor_master`
- `condor_collector`
- `condor_negotiator`
- `condor_startd`
- `condor_schedd`

On a central manager machine that does not submit jobs nor execute them, there will be processes for:

- `condor_master`
- `condor_collector`
- `condor_negotiator`

For a machine that only submits jobs, there will be processes for:

- `condor_master`
- `condor_schedd`

For a machine that only executes jobs, there will be processes for:

- `condor_master`
- `condor_startd`

Once you are sure the Condor daemons are running, check to make sure that they are communicating with each other. You can run `condor_status` to get a one line summary of the status of each machine in your pool.

Once you are sure Condor is working properly, you should add `condor_master` into your startup/bootup scripts (i.e. `/etc/rc`) so that your machine runs `condor_master` upon bootup. `condor_master` will then fire up the necessary Condor daemons whenever your machine is rebooted.

If your system uses System-V style init scripts, you can look in `<release_dir>/etc/examples/condor.boot` for a script that can be used to start and stop Condor automatically by init. Normally, you would install this script as `/etc/init.d/condor` and put in soft link from various directories (for example, `/etc/rc2.d`) that point back to `/etc/init.d/condor`. The exact location of these scripts and links will vary on different platforms.

If your system uses BSD style boot scripts, you probably have an `/etc/rc.local` file. Add a line to start up `<release_dir>/sbin/condor_master`.

Now that the Condor daemons are running, there are a few things you can and should do:

1. (Optional) Do a full install for the *condor_compile* script. *condor_compile* assists in linking jobs with the Condor libraries to take advantage of all of Condor's features. As it is currently installed, it will work by placing it in front of any of the following commands that you would normally use to link your code: `gcc`, `g++`, `g77`, `cc`, `acc`, `c89`, `CC`, `f77`, `fort77` and `ld`. If you complete the full install, you will be able to use *condor_compile* with any command whatsoever, in particular, `make`. See section 3.12.3 on page 394 in the manual for directions.
2. Try building and submitting some test jobs. See `examples/README` for details.
3. If your site uses the AFS network file system, see section 3.12.1 on page 389 in the manual.
4. We strongly recommend that you start up Condor (run the *condor_master* daemon) as user root. If you must start Condor as some user other than root, see section 3.6.11 on page 316.

3.2.5 Installation on Windows

This section contains the instructions for installing the Microsoft Windows version of Condor. The install program will set up a slightly customized configuration file that may be further customized after the installation has completed.

Please read the copyright and disclaimer information in section ?? on page ?? of the manual, or in the file `LICENSE.TXT`, before proceeding. Installation and use of Condor is acknowledgment that you have read and agreed to these terms.

Be sure that the Condor tools run are of the same version as the daemons installed. If they were not (such as 6.9.12 daemons, when running 6.8.4 *condor_submit*), then things will not work. There may be errors generated by the *condor_schedd* daemon in the log. It is likely that a job would be correctly placed in the queue, but the job will never run.

The Condor executable for distribution is packaged in a single file such as:

```
condor-6.7.8-winnt40-x86.msi
```

This file is approximately 80 Mbytes in size, and may be removed once Condor is fully installed.

Before installing Condor, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, follow the directions given at <http://www.cs.wisc.edu/condor/mail-lists/>.

Installation Requirements

- Condor for Windows requires Windows 2000 (or better) or Windows XP.
- 300 megabytes of free disk space is recommended. Significantly more disk space could be desired to be able to run jobs with large data files.
- Condor for Windows will operate on either an NTFS or FAT file system. However, for security purposes, NTFS is preferred.

Preparing to Install Condor under Windows

Before installing the Windows version of Condor, there are two major decisions to make about the basic layout of the pool.

1. What machine will be the central manager?
2. Do I have enough disk space for Condor?

If you feel that you already know the answers to these questions, skip to the Windows Installation Procedure section below, section 3.2.5 on page 129. If you are unsure, read on.

- What machine will be the central manager?

One machine in your pool must be the central manager. This is the centralized information repository for the Condor pool and is also the machine that matches available machines with waiting jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, we recommend you install it on a machine that is likely to stay up all the time, or at the very least, one that will be rebooted quickly if it does crash. Also, because all the services will send updates (by default every 5 minutes) to this machine, it is advisable to consider network traffic and your network layout when choosing the central manager.

For Personal Condor, your machine will act as your central manager.

Install Condor on the central manager before installing on the other machines within the pool.

- Do I have enough disk space for Condor?

The Condor release directory takes up a fair amount of space. The size requirement for the release directory is approximately 200 Mbytes.

Condor itself, however, needs space to store all of your jobs, and their input files. If you will be submitting large amounts of jobs, you should consider installing Condor on a volume with a large amount of free space.

Installation Procedure Using the Included Set Up Program

Installation of Condor must be done by a user with administrator privileges. After installation, the Condor services will be run under the local system account. When Condor is running a user job, however, it will run that user job with normal user permissions.

Download Condor, and start the installation process by running the file (or by double clicking on the file). The Condor installation is completed by answering questions and choosing options within the following steps.

If Condor is already installed. For upgrade purposes, you may be running the installation of Condor after it has been previously installed. In this case, a dialog box will appear before the installation of Condor proceeds. The question asks if you wish to preserve your current Condor configuration files. Answer yes or no, as appropriate.

If you answer yes, your configuration files will not be changed, and you will proceed to the point where the new binaries will be installed.

If you answer no, then there will be a second question that asks if you want to use answers given during the previous installation as default answers.

STEP 1: License Agreement. The first step in installing Condor is a welcome screen and license agreement. You are reminded that it is best to run the installation when no other Windows programs are running. If you need to close other Windows programs, it is safe to cancel the installation and close them. You are asked to agree to the license. Answer yes or no. If you should disagree with the License, the installation will not continue.

After agreeing to the license terms, the next Window is where fill in your name and company information, or use the defaults as given.

STEP 2: Condor Pool Configuration. The Condor installation will require different information depending on whether the installer will be creating a new pool, or joining an existing one.

If you are creating a new pool, the installation program requires that this machine is the central manager. For the creation of a new Condor pool, you will be asked some basic information about your new pool:

Name of the pool

hostname of this machine.

Size of pool Condor needs to know if this a Personal Condor installation, or if there will be more than one machine in the pool. A Personal Condor pool implies that there is only one machine in the pool. For Personal Condor, several of the following steps are omitted as noted.

If you are joining an existing pool, all the installation program requires is the host name of the central manager for your pool.

STEP 3: This Machine's Roles. This step is omitted for the installation of Personal Condor.

Each machine within a Condor pool may either submit jobs or execute submitted jobs, or both submit and execute jobs. This step allows the installation on this machine to choose if the machine will only submit jobs, only execute submitted jobs, or both. The common case is both, so the default is both.

STEP 4: Where will Condor be installed? The next step is where the destination of the Condor files will be decided. It is recommended that Condor be installed in the location shown as the default in the dialog box: C:\Condor.

Installation on the local disk is chosen for several reasons.

The Condor services run as local system, and within Microsoft Windows, local system has no network privileges. Therefore, for Condor to operate, Condor should be installed on a local hard drive as opposed to a network drive (file server).

The second reason for installation on the local disk is that the Windows usage of drive letters has implications for where Condor is placed. The drive letter used must be not change, even when different users are logged in. Local drive letters do not change under normal operation of Windows.

While it is strongly discouraged, it may be possible to place Condor on a hard drive that is not local, if a dependency is added to the service control manager such that Condor starts after the required file services are available.

STEP 5: Where is the Java Virtual Machine? While not required, it is possible for Condor to run jobs in the Java universe. In order for Condor to have support for java, you must supply a path to `java.exe` on your system. The installer will tell you if the path is invalid before proceeding to the next step. To disable the Java universe, simply leave this field blank.

STEP 6: Where should Condor send e-mail if things go wrong? Various parts of Condor will send e-mail to a Condor administrator if something goes wrong and requires human attention. You specify the e-mail address and the SMTP relay host of this administrator. Please pay close attention to this email since it will indicate problems in your Condor pool.

STEP 7: The domain. This step is omitted for the installation of Personal Condor.

Enter the machine's accounting (or UID) domain. On this version of Condor for Windows, this setting only used for User priorities (see section 3.4 on page 244) and to form a default email address for the user.

STEP 8: Access permissions. This step is omitted for the installation of Personal Condor.

Machines within the Condor pool will need various types of access permission. The three categories of permission are read, write, and administrator. Enter the machines to be given access permissions.

Read Read access allows a machine to obtain information about Condor such as the status of machines in the pool and the job queues. All machines in the pool should be given read

access. In addition, giving read access to *.cs.wisc.edu will allow the Condor team to obtain information about your Condor pool in the event that debugging is needed.

Write All machines in the pool should be given write access. It allows the machines you specify to send information to your local Condor daemons, for example, to start a Condor Job. Note that for a machine to join the Condor pool, it must have both read and write access to all of the machines in the pool.

Administrator A machine with administrator access will be allowed more extended permission to things such as change other user's priorities, modify the job queue, turn Condor services on and off, and restart Condor. The central manager should be given administrator access and is the default listed. This setting is granted to the entire machine, so care should be taken not to make this too open.

For more details on these access permissions, and others that can be manually changed in your `condor_config` file, please see the section titled Setting Up IP/Host-Based Security in Condor in section 3.6.9 on page 307.

STEP 9: Job Start Policy. Condor will execute submitted jobs on machines based on a preference given at installation. Three options are given, and the first is most commonly used by Condor pools. This specification may be changed or refined in the machine ClassAd requirements attribute.

The three choices:

After 15 minutes of no console activity and low CPU activity.

Always run Condor jobs.

After 15 minutes of no console activity.

Console activity is the use of the mouse or keyboard. For instance, if you are reading this document on line, and are using either the mouse or the keyboard to change your position, you are generating Console activity.

Low CPU activity is defined as a load of less than 30%(and is configurable in your `condor_config` file). If you have a multiple processor machine, this is the average percentage of CPU activity for both processors.

For testing purposes, it is often helpful to use the Always run Condor jobs option. For production mode, however, most people chose the After 15 minutes of no console activity and low CPU activity.

STEP 10: Job Vacate Policy. This step is omitted if Condor jobs are always run as the option chosen in STEP 9.

If Condor is executing a job and the user returns, Condor will immediately suspend the job, and after five minutes Condor will decide what to do with the partially completed job. There are currently two options for the job.

The job is killed 5 minutes after your return. The job is suspended immediately once there is console activity. If the console activity continues, then the job is vacated (killed) after 5 minutes. Since this version does not include check-pointing, the job will be restarted from the beginning at a later time. The job will be placed back into the queue.

Suspend job, leaving it in memory. The job is suspended immediately. At a later time, when the console activity has stopped for ten minutes, the execution of Condor job will be resumed (the job will be unsuspended). The drawback to this option is that since the job will remain in memory, it will occupy swap space. In many instances, however, the amount of swap space that the job will occupy is small.

So which one do you choose? Killing a job is less intrusive on the workstation owner than leaving it in memory for a later time. A suspended job left in memory will require swap space, which could possibly be a scarce resource. Leaving a job in memory, however, has the benefit that accumulated run time is not lost for a partially completed job.

STEP 11: Review entered information. Check that the entered information is correctly entered. You have the option to return to previous dialog boxes to fix entries.

Unattended Installation Procedure Using the Included Set Up Program

This section details how to run the Condor for Windows installer in an unattended batch mode. This mode is one that occurs completely from the command prompt, without the GUI interface.

The Condor for Windows installer uses the Microsoft Installer (MSI) technology, and it can be configured for unattended installs analogous to any other ordinary MSI installer.

The following is a sample batch file that is used to set all the properties necessary for an unattended install.

```
@echo on
set ARGS=
set ARGS=%ARGS% NEWPOOL="N"
set ARGS=%ARGS% POOLNAME=" "
set ARGS=%ARGS% RUNJOBS="C"
set ARGS=%ARGS% VACATEJOBS="Y"
set ARGS=%ARGS% SUBMITJOBS="Y"
set ARGS=%ARGS% CONDOREMAIL="you@yours.com"
set ARGS=%ARGS% SMTPSERVER="smtp.localhost"
set ARGS=%ARGS% HOSTALLOWREAD="*"
set ARGS=%ARGS% HOSTALLOWWRITE="*"
set ARGS=%ARGS% HOSTALLOWADMINISTRATOR="% (FULL_HOSTNAME) "
set ARGS=%ARGS% INSTALLDIR="C:\Condor"
set ARGS=%ARGS% INSTALLDIR_NTS="C:\Condor"
set ARGS=%ARGS% POOLHOSTNAME="% (FULL_HOSTNAME) "
set ARGS=%ARGS% ACCOUNTINGDOMAIN="none"
set ARGS=%ARGS% JVMLOCATION="C:\Windows\system32\java.exe"
set ARGS=%ARGS% USEVMUNIVERSE="N"
set ARGS=%ARGS% VMVERSION="server1.4"
set ARGS=%ARGS% VMMEMORY="128"
set ARGS=%ARGS% VMMAXNUMBER="% (NUM_CPUS) "
```

```
set ARGS=%ARGS% VMNETWORKING="N"

msiexec /qb /l* condor-install-log.txt /i condor-7.1.0-winnt50-x86.msi %ARGS%
```

Each property corresponds to answers that would have been supplied while running an interactive installer. The following is a brief explanation of each property as it applies to unattended installations:

NEWPOOL = `< Y | N >` determines whether the installer will create a new pool with the target machine as the central manager.

POOLNAME sets the name of the pool, if a new pool is to be created. Possible values are either the name or the empty string `" "`.

RUNJOBS = `< N | A | I | C >` determines when Condor will run jobs. This can be set to:

- Never run jobs (N)
- Always run jobs (A)
- Only run jobs when the keyboard and mouse are Idle (I)
- Only run jobs when the keyboard and mouse are idle and the CPU usage is low (C)

VACATEJOBS = `< Y | N >` determines what Condor should do when it has to stop the execution of a user job. When set to Y, Condor will vacate the job and start it somewhere else if possible. When set to N, Condor will merely suspend the job in memory and wait for the machine to become available again.

SUBMITJOBS = `< Y | N >` will cause the installer to configure the machine as a submit node when set to Y.

CONDOREMAIL sets the e-mail address of the Condor administrator. Possible values are an e-mail address or the empty string `" "`.

HOSTALLOWREAD is a list of host names that are allowed to issue READ commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_READ` setting in the configuration file, as described in section 3.6.9 on page 307.

HOSTALLOWWRITE is a list of host names that are allowed to issue WRITE commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_WRITE` setting in the configuration file, as described in section 3.6.9 on page 307.

HOSTALLOWADMINISTRATOR is a list of host names that are allowed to issue ADMINISTRATOR commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_ADMINISTRATOR` setting in the configuration file, as described in section 3.6.9 on page 307.

INSTALLDIR defines the path to the directory where Condor will be installed.

INSTALLDIR_NTS should be set to whatever **INSTALLDIR** is set to, with the additional restriction that it cannot end in a backslash. The installer will be fixed in an upcoming version of Condor to not require this property.

POOLHOSTNAME defines the host name of the pool's central manager.

ACCOUNTINGDOMAIN defines the accounting (or UID) domain the target machine will be in.

JVMLOCATION defines the path to Java virtual machine on the target machine.

SMTPSERVER defines the host name of the SMTP server that the target machine is to use to send e-mail.

VMVERSION defines the version of VMware installed on the target machine.

VMMEMORY an integer value that defines the maximum memory each VM run on the target machine.

VMMAXNUMBER an integer value that defines the number of VMs that can be run in parallel on the target machine.

VMNETWORKING = < N | A | B | C > determines if VM Universe can use networking. This can be set to:

- None (N)
- NAT (A)
- Bridged (B)
- NAT and Bridged (C)

USEVMUNIVERSE = < Y | N > will cause the installer to enable VM Universe jobs on the target machine.

PERLLOCATION defines the path to *Perl* on the target machine. This is required in order to use the **vm** universe.

After defining each of these properties for the MSI installer, the installer can be started with the *msiexec* command. The following command starts the installer in unattended mode, and it dumps a journal of the installer's progress to a log file:

```
msiexec /qb /l xv* condor-install-log.txt /i condor-7.2.2-winnt50-x86.msi [property=value] ...
```

More information on the features of *msiexec* can be found at Microsoft's website at <http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/msiexec.msp>.

Manual Installation Condor on Windows

If you are to install Condor on many different machines, you may wish to use some other mechanism to install Condor on additional machines rather than running the Setup program described above on each machine.

WARNING: This is for advanced users only! All others should use the Setup program described above.

Here is a brief overview of how to install Condor manually without using the provided GUI-based setup program:

The Service The service that Condor will install is called "Condor". The Startup Type is Automatic. The service should log on as System Account, but **do not enable** "Allow Service to Interact with Desktop". The program that is run is *condor_master.exe*.

The Condor service can be installed and removed using the *sc.exe* tool, which is included in Windows XP and Windows 2003 Server. The tool is also available as part of the Windows 2000 Resource Kit.

Installation can be done as follows:

```
sc create Condor binpath= c:\condor\bin\condor_master.exe
```

To remove the service, use:

```
sc delete Condor
```

The Registry Condor uses a few registry entries in its operation. The key that Condor uses is HKEY_LOCAL_MACHINE/Software/Condor. The values that Condor puts in this registry key serve two purposes.

1. The values of CONDOR_CONFIG and RELEASE_DIR are used for Condor to start its service.
CONDOR_CONFIG should point to the *condor_config* file. In this version of Condor, it **must** reside on the local disk.
RELEASE_DIR should point to the directory where Condor is installed. This is typically C:\Condor, and again, this **must** reside on the local disk.
2. The other purpose is storing the entries from the last installation so that they can be used for the next one.

The File System The files that are needed for Condor to operate are identical to the Unix version of Condor, except that executable files end in *.exe*. For example the on Unix one of the files is *condor_master* and on Condor the corresponding file is *condor_master.exe*.

These files currently must reside on the local disk for a variety of reasons. Advanced Windows users might be able to put the files on remote resources. The main concern is twofold. First, the files must be there when the service is started. Second, the files must always be in the same spot (including drive letter), no matter who is logged into the machine.

Note also that when installing manually, you will need to create the directories that Condor will expect to be present given your configuration. This normally is simply a matter of creating the `log`, `spool`, and `execute` directories.

Condor Is Installed Under Windows ... Now What?

After the installation of Condor is completed, the Condor service must be started. If you used the GUI-based setup program to install Condor, the Condor service should already be started. If you installed manually, Condor must be started by hand, or you can simply reboot. **NOTE:** The Condor service will start automatically whenever you reboot your machine.

To start Condor by hand:

1. From the Start menu, choose Settings.
2. From the Settings menu, choose Control Panel.
3. From the Control Panel, choose Services.
4. From Services, choose Condor, and Start.

Or, alternatively you can enter the following command from a command prompt:

```
net start condor
```

Run the Task Manager (Control-Shift-Escape) to check that Condor services are running. The following tasks should be running:

- *condor_master.exe*
- *condor_negotiator.exe*, if this machine is a central manager.
- *condor_collector.exe*, if this machine is a central manager.
- *condor_startd.exe*, if you indicated that this Condor node should start jobs
- *condor_schedd.exe*, if you indicated that this Condor node should submit jobs to the Condor pool.

Also, you should now be able to open up a new cmd (DOS prompt) window, and the Condor bin directory should be in your path, so you can issue the normal Condor commands, such as *condor_q* and *condor_status*.

Condor is Running Under Windows ... Now What?

Once Condor services are running, try submitting test jobs. Example 2 within section 2.5.1 on page 20 presents a vanilla universe job.

3.2.6 RPMs

RPMs are available in Version 7.2.3. This packaging method provides for installation and configuration in one easy step. It is currently available for Linux systems only.

The format of the installation command is

```
rpm -i <filename> --prefix=<installation dir>
```

The user provides the path name to the directory used for the installation. The *rpm* program calls *condor_configure* to do portions of the installation. If the *condor* user is present on the system, the installation script will assume that that is the effective user that Condor should run as (see section 3.6.11 on page 315). If the *condor* user is not present, the *daemon* user will be used. This user will be present on all Linux systems. Note that the user can later be changed by running the *condor_configure* program using the **owner** option, of the format:

```
condor_configure --owner=<user>
```

After a successful installation, the `CONDOR_CONFIG` configuration variable must be set to point to

```
<installation dir>/etc/condor_config
```

before starting Condor daemons or invoking Condor tools.

RPM upgrade (**-u** option) does not currently work for Condor Version 7.2.3.

3.2.7 Upgrading - Installing a Newer Version of Condor

An upgrade changes the running version of Condor from the current installation to a newer version. The safe method to install and start running a newer version of Condor in essence is: shutdown the current installation of Condor, install the newer version, and then restart Condor using the newer version.

To allow for falling back to the current version, place the new version in a separate directory. Copy the existing configuration files, and modify the copy to point to and use the new version. Set the `CONDOR_CONFIG` environment variable to point to the new copy of the configuration, so the new version of Condor will use the new configuration when restarted.

When upgrading from an earlier version of Condor to a version of 6.8, note that the configuration settings must be modified for security reasons. Specifically, the `HOSTALLOW_WRITE` configuration variable must be explicitly changed, or no jobs may be submitted, and error messages will be issued by Condor tools.

3.2.8 Installing the CondorView Client Contrib Module

The CondorView Client contrib module is used to automatically generate World Wide Web pages to display usage statistics of a Condor pool. Included in the module is a shell script which invokes the *condor_stats* command to retrieve pool usage statistics from the CondorView server, and generate HTML pages from the results. Also included is a Java applet, which graphically visualizes Condor usage information. Users can interact with the applet to customize the visualization and to zoom in to a specific time frame. Figure 3.2 on page 138 is a screen shot of a web page created by CondorView. To get a further feel for what pages generated by CondorView look like, view the statistics for the University of Wisconsin-Madison pool by visiting the URL <http://www.cs.wisc.edu/condor> and clicking on Condor View.

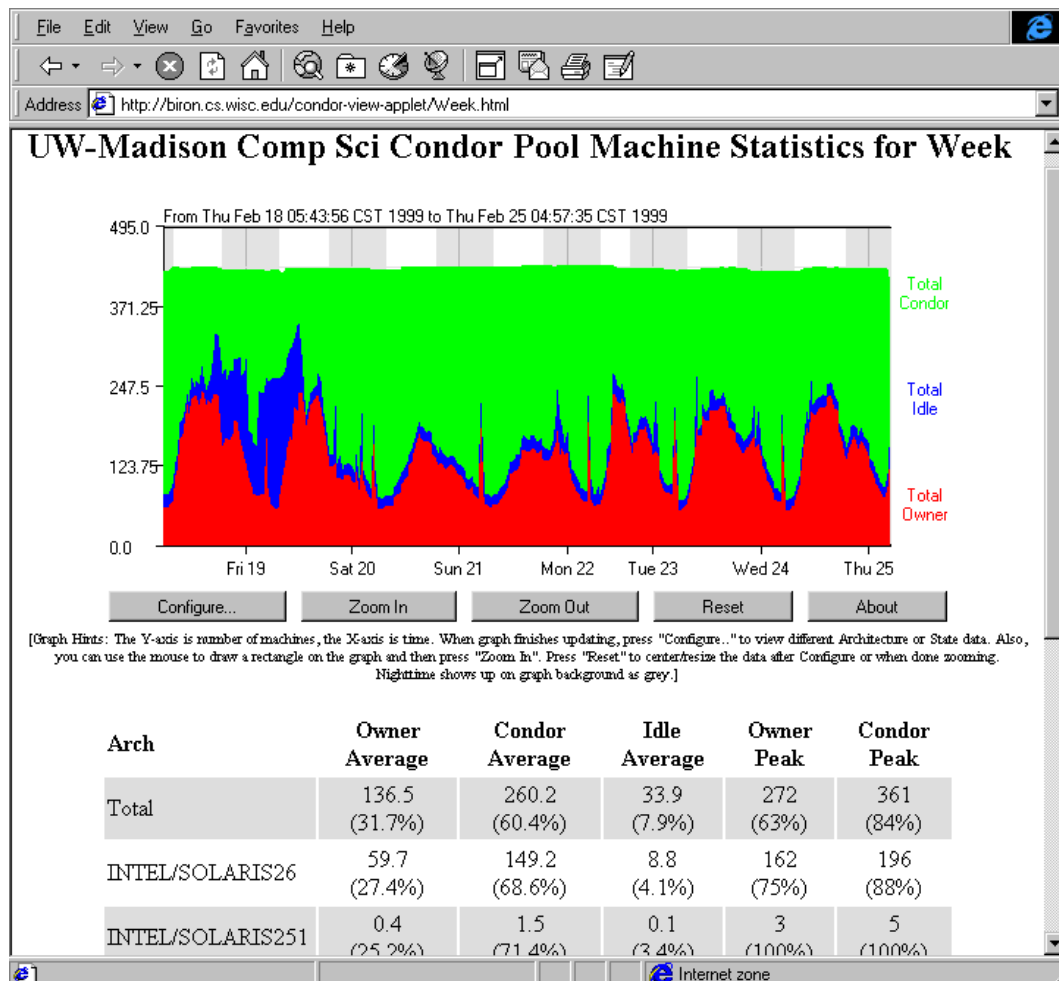


Figure 3.2: Screen shot of CondorView Client

After unpacking and installing the CondorView Client, a script named *make_stats* can be invoked to create HTML pages displaying Condor usage for the past hour, day, week, or month. By using the Unix *cron* facility to periodically execute *make_stats*, Condor pool usage statistics can be kept up to date automatically. This simple model allows the CondorView Client to be easily installed; no Web server CGI interface is needed.

Step-by-Step Installation of the CondorView Client

1. Make certain that the CondorView Server is configured. Section 3.12.5 describes configuration of the server. The server logs information on disk in order to provide a persistent, historical database of pool statistics. The CondorView Client makes queries over the network to this database. The *condor_collector* includes this database support. To activate the persistent database logging, add the following entries to the configuration file for the *condor_collector* chosen to act as the ViewServer.

```
POOL_HISTORY_DIR = /full/path/to/directory/to/store/historical/data
KEEP_POOL_HISTORY = True
```

2. Create a directory where CondorView is to place the HTML files. This directory should be one published by a web server, so that HTML files which exist in this directory can be accessed using a web browser. This directory is referred to as the **VIEWDIR** directory.
3. Download the *view_client* contrib module.
4. Unpack or untar this contrib module into the directory **VIEWDIR**. This creates several files and subdirectories. Further unpack the jar file within the **VIEWDIR** directory with:

```
jar -xf condorview.jar
```

5. Edit the *make_stats* script. At the beginning of the file are six parameters to customize. The parameters are

ORGNAME A brief name that identifies an organization. An example is “Univ of Wisconsin”. Do not use any slashes in the name or other special regular-expression characters. Avoid the characters `\` and `$`.

CONDORADMIN The e-mail address of the Condor administrator at your site. This e-mail address will appear at the bottom of the web pages.

VIEWDIR The full path name (*not* a relative path) to the **VIEWDIR** directory set by installation step 2. It is the directory that contains the *make_stats* script.

STATSDIR The full path name of the directory which contains the *condor_stats* binary. The *condor_stats* program is included in the `<release_dir>/bin` directory. The value for **STATSDIR** is added to the **PATH** parameter by default.

PATH A list of subdirectories, separated by colons, where the *make_stats* script can find the *awk*, *bc*, *sed*, *date*, and *condor_stats* programs. If *perl* is installed, the path should also include the directory where *perl* is installed. The following default works on most systems:

```
PATH=/bin:/usr/bin:$STATSDIR:/usr/local/bin
```

6. To create all of the initial HTML files, run

```
./make_stats setup
```

Open the file `index.html` to verify that things look good.

7. Add the *make_stats* program to *cron*. Running *make_stats* in step 6 created a `cronentries` file. This `cronentries` file is ready to be processed by the Unix *crontab* command. The *crontab* manual page contains details about the *crontab* command and the *cron* daemon. Look at the `cronentries` file; by default, it will run *make_stats hour* every 15 minutes, *make_stats day* once an hour, *make_stats week* twice per day, and *make_stats month* once per day. These are reasonable defaults. Add these commands to cron on any system that can access the `VIEWDIR` and `STATSDIR` directories, even on a system that does not have Condor installed. The commands do not need to run as root user; in fact, they should probably not run as root. These commands can run as any user that has read/write access to the `VIEWDIR` directory. To add these commands to cron, run

```
crontab cronentries
```

8. Point the web browser at the `VIEWDIR` directory to complete the installation.

3.2.9 Dynamic Deployment

Dynamic deployment is a mechanism that allows rapid, automated installation and start up of Condor resources on a given machine. In this way any machine can be added to a Condor pool. The dynamic deployment tool set also provides tools to remove a machine from the pool, without leaving residual effects on the machine such as leftover installations, log files, and working directories.

Installation and start up is provided by *condor_cold_start*. The *condor_cold_start* program determines the operating system and architecture of the target machine, and transfers the correct installation package from an ftp, http, or grid ftp site. After transfer, it installs Condor and creates a local working directory for Condor to run in. As a last step, *condor_cold_start* begins running Condor in a manner which allows for later easy and reliable shut down.

The program that reliably shuts down and uninstalls a previously dynamically installed Condor instance is *condor_cold_stop*. *condor_cold_stop* begins by safely and reliably shutting off the running Condor installation. It ensures that Condor has completely shut down before continuing, and optionally ensures that there are no queued jobs at the site. Next, *condor_cold_stop* removes and optionally archives the Condor working directories, including the `log` directory. These archives can be stored to a mounted file system or to a grid ftp site. As a last step, *condor_cold_stop* uninstalls the Condor executables and libraries. The end result is that the machine resources are left unchanged after a dynamic deployment of Condor leaves.

Configuration and Usage

Dynamic deployment is designed for the expert Condor user and administrator. Tool design choices were made for functionality, not ease-of-use.

Like every installation of Condor, a dynamically deployed installation relies on a configuration. To add a target machine to a previously created Condor pool, the global configuration file for that pool is a good starting point. Modifications to that configuration can be made in a separate, local configuration file used in the dynamic deployment. The global configuration file must be placed on an ftp, http, grid ftp, or file server accessible by *condor_cold_start*. The local configuration file is to be on a file system accessible by the target machine. There are some specific configuration variables that may be set for dynamic deployment. A list of executables and directories which must be present for Condor to start on the target machine may be set with the configuration variables `DEPLOYMENT_REQUIRED_EXECS` and `DEPLOYMENT_REQUIRED_DIRS`. If defined and the comma-separated list of executables or directories are not present, then *condor_cold_start* exits with error. Note this does not affect what is installed, only whether start up is successful.

A list of executables and directories which are recommended to be present for Condor to start on the target machine may be set with the configuration variables `DEPLOYMENT_RECOMMENDED_EXECS` and `DEPLOYMENT_RECOMMENDED_DIRS`. If defined and the comma-separated lists of executables or directories are not present, then *condor_cold_start* prints a warning message and continues. Here is a portion of the configuration relevant to a dynamic deployment of a Condor submit node:

```
DEPLOYMENT_REQUIRED_EXECS = MASTER, SCHEDD, PREEN, STARTER, \
                           STARTER_STANDARD, SHADOW, \
                           SHADOW_STANDARD, GRIDMANAGER, GAHP, CONDOR_GAHP
DEPLOYMENT_REQUIRED_DIRS  = SPOOL, LOG, EXECUTE
DEPLOYMENT_RECOMMENDED_EXECS = CREDD
DEPLOYMENT_RECOMMENDED_DIRS = LIB, LIBEXEC
```

Additionally, the user must specify which Condor services will be started. This is done through the `DAEMON_LIST` configuration variable. Another excerpt from a dynamic submit node deployment configuration:

```
DAEMON_LIST = MASTER, SCHEDD
```

Finally, the location of the dynamically installed Condor executables is tricky to set, since the location is unknown before installation. Therefore, the variable `DEPLOYMENT_RELEASE_DIR` is defined in the environment. It corresponds to the location of the dynamic Condor installation. If, as is often the case, the configuration file specifies the location of Condor executables in relation to the `RELEASE_DIR` variable, the configuration can be made dynamically deployable by setting `RELEASE_DIR` to `DEPLOYMENT_RELEASE_DIR` as

```
RELEASE_DIR = $(DEPLOYMENT_RELEASE_DIR)
```

In addition to setting up the configuration, the user must also determine where the installation package will reside. The installation package can be in either tar or gzipped tar form, and may reside on a ftp, http, grid ftp, or file server. Create this installation package by tar'ing up the binaries and libraries needed, and place them on the appropriate server. The binaries can be tar'ed in a flat structure or within `bin` and `sbin`. Here is a list of files to give an example structure for a dynamic deployment of the *condor_schedd* daemon.

```
% tar tfz latest-i686-Linux-2.4.21-37.ELsmp.tar.gz
bin/
bin/condor_config_val
bin/condor_q
sbin/
sbin/condor_preen
sbin/condor_shadow.std
sbin/condor_starter.std
sbin/condor_schedd
sbin/condor_master
sbin/condor_gridmanager
sbin/gt4_gahp
sbin/gahp_server
sbin/condor_starter
sbin/condor_shadow
sbin/condor_c-gahp
sbin/condor_off
```

3.3 Configuration

This section describes how to configure all parts of the Condor system. General information about the configuration files and their syntax is followed by a description of settings that affect all Condor daemons and tools. The settings that control the policy under which Condor will start, suspend, resume, vacate or kill jobs are described in section 3.5 on Startd Policy Configuration.

3.3.1 Introduction to Configuration Files

The Condor configuration files are used to customize how Condor operates at a given site. The basic configuration as shipped with Condor works well for most sites.

Each Condor program will, as part of its initialization process, configure itself by calling a library routine which parses the various configuration files that might be used including pool-wide, platform-specific, and machine-specific configuration files. Environment variables may also contribute to the configuration.

The result of configuration is a list of key/value pairs. Each key is a configuration variable name, and each value is a string literal that may utilize macro substitution (as defined below). Note that the string literal value portion of a pair is not an expression, and therefore it is not evaluated. Those configuration variables that express the policy for starting and stopping of jobs appear as expressions in the configuration file. However, these expressions (for configuration) are string literals. At

appropriate times, Condor daemons and tools use these strings as expressions, parsing them in order to do evaluation.

Ordered Evaluation to Set the Configuration

Multiple files, as well as a program's environment variables determine the configuration. The order in which attributes are defined is important, as later definitions override existing definitions. The order in which the (multiple) configuration files are parsed is designed to ensure the security of the system. Attributes which must be set a specific way must appear in the last file to be parsed. This prevents both the naive and the malicious Condor user from subverting the system through its configuration. The order in which items are parsed is

1. global configuration file
2. local configuration file
3. specific environment variables prefixed with `_CONDOR_`

The locations for these files are as given in section 3.2.2 on page 121.

Some Condor tools utilize environment variables to set their configuration. These tools search for specifically-named environment variables. The variables are prefixed by the string `_CONDOR_` or `_condor_`. The tools strip off the prefix, and utilize what remains as configuration. As the use of environment variables is the last within the ordered evaluation, the environment variable definition is used. The security of the system is not compromised, as only specific variables are considered for definition in this manner, not any environment variables with the `_CONDOR_` prefix.

Configuration File Macros

Macro definitions are of the form:

```
<macro_name> = <macro_definition>
```

The macro name given on the left hand side of the definition is a case sensitive identifier. There must be white space between the macro name, the equals sign (=), and the macro definition. The macro definition is a string literal that may utilize macro substitution.

Macro invocations are of the form:

```
$(macro_name)
```

Macro definitions may contain references to other macros, even ones that are not yet defined, as long as they are eventually defined in the configuration files. All macro expansion is done after all configuration files have been parsed, with the exception of macros that reference themselves.

```
A = xxx
C = $(A)
```

is a legal set of macro definitions, and the resulting value of C is xxx. Note that C is actually bound to \$(A), not its value.

As a further example,

```
A = xxx
C = $(A)
A = YYY
```

is also a legal set of macro definitions, and the resulting value of C is YYY.

A macro may be incrementally defined by invoking itself in its definition. For example,

```
A = xxx
B = $(A)
A = $(A)YYY
A = $(A)ZZZ
```

is a legal set of macro definitions, and the resulting value of A is xxxyyyzzz. Note that invocations of a macro in its own definition are immediately expanded. \$(A) is immediately expanded in line 3 of the example. If it were not, then the definition would be impossible to evaluate.

Recursively defined macros such as

```
A = $(B)
B = $(A)
```

are not allowed. They create definitions that Condor refuses to parse.

All entries in a configuration file must have an operator, which will be an equals sign (=). Identifiers are alphanumeric characters combined with the underscore character, optionally with a subsystem name and a period as a prefix. As a special case, a line without an operator that begins with a left square bracket will be ignored. The following two-line example treats the first line as a comment, and correctly handles the second line.

```
[Condor Settings]
my_classad = [ foo=bar ]
```

To simplify pool administration, any configuration variable name may be prefixed by a subsystem (see the \$(SUBSYSTEM) macro in section 3.3.1 for the list of subsystems) and the period (.) character. For configuration variables defined this way, the value is applied to the specific subsystem. For example, the ports that Condor may use can be restricted to a range using the HIGHPORT and LOWPORT configuration variables. If the range of intended ports is different for specific daemons, this syntax may be used.

```
MASTER.LOWPORT    = 20000
MASTER.HIGHPORT   = 20100
NEGOTIATOR.LOWPORT = 22000
NEGOTIATOR.HIGHPORT = 22100
```

Note that all configuration variables may utilize this syntax, but nonsense configuration variables may result. For example, it makes no sense to define

```
NEGOTIATOR.MASTER_UPDATE_INTERVAL = 60
```

since the *condor_negotiator* daemon does not use the MASTER_UPDATE_INTERVAL variable.

It makes little sense to do so, but Condor will configure correctly with a definition such as

```
MASTER.MASTER_UPDATE_INTERVAL = 60
```

The *condor_master* uses this configuration variable, and the prefix of MASTER. causes this configuration to be specific to the *condor_master* daemon.

This syntax has been further expanded to allow for the specification of a local name on the command line using the command line option

```
-local-name <local-name>
```

This allows multiple instances of a daemon to be run by the same *condor_master* daemon, each instance with its own local configuration variable.

The ordering used to look up a variable, called <parameter name>:

1. <subsystem name>.<local name>.<parameter name>
2. <local name>.<parameter name>
3. <subsystem name>.<parameter name>
4. <parameter name>

If this local name is not specified on the command line, numbers 1 and 2 are skipped. As soon as the first match is found, the search is completed, and the corresponding value is used.

This example configures a *condor_master* to run 2 *condor_schedd* daemons. The *condor_master* daemon needs the configuration:

```
XYZZY          = $(SCHEDD)
XYZZY_ARGS     = -local-name xyzzy
DAEMON_LIST    = $(DAEMON_LIST) XYZZY
DC_DAEMON_LIST = + XYZZY
XYZZY_LOG      = $(LOG)/SchedLog.xyzzy
```

Using this example configuration, the *condor_master* starts up a second *condor_schedd* daemon, where this second *condor_schedd* daemon is passed **-local-name** *xyzzzy* on the command line.

Continuing the example, configure the *condor_schedd* daemon named *xyzzzy*. This *condor_schedd* daemon will share all configuration variable definitions with the other *condor_schedd* daemon, except for those specified separately.

```
SCHEDD.XYZZY.SCHEDD_NAME = XYZZY
SCHEDD.XYZZY.SCHEDD_LOG  = $(XYZZY_LOG)
SCHEDD.XYZZY.SPOOL       = $(SPOOL).XYZZY
```

Note that the example `SCHEDD_NAME` and `SPOOL` are specific to the *condor_schedd* daemon, as opposed to a different daemon such as the *condor_startd*. Other Condor daemons using this feature will have different requirements for which parameters need to be specified individually. This example works for the *condor_schedd*, and more local configuration can, and likely would be specified.

Also note that each daemon's log file must be specified individually, and in two places: one specification is for use by the *condor_master*, and the other is for use by the daemon itself. In the example, the *XYZZY condor_schedd* configuration variable `SCHEDD.XYZZY.SCHEDD_LOG` definition references the *condor_master* daemon's `XYZZY_LOG`.

Comments and Line Continuations

A Condor configuration file may contain comments and line continuations. A comment is any line beginning with a pound character (#). A continuation is any entry that continues across multiples lines. Line continuation is accomplished by placing the backslash character (\) at the end of any line to be continued onto another. Valid examples of line continuation are

```
START = (KeyboardIdle > 15 * $(MINUTE)) && \
((LoadAvg - CondorLoadAvg) <= 0.3)
```

and

```
ADMIN_MACHINES = condor.cs.wisc.edu, raven.cs.wisc.edu, \
stork.cs.wisc.edu, ostrich.cs.wisc.edu, \
bigbird.cs.wisc.edu
HOSTALLOW_ADMIN = $(ADMIN_MACHINES)
```

Note that a line continuation character may currently be used within a comment, so the following example does *not* set the configuration variable `FOO`:

```
# This comment includes the following line, so FOO is NOT set \
FOO = BAR
```

It is a poor idea to use this functionality, as it is likely to stop working in future Condor releases.

Executing a Program to Produce Configuration Macros

Instead of reading from a file, Condor may run a program to obtain configuration macros. The vertical bar character (|) as the last character defining a file name provides the syntax necessary to tell Condor to run a program. This syntax may only be used in the definition of the `CONDOR_CONFIG` environment variable, or the `LOCAL_CONFIG_FILE` configuration variable.

The command line for the program is formed by the characters preceding the vertical bar character. The standard output of the program is parsed as a configuration file would be.

An example:

```
LOCAL_CONFIG_FILE = /bin/make_the_config|
```

Program `/bin/make_the_config` is executed, and its output is the set of configuration macros.

Note that either a program is executed to generate the configuration macros or the configuration is read from one or more files. The syntax uses space characters to separate command line elements, if an executed program produces the configuration macros. Space characters would otherwise separate the list of files. This syntax does not permit distinguishing one from the other, so only one may be specified.

Pre-Defined Macros

Condor provides pre-defined macros that help configure Condor. Pre-defined macros are listed as `$(macro_name)`.

This first set are entries whose values are determined at run time and cannot be overwritten. These are inserted automatically by the library routine which parses the configuration files.

\$(FULL_HOSTNAME) The fully qualified host name of the local machine, which is host name plus domain name.

\$(HOSTNAME) The host name of the local machine (no domain name).

\$(IP_ADDRESS) The ASCII string version of the local machine's IP address.

\$(TILDE) The full path to the home directory of the Unix user condor, if such a user exists on the local machine.

\$(SUBSYSTEM) The subsystem name of the daemon or tool that is evaluating the macro. This is a unique string which identifies a given daemon within the Condor system. The possible subsystem names are:

- STARTD
- SCHEDD
- MASTER
- COLLECTOR
- NEGOTIATOR
- KBDD
- SHADOW
- STARTER
- CKPT_SERVER
- SUBMIT
- GRIDMANAGER
- TOOL
- STORK
- HAD
- REPLICATION
- QUILL
- DBMSD
- JOB_ROUTER
- LEASEMANAGER

This second set of macros are entries whose default values are determined automatically at run time but which can be overwritten.

\$(ARCH) Defines the string used to identify the architecture of the local machine to Condor. The *condor_startd* will advertise itself with this attribute so that users can submit binaries compiled for a given platform and force them to run on the correct machines. *condor_submit* will append a requirement to the job ClassAd that it must run on the same ARCH and OPSYS of the machine where it was submitted, unless the user specifies ARCH and/or OPSYS explicitly in their submit file. See the *condor_submit* manual page on page 795 for details.

\$(OPSYS) Defines the string used to identify the operating system of the local machine to Condor. If it is not defined in the configuration file, Condor will automatically insert the operating system of this machine as determined by *uname*.

\$(UNAMEARCH) The architecture as reported by *uname(2)*'s machine field. Always the same as ARCH on Windows.

\$(UNAMEOPSYS) The operating system as reported by *uname(2)*'s sysname field. Always the same as OPSYS on Windows.

\$(PID) The process ID for the daemon or tool.

`$(PPID)` The process ID of the parent process for the daemon or tool.

`$(USERNAME)` The user name of the UID of the daemon or tool. For daemons started as root, but running under another UID (typically the user condor), this will be the other UID.

`$(FILESYSTEMDOMAIN)` Defaults to the fully qualified host name of the machine it is evaluated on. See section 3.3.7, Shared File System Configuration File Entries for the full description of its use and under what conditions you would want to change it.

`$(UIDDOMAIN)` Defaults to the fully qualified host name of the machine it is evaluated on. See section 3.3.7 for the full description of this configuration variable.

Since `$(ARCH)` and `$(OPSYS)` will automatically be set to the correct values, we recommend that you do not overwrite them. Only do so if you know what you are doing.

3.3.2 The Special Configuration Macros `$ENV()`, `$RANDOM_CHOICE()`, and `$RANDOM_INTEGER()`

References to the Condor process's environment are allowed in the configuration files. Environment references use the `ENV` macro and are of the form:

```
$ENV(environment_variable_name)
```

For example,

```
A = $ENV(HOME)
```

binds `A` to the value of the `HOME` environment variable. Environment references are not currently used in standard Condor configurations. However, they can sometimes be useful in custom configurations.

This same syntax is used in the `RANDOM_CHOICE()` macro to allow a random choice of a parameter within a configuration file. These references are of the form:

```
$RANDOM_CHOICE(list of parameters)
```

This allows a random choice within the parameter list to be made at configuration time. Of the list of parameters, one is chosen when encountered during configuration. For example, if one of the integers 0-8 (inclusive) should be randomly chosen, the macro usage is

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

The `RANDOM_INTEGER()` macro is similar to the `RANDOM_CHOICE()` macro, and is used to select a random integer within a configuration file. References are of the form:

```
$RANDOM_INTEGER(min, max [, step])
```

A random integer within the range `min` and `max`, inclusive, is selected at configuration time. The optional `step` parameter controls the stride within the range, and it defaults to the value 1. For example, to randomly chose an even integer in the range 0-8 (inclusive), the macro usage is

```
$RANDOM_INTEGER(0, 8, 2)
```

See section 7.2 on page 555 for an actual use of this specialized macro.

3.3.3 Condor-wide Configuration File Entries

This section describes settings which affect all parts of the Condor system. Other system-wide settings can be found in section 3.3.6 on “Network-Related Configuration File Entries”, and section 3.3.7 on “Shared File System Configuration File Entries”.

CONDOR_HOST This macro may be used to define the `$(NEGOTIATOR_HOST)` and is used to define the `$(COLLECTOR_HOST)` macro. Normally the *condor_collector* and *condor_negotiator* would run on the same machine. If for some reason they were not run on the same machine, `$(CONDOR_HOST)` would not be needed. Some of the host-based security macros use `$(CONDOR_HOST)` by default. See section 3.6.9, on Setting up IP/host-based security in Condor for details.

COLLECTOR_HOST The host name of the machine where the *condor_collector* is running for your pool. Normally, it is defined relative to the `$(CONDOR_HOST)` macro. There is no default value for this macro; `COLLECTOR_HOST` must be defined for the pool to work properly.

In addition to defining the host name, this setting can optionally be used to specify the network port of the *condor_collector*. The port is separated from the host name by a colon (`:`). For example,

```
COLLECTOR_HOST = $(CONDOR_HOST):1234
```

If no port is specified, the default port of 9618 is used. Using the default port is recommended for most sites. It is only changed if there is a conflict with another service listening on the same network port. For more information about specifying a non-standard port for the *condor_collector* daemon, see section 3.7.1 on page 326.

NEGOTIATOR_HOST This configuration variable is no longer used. It previously defined the host name of the machine where the *condor_negotiator* is running. At present, the port where the *condor_negotiator* is listening is dynamically allocated.

CONDOR_VIEW_HOST The host name of the machine, optionally appended by a colon and the port number, where the CondorView server is running. This service is optional, and requires

additional configuration to enable it. There is no default value for `CONDOR_VIEW_HOST`. If `CONDOR_VIEW_HOST` is not defined, no CondorView server is used. See section 3.12.5 on page 396 for more details.

SCHEDD_HOST The host name of the machine where the *condor_schedd* is running for your pool. This is the host that queues submitted jobs. Note that, in most condor installations, there is a *condor_schedd* running on each host from which jobs are submitted. The default value of `SCHEDD_HOST` is the current host. For most pools, this macro is not defined.

RELEASE_DIR The full path to the Condor release directory, which holds the `bin`, `etc`, `lib`, and `sbin` directories. Other macros are defined relative to this one. There is no default value for `RELEASE_DIR`.

BIN This directory points to the Condor directory where user-level programs are installed. It is usually defined relative to the `$(RELEASE_DIR)` macro. There is no default value for `BIN`.

LIB This directory points to the Condor directory where libraries used to link jobs for Condor's standard universe are stored. The *condor_compile* program uses this macro to find these libraries, so it must be defined for *condor_compile* to function. `$(LIB)` is usually defined relative to the `$(RELEASE_DIR)` macro, and has no default value.

LIBEXEC This directory points to the Condor directory where support commands that Condor needs will be placed. Do not add this directory to a user or system-wide path.

INCLUDE This directory points to the Condor directory where header files reside. `$(INCLUDE)` would usually be defined relative to the `$(RELEASE_DIR)` configuration macro. There is no default value, but if defined, it can make inclusion of necessary header files for compilation of programs (such as those programs that use `libcondorapi.a`) easier through the use of *condor_config_val*.

SBIN This directory points to the Condor directory where Condor's system binaries (such as the binaries for the Condor daemons) and administrative tools are installed. Whatever directory `$(SBIN)` points to ought to be in the `PATH` of users acting as Condor administrators. `SBIN` has no default value.

LOCAL_DIR The location of the local Condor directory on each machine in your pool. One common option is to use the condor user's home directory which may be specified with `$(TILDE)`. There is no default value for `LOCAL_DIR`. For example:

```
LOCAL_DIR = $(tilde)
```

On machines with a shared file system, where either the `$(TILDE)` directory or another directory you want to use is shared among all machines in your pool, you might use the `$(HOSTNAME)` macro and have a directory with many subdirectories, one for each machine in your pool, each named by host names. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

LOG Used to specify the directory where each Condor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the `$(LOG)` macro by default. The log directory also acts as the current working directory of the Condor daemons as the run, so if one of them should produce a core file for any reason, it would be placed in the directory defined by this macro. `LOG` is required to be defined. Normally, `$(LOG)` is defined in terms of `$(LOCAL_DIR)`.

SPOOL The spool directory is where certain files used by the *condor_schedd* are stored, such as the job queue file and the initial executables of any jobs that have been submitted. In addition, for systems not using a checkpoint server, all the checkpoint files from jobs that have been submitted from a given machine will be store in that machine's spool directory. Therefore, you will want to ensure that the spool directory is located on a partition with enough disk space. If a given machine is only set up to execute Condor jobs and not submit them, it would not need a spool directory (or this macro defined). There is no default value for `SPOOL`, and the *condor_schedd* will not function without it `SPOOL` defined. Normally, `$(SPOOL)` is defined in terms of `$(LOCAL_DIR)`.

EXECUTE This directory acts as a place to create the scratch directory of any Condor job that is executing on the local machine. The scratch directory is the destination of any input files that were specified for transfer. It also serves as the job's working directory if the job is using file transfer mode and no other working directory was specified. If a given machine is set up to only submit jobs and not execute them, it would not need an execute directory, and this macro need not be defined. There is no default value for `EXECUTE`, and the *condor_startd* will not function if `EXECUTE` is undefined. Normally, `$(EXECUTE)` is defined in terms of `$(LOCAL_DIR)`. To customize the execute directory independently for each batch slot, use `SLOTx_EXECUTE`.

SLOTx_EXECUTE Specifies an execute directory for use by a specific batch slot. (*x* should be the number of the batch slot, such as 1, 2, 3, etc.) This execute directory serves the same purpose as `EXECUTE`, but it allows you to configure the directory independently for each batch slot. Having slots each using a different partition would be useful, for example, in preventing one job from filling up the same disk that other jobs are trying to write to. If this parameter is undefined for a given batch slot, it will use `EXECUTE` as the default. Note that each slot will advertise `TotalDisk` and `Disk` for the partition containing its execute directory.

LOCAL_CONFIG_FILE Identifies the location of the local, machine-specific configuration file for each machine in the pool. The two most common choices would be putting this file in the `$(LOCAL_DIR)`, or putting all local configuration files for the pool in a shared directory, each one named by host name. For example,

```
LOCAL_CONFIG_FILE = $(LOCAL_DIR)/condor_config.local
```

or,

```
LOCAL_CONFIG_FILE = $(release_dir)/etc/$(hostname).local
```

or, not using the release directory

```
LOCAL_CONFIG_FILE = /full/path/to/configs/$(hostname).local
```

The value of `$(LOCAL_CONFIG_FILE)` is treated as a list of files, not a single file. The items in the list are delimited by either commas or space characters. This allows the specification of multiple files as the local configuration file, each one processed in the order given (with parameters set in later files overriding values from previous files). This allows the use of one global configuration file for multiple platforms in the pool, defines a platform-specific configuration file for each platform, and uses a local configuration file for each machine. If the list of files is changed in one of the later read files, the new list replaces the old list, but any files that have already been processed remain processed, and are removed from the new list if they are present to prevent cycles. See section 3.3.1 on page 147 for directions on using a program to generate the configuration macros that would otherwise reside in one or more files as described here. If `LOCAL_CONFIG_FILE` is not defined, no local configuration files are processed. For more information on this, see section 3.12.2 about Configuring Condor for Multiple Platforms on page 391.

REQUIRE_LOCAL_CONFIG_FILE A boolean value that defaults to `True`. When `True`, Condor exits with an error, if any file listed in `LOCAL_CONFIG_FILE` cannot be read. A value of `False` allows local configuration files to be missing. This is most useful for sites that have both large numbers of machines in the pool and a local configuration file that uses the `$(HOSTNAME)` macro in its definition. Instead of having an empty file for every host in the pool, files can simply be omitted.

LOCAL_CONFIG_DIR Beginning in Condor 6.7.18, a directory may be used as a container for local configuration files. The files found in the directory are sorted into lexicographical order, and then each file is treated as though it was listed in `LOCAL_CONFIG_FILE`. `LOCAL_CONFIG_DIR` is processed before any files listed in `LOCAL_CONFIG_FILE`, and is checked again after processing the `LOCAL_CONFIG_FILE` list. It is a list of directories, and each directory is processed in the order it appears in the list. The process is not recursive, so any directories found inside the directory being processed are ignored.

CONDOR_IDS The User ID (UID) and Group ID (GID) pair that the Condor daemons should run as, if the daemons are spawned as root. This value can also be specified in the `CONDOR_IDS` environment variable. If the Condor daemons are not started as root, then neither this `CONDOR_IDS` configuration macro nor the `CONDOR_IDS` environment variable are used. The value is given by two integers, separated by a period. For example, `CONDOR_IDS = 1234.1234`. If this pair is not specified in either the configuration file or in the environment, and the Condor daemons are spawned as root, then Condor will search for a condor user on the system, and run as that user's UID and GID. See section 3.6.11 on UIDs in Condor for more details.

CONDOR_ADMIN The email address that Condor will send mail to if something goes wrong in your pool. For example, if a daemon crashes, the *condor_master* can send an *obituary* to this address with the last few lines of that daemon's log file and a brief message that describes what signal or exit status that daemon exited with. There is no default value for CONDOR_ADMIN.

CONDOR_SUPPORT_EMAIL The email address to be included at the bottom of all email Condor sends out under the label "Email address of the local Condor administrator:". This is the address where Condor users at your site should send their questions about Condor and get technical support. If this setting is not defined, Condor will use the address specified in CONDOR_ADMIN (described above).

MAIL The full path to a mail sending program that uses **-s** to specify a subject for the message. On all platforms, the default shipped with Condor should work. Only if you installed things in a non-standard location on your system would you need to change this setting. There is no default value for MAIL, and the *condor_schedd* will not function unless MAIL is defined.

RESERVED_SWAP Determines how much swap space you want to reserve for your own machine. Condor will not start up more *condor_shadow* processes if the amount of free swap space on your machine falls below this level. RESERVED_SWAP is specified in megabytes. The default value of RESERVED_SWAP is 5 megabytes.

RESERVED_DISK Determines how much disk space you want to reserve for your own machine. When Condor is reporting the amount of free disk space in a given partition on your machine, it will always subtract this amount. An example is the *condor_startd*, which advertises the amount of free space in the `$(EXECUTE)` directory. The default value of RESERVED_DISK is zero.

LOCK Condor needs to create lock files to synchronize access to various log files. Because of problems with network file systems and file locking over the years, we *highly* recommend that you put these lock files on a local partition on each machine. If you do not have your `$(LOCAL_DIR)` on a local partition, be sure to change this entry.

Whatever user or group Condor is running as needs to have write access to this directory. If you are not running as root, this is whatever user you started up the *condor_master* as. If you are running as root, and there is a condor account, it is most likely condor. Otherwise, it is whatever you set in the CONDOR_IDS environment variable, or whatever you define in the CONDOR_IDS setting in the Condor config files. See section 3.6.11 on UIDs in Condor for details.

If no value for LOCK is provided, the value of LOG is used.

HISTORY Defines the location of the Condor history file, which stores information about all Condor jobs that have completed on a given machine. This macro is used by both the *condor_schedd* which appends the information and *condor_history*, the user-level program used to view the history file. This configuration macro is given the default value of `$(SPOOL)/history` in the default configuration. If not defined, no history file is kept.

ENABLE_HISTORY_ROTATION If this is defined to be true, then the history file will be rotated. If it is false, then it will not be rotated, and it will grow indefinitely, to the limits allowed by the operating system. If this is not defined, it is assumed to be true. The rotated files will be stored in the same directory as the history file.

MAX_HISTORY_LOG Defines the maximum size for the history file, in bytes. It defaults to 20MB. This parameter is only used if history file rotation is enabled.

MAX_HISTORY_ROTATIONS When history file rotation is turned on, this controls how many backup files there are. It default to 2, which means that there may be up to three history files (two backups, plus the history file that is being currently written to). When the history file is rotated, and this rotation would cause the number of backups to be too large, the oldest file is removed.

MAX_JOB_QUEUE_LOG_ROTATIONS The schedd periodically rotates the job queue database file in order to save disk space. This option controls how many rotated files are saved. It defaults to 1, which means there may be up to two history files (the previous one, which was rotated out of use, and the current one that is being written to). When the job queue file is rotated, and this rotation would cause the number of backups to be larger the the maximum specified, the oldest file is removed. The primary reason to save one or more rotated job queue files is if you are using Quill, and you want to ensure that Quill keeps an accurate history of all events logged in the job queue file. Quill keeps track of where it last left off when reading logged events, so when the file is rotated, Quill will resume reading from where it last left off, provided that the rotated file still exists. If Quill finds that it needs to read events from a rotated file that has been deleted, it will be forced to skip the missing events and resume reading in the next chronological job queue file that can be found. Such an event should not lead to an inconsistency in Quill's view of the current queue contents, but it would create a inconsistency in Quill's record of the history of the job queue.

DEFAULT_DOMAINNAME The value to be appended to a machine's host name, representing a domain name, which Condor then uses to form a fully qualified host name. This is required if there is no fully qualified host name in file `/etc/hosts` or in NIS. Set the value in the global configuration file, as Condor may depend on knowing this value in order to locate the local configuration file(s). The default value as given in the sample configuration file of the Condor download is bogus, and must be changed. If this variable is removed from the global configuration file, or if the definition is empty, then Condor attempts to discover the value.

NO_DNS A boolean value that defaults to `False`. When `True`, Condor constructs host names using the host's IP address together with the value defined for `DEFAULT_DOMAINNAME`.

CM_IP_ADDR If neither `COLLECTOR_HOST` nor `COLLECTOR_IP_ADDR` macros are defined, then this macro will be used to determine the IP address of the central manager (collector daemon). This macro is defined by an IP address.

EMAIL_DOMAIN By default, if a user does not specify `notify_user` in the submit description file, any email Condor sends about that job will go to `"username@UID.DOMAIN"`. If your machines all share a common UID domain (so that you would set `UID.DOMAIN` to be the same across all machines in your pool), but email to `user@UID.DOMAIN` is not the right place for Condor to send email for your site, you can define the default domain to use for email. A common example would be to set `EMAIL_DOMAIN` to the fully qualified host name of each machine in your pool, so users submitting jobs from a specific machine would get email sent to `user@machine.your.domain`, instead of `user@your.domain`. You would do this

by setting `EMAIL_DOMAIN` to `$(FULL_HOSTNAME)`. In general, you should leave this setting commented out unless two things are true: 1) `UID_DOMAIN` is set to your domain, not `$(FULL_HOSTNAME)`, and 2) email to `user@UID_DOMAIN` will not work.

CREATE_CORE_FILES Defines whether or not Condor daemons are to create a core file in the `LOG` directory if something really bad happens. It is used to set the resource limit for the size of a core file. If not defined, it leaves in place whatever limit was in effect when the Condor daemons (normally the *condor_master*) were started. This allows Condor to inherit the default system core file generation behavior at start up. For Unix operating systems, this behavior can be inherited from the parent shell, or specified in a shell script that starts Condor. If this parameter is set and `True`, the limit is increased to the maximum. If it is set to `False`, the limit is set at 0 (which means that no core files are created). Core files greatly help the Condor developers debug any problems you might be having. By using the parameter, you do not have to worry about tracking down where in your boot scripts you need to set the core limit before starting Condor. You set the parameter to whatever behavior you want Condor to enforce. This parameter defaults to undefined to allow the initial operating system default value to take precedence, and is commented out in the default configuration file.

CKPT_PROBE Defines the path and executable name of the helper process Condor will use to determine information for the `CheckpointPlatform` attribute in the machine's `ClassAd`. The default value is `$(LIBEXEC)/condor_ckpt_probe`.

ABORT_ON_EXCEPTION When Condor programs detect a fatal internal exception, they normally log an error message and exit. If you have turned on `CREATE_CORE_FILES`, in some cases you may also want to turn on `ABORT_ON_EXCEPTION` so that core files are generated when an exception occurs. Set the following to `True` if that is what you want.

Q_QUERY_TIMEOUT Defines the timeout (in seconds) that *condor_q* uses when trying to connect to the *condor_schedd*. Defaults to 20 seconds.

DEAD_COLLECTORMAX_AVOIDANCE_TIME Defines the interval of time (in seconds) between checks for a failed primary *condor_collector* daemon. If connections to the dead primary *condor_collector* take very little time to fail, new attempts to query the primary *condor_collector* may be more frequent than the specified maximum avoidance time. The default value equals one hour. This variable has relevance to flocked jobs, as it defines the maximum time they may be reporting to the primary *condor_collector* without the *condor_negotiator* noticing.

PASSWD_CACHE_REFRESH Condor can cause NIS servers to become overwhelmed by queries for uid and group information in large pools. In order to avoid this problem, Condor caches UID and group information internally. This integer value allows pool administrators to specify (in seconds) how long Condor should wait until refreshes a cache entry. The default is set to 300 seconds, or 5 minutes, plus a random number of seconds between 0 and 60 to avoid having lots of processes refreshing at the same time. This means that if a pool administrator updates the user or group database (for example, `/etc/passwd` or `/etc/group`), it can take up to 6 minutes before Condor will have the updated information. This caching feature can be disabled by setting the refresh interval to 0. In addition, the cache can also be flushed explicitly by running the command

```
condor_reconfig -full
```

This configuration variable has no effect on Windows.

SYSAPIGETLOADAVG If set to False, then Condor will not attempt to compute the load average on the system, and instead will always report the system load average to be 0.0. Defaults to True.

NETWORK_MAX_PENDING_CONNECTS This specifies a limit to the maximum number of simultaneous network connection attempts. This is primarily relevant to *condor_schedd*, which may try to connect to large numbers of startds when claiming them. The negotiator may also connect to large numbers of startds when initiating security sessions used for sending MATCH messages. On Unix, the default for this parameter is eighty percent of the process file descriptor limit. On windows, the default is 1600.

WANT_UDP_COMMAND_SOCKET This setting, added in version 6.9.5, controls if Condor daemons should create a UDP command socket in addition to the TCP command socket (which is required). The default is True, and modifying it requires restarting all Condor daemons, not just a *condor_reconfig* or SIGHUP.

Normally, updates sent to the *condor_collector* use UDP, in addition to certain keep alive messages and other non-essential communication. However, in certain situations, it might be desirable to disable the UDP command port (for example, to reduce the number of ports represented by a GCB broker, etc).

Unfortunately, due to a limitation in how these command sockets are created, it is not possible to define this setting on a per-daemon basis, for example, by trying to set `STARTD.WANT_UDP_COMMAND_SOCKET`. At least for now, this setting must be defined machine wide to function correctly.

If this setting is set to true on a machine running a *condor_collector*, the pool should be configured to use TCP updates to that collector (see section 3.7.4 on page 345 for more information).

3.3.4 Daemon Logging Configuration File Entries

These entries control how and where the Condor daemons write to log files. Many of the entries in this section represents multiple macros. There is one for each subsystem (listed in section 3.3.1). The macro name for each substitutes <SUBSYS> with the name of the subsystem corresponding to the daemon.

<SUBSYS>LOG The name of the log file for a given subsystem. For example, `$(STARTD_LOG)` gives the location of the log file for *condor_startd*.

MAX<SUBSYS>LOG Controls the maximum length in bytes to which a log will be allowed to grow. Each log file will grow to the specified length, then be saved to a file with the suffix `.old`. The `.old` files are overwritten each time the log is saved, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file. A value of 0 specifies that the file may grow without bounds. The default is 1 Mbyte.

TRUNC_<SUBSYS>_LOG_ON_OPEN If this macro is defined and set to `True`, the affected log will be truncated and started from an empty file with each invocation of the program. Otherwise, new invocations of the program will append to the previous log file. By default this setting is `False` for all daemons.

<SUBSYS>_LOCK This macro specifies the lock file used to synchronize append operations to the log file for this subsystem. It must be a separate file from the `$ (<SUBSYS>_LOG)` file, since the `$ (<SUBSYS>_LOG)` file may be rotated and you want to be able to synchronize access across log file rotations. A lock file is only required for log files which are accessed by more than one process. Currently, this includes only the `SHADOW` subsystem. This macro is defined relative to the `$ (LOCK)` macro.

FILE_LOCK_VIA_MUTEX This macro setting only works on Win32 – it is ignored on Unix. If set to be `True`, then log locking is implemented via a kernel mutex instead of via file locking. On Win32, mutex access is FIFO, while obtaining a file lock is non-deterministic. Thus setting to `True` fixes problems on Win32 where processes (usually shadows) could starve waiting for a lock on a log file. Defaults to `True` on Win32, and is always `False` on Unix.

ENABLE_USER_LOG_LOCKING When `True` (the default value), a user's job log (as specified in a submit description file) will be locked before being written to. If `False`, Condor will not lock the file before writing.

TOUCH_LOG_INTERVAL The time interval in seconds between when daemons touch their log files. The change in last modification time for the log file is useful when a daemon restarts after failure or shut down. The last modification date is printed, and it provides an upper bound on the length of time that the daemon was not running. Defaults to 60 seconds.

LOGS_USE_TIMESTAMP This macro controls how the current time is formatted at the start of each line in the daemon log files. When `True`, the Unix time is printed (number of seconds since 00:00:00 UTC, January 1, 1970). When `False` (the default value), the time is printed like so: `<Month>/<Day> <Hour>:<Minute>:<Second>` in the local timezone.

<SUBSYS>_DEBUG All of the Condor daemons can produce different levels of output depending on how much information is desired. The various levels of verbosity for a given daemon are determined by this macro. All daemons have the default level `D_ALWAYS`, and log messages for that level will be printed to the daemon's log, regardless of this macro's setting. Settings are a comma- or space-separated list of the following values:

D_ALL This flag turns on *all* debugging output by enabling all of the debug levels at once. There is no need to list any other debug levels in addition to `D_ALL`; doing so would be redundant. Be warned: this will generate about a *HUGE* amount of output. To obtain a higher level of output than the default, consider using `D_FULLDEBUG` before using this option.

D_FULLDEBUG This level provides verbose output of a general nature into the log files. Frequent log messages for very specific debugging purposes would be excluded. In those cases, the messages would be viewed by having that another flag and `D_FULLDEBUG` both listed in the configuration file.

- D_DAEMONCORE** Provides log file entries specific to DaemonCore, such as timers the daemons have set and the commands that are registered. If both `D_FULLDEBUG` and `D_DAEMONCORE` are set, expect *very* verbose output.
- D_PRIV** This flag provides log messages about the *privilege state* switching that the daemons do. See section 3.6.11 on UIDs in Condor for details.
- D_COMMAND** With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command comes in. The name and integer of the command, whether the command was sent via UDP or TCP, and where the command was sent from are all logged. Because the messages about the command used by *condor_kbdd* to communicate with the *condor_startd* whenever there is activity on the X server, and the command used for keep-alives are both only printed with `D_FULLDEBUG` enabled, it is best if this setting is used for all daemons.
- D_LOAD** The *condor_startd* keeps track of the load average on the machine where it is running. Both the general system load average, and the load average being generated by Condor's activity there are determined. With this flag set, the *condor_startd* will log a message with the current state of both of these load averages whenever it computes them. This flag only affects the *condor_startd*.
- D_KEYBOARD** With this flag set, the *condor_startd* will print out a log message with the current values for remote and local keyboard idle time. This flag affects only the *condor_startd*.
- D_JOB** When this flag is set, the *condor_startd* will send to its log file the contents of any job ClassAd that the *condor_schedd* sends to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.
- D_MACHINE** When this flag is set, the *condor_startd* will send to its log file the contents of its resource ClassAd when the *condor_schedd* tries to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.
- D_SYSCALLS** This flag is used to make the *condor_shadow* log remote syscall requests and return values. This can help track down problems a user is having with a particular job by providing the system calls the job is performing. If any are failing, the reason for the failure is given. The *condor_schedd* also uses this flag for the server portion of the queue management code. With `D_SYSCALLS` defined in `SCHEDD_DEBUG` there will be verbose logging of all queue management operations the *condor_schedd* performs.
- D_MATCH** When this flag is set, the *condor_negotiator* logs a message for every match.
- D_NETWORK** When this flag is set, all Condor daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive. This flag is not yet fully supported in the *condor_shadow*.
- D_HOSTNAME** When this flag is set, the Condor daemons and/or tools will print verbose messages explaining how they resolve host names, domain names, and IP addresses. This is useful for sites that are having trouble getting Condor to work because of problems with DNS, NIS or other host name resolving systems in use.
- D_CKPT** When this flag is set, the Condor process checkpoint support code, which is linked into a STANDARD universe user job, will output some low-level details about the checkpoint procedure into the `$(SHADOW_LOG)`.

D_SECURITY This flag will enable debug messages pertaining to the setup of secure network communication, including messages for the negotiation of a socket authentication mechanism, the management of a session key cache, and messages about the authentication process itself. See section 3.6.1 for more information about secure communication configuration.

D_PROCFAMILY Condor often times needs to manage an entire family of processes, (that is, a process and all descendants of that process). This debug flag will turn on debugging output for the management of families of processes.

D_ACCOUNTANT When this flag is set, the *condor_negotiator* will output debug messages relating to the computation of user priorities (see section 3.4).

D_PROTOCOL Enable debug messages relating to the protocol for Condor's matchmaking and resource claiming framework.

D_PID This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If **D_PID** is set, Condor will always print out the process identifier (PID) of the process writing each line to the log file. This is especially helpful for Condor daemons that can fork multiple helper-processes (such as the *condor_schedd* or *condor_collector*) so the log file will clearly show which thread of execution is generating each log message.

D_FDS This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If **D_FDS** is set, Condor will always print out the file descriptor that the open of the log file was allocated by the operating system. This can be helpful in debugging Condor's use of system file descriptors as it will generally track the number of file descriptors that Condor has open.

ALL_DEBUG Used to make all subsystems share a debug flag. Set the parameter **ALL_DEBUG** instead of changing all of the individual parameters. For example, to turn on all debugging in all subsystems, set **ALL_DEBUG = D_ALL**.

TOOL_DEBUG Uses the same values (debugging levels) as **<SUBSYS>_DEBUG** to describe the amount of debugging information sent to *stderr* for Condor tools.

SUBMIT_DEBUG Uses the same values (debugging levels) as **<SUBSYS>_DEBUG** to describe the amount of debugging information sent to *stderr* for *condor_submit*.

Log files may optionally be specified per debug level as follows:

<SUBSYS>_<LEVEL>_LOG This is the name of a log file for messages at a specific debug level for a specific subsystem. If the debug level is included in **\$(<SUBSYS>_DEBUG)**, then all messages of this debug level will be written both to the **\$(<SUBSYS>_LOG)** file and the **\$(<SUBSYS>_<LEVEL>_LOG)** file. For example, **\$(<SHADOW_SYSCALLS>_LOG)** specifies a log file for all remote system call debug messages.

MAX_<SUBSYS>_<LEVEL>_LOG Similar to **MAX_<SUBSYS>_LOG**.

TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN Similar to TRUNC_<SUBSYS>_LOG_ON_OPEN.

The following macros control where and what is written to the event log, a file that receives job user log events, but across all users and user's jobs.

EVENT_LOG The full path and file name of the event log. There is no default value for this variable, so no event log will be written, if not defined.

EVENT_LOG_MAX_SIZE Controls the maximum length in bytes to which the event log will be allowed to grow. The log file will grow to the specified length, then be saved to a file with the suffix .old. The .old files are overwritten each time the log is saved. A value of 0 specifies that the file may grow without bounds (and disables rotation). The default is 1 Mbyte. For backwards compatibility, MAX_EVENT_LOG will be used if EVENT_LOG_MAX_SIZE is not defined. If EVENT_LOG is not defined, this parameter has no affect.

MAX_EVENT_LOG See EVENT_LOG_MAX_SIZE.

EVENT_LOG_MAX_ROTATIONS Controls the maximum number of rotations of the event log that will be stored. If this value is 1 (the default), the event log will be rotated to a “.old” file as described above. However, if this is greater than 1, then multiple rotation files will be stores, up to EVENT_LOG_MAX_ROTATIONS of them. These files will be named, instead of the “.old” suffix, “.1”, “.2”, with the “.1” being the most recent rotation. This is an integer parameter with a default value of 1. If EVENT_LOG is not defined, or if EVENT_LOG_MAX_SIZE has a value of 0 (which disables event log rotation), this parameter has no affect.

EVENT_LOG_ROTATION_LOCK Controls the lock file that will be used to ensure that, when rotating files, the rotation is done by a single process. This is a string parameter; it's default value is the file path of the event log itself, with a “.lock” appended. If EVENT_LOG is not defined, or if EVENT_LOG_MAX_SIZE has a value of 0 (which disables event log rotation), this parameter has no affect.

EVENT_LOG_FSYNC A boolean value that controls whether Condor will perform an `fsync()` after writing each event to the event log. When `True`, an `fsync()` operation is performed after each event. This `fsync()` operation forces the operating system to synchronize the updates to the event log to the disk, but can negatively affect the performance of the system. Defaults to `False`.

EVENT_LOG_LOCKING A boolean value that defaults to `True`. When `True`, the event log (as specified by `EVENT_LOG`) will be locked before being written to. When `False`, Condor does not lock the file before writing.

EVENT_LOG_USE_XML A boolean value that defaults to `False`. When `True`, events are logged in XML format. If `EVENT_LOG` is not defined, this parameter has no affect.

EVENT_LOG_JOB_AD_INFORMATION_ATTRS A comma-separated list of job ClassAd attributes, whose evaluated values form a new event, the `JobAdInformationEvent`. This new event is placed in the event log in addition to each logged event. If `EVENT_LOG` is not defined, this parameter has no affect.

3.3.5 DaemonCore Configuration File Entries

Please read section 3.9 for details on DaemonCore. There are certain configuration file settings that DaemonCore uses which affect all Condor daemons (except the checkpoint server, standard universe shadow, and standard universe starter, none of which use DaemonCore).

HOSTALLOW . . All macros that begin with either `HOSTALLOW` or `HOSTDENY` are settings for Condor's host-based security. See section 3.6.9 on Setting up IP/host-based security in Condor for details on these macros and how to configure them.

ENABLE_RUNTIME_CONFIG The *condor_config_val* tool has an option **-rset** for dynamically setting run time configuration values (which only effect the in-memory configuration variables). Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, Condor administrators must specifically enable it by setting `ENABLE_RUNTIME_CONFIG` to `True`, and specify what configuration variables can be changed using the `SETTABLE_ATTRS...` family of configuration options (described below). Defaults to `False`.

ENABLE_PERSISTENT_CONFIG The *condor_config_val* tool has a **-set** option for dynamically setting persistent configuration values. These values override options in the normal Condor configuration files. Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, Condor administrators must specifically enable it by setting `ENABLE_PERSISTENT_CONFIG` to `True`, creating a directory where the Condor daemons will hold these dynamically-generated persistent configuration files (declared using `PERSISTENT_CONFIG_DIR`, described below) and specify what configuration variables can be changed using the `SETTABLE_ATTRS...` family of configuration options (described below). Defaults to `False`.

PERSISTENT_CONFIG_DIR Directory where daemons should store dynamically-generated persistent configuration files (used to support *condor_config_val -set*) This directory should **only** be writable by root, or the user the Condor daemons are running as (if non-root). There is no default, administrators that wish to use this functionality must create this directory and define this setting. This directory must not be shared by multiple Condor installations, though it can be shared by all Condor daemons on the same host. Keep in mind that this directory should not be placed on an NFS mount where "root-squashing" is in effect, or else Condor daemons running as root will not be able to write to them. A directory (only writable by root) on the local file system is usually the best location for this directory.

SETTABLE_ATTRS . . All macros that begin with `SETTABLE_ATTRS` or `<SUBSYS>_SETTABLE_ATTRS` are settings used to restrict the configuration values that can be changed using the *condor_config_val* command. Section 3.6.9 on Setting up IP/Host-Based Security in Condor for details on these macros and how to configure them. In particular, section 3.6.9 on page 307 contains details specific to these macros.

SHUTDOWN_GRACEFUL_TIMEOUT Determines how long Condor will allow daemons try their graceful shutdown methods before they do a hard shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

<SUBSYS>ADDRESS_FILE A complete path to a file that is to contain an IP address and port number for a daemon. Every Condor daemon that uses DaemonCore has a command port where commands are sent. The IP/port of the daemon is put in that daemon's ClassAd, so that other machines in the pool can query the *condor_collector* (which listens on a well-known port) to find the address of a given daemon on a given machine. When tools and daemons are all executing on the same single machine, communications do not require a query of the *condor_collector* daemon. Instead, they look in a file on the local disk to find the IP/port. This macro causes daemons to write the IP/port of their command socket to a specified file. In this way, local tools will continue to operate, even if the machine running the *condor_collector* crashes. Using this file will also generate slightly less network traffic in the pool, since tools including *condor_q* and *condor_rm* do not need to send any messages over the network to locate the *condor_schedd* daemon. This macro is not necessary for the *condor_collector* daemon, since its command socket is at a well-known port.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<SUBSYS>DAEMONAD_FILE A complete path to a file that is to contain the ClassAd for a daemon. When the daemon sends a ClassAd describing itself to the *condor_collector*, it will also place a copy of the ClassAd in this file. Currently, this setting only works for the *condor_schedd* (that is SCHEDD_DAEMONAD_FILE) and is required for Quill.

<SUBSYS>ATTRS or <SUBSYS>EXPRS Allows any DaemonCore daemon to advertise arbitrary expressions from the configuration file in its ClassAd. Give the comma-separated list of entries from the configuration file you want in the given daemon's ClassAd. Frequently used to add attributes to machines so that the machines can discriminate between other machines in a job's **rank** and **requirements**.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<SUBSYS>_EXPRS is a historic setting that functions identically to <SUBSYS>_ATTRS. Use <SUBSYS>_ATTRS.

NOTE: The *condor_kbdd* does not send ClassAds now, so this entry does not affect it. The *condor_startd*, *condor_schedd*, *condor_master*, and *condor_collector* do send ClassAds, so those would be valid subsystems to set this entry for.

SUBMIT_EXPRS not part of the <SUBSYS>_EXPRS, it is documented in section 3.3.14

Because of the different syntax of the configuration file and ClassAds, a little extra work is required to get a given entry into a ClassAd. In particular, ClassAds require quote marks (") around strings. Numeric values and boolean expressions can go in directly. For example, if the *condor_startd* is to advertise a string macro, a numeric macro, and a boolean expression, do something similar to:

```
STRING = This is a string
NUMBER = 666
BOOL1 = True
BOOL2 = CurrentTime >= $(NUMBER) || $(BOOL1)
MY_STRING = "$(STRING)"
```

```
STARTD_ATTRS = MY_STRING, NUMBER, BOOL1, BOOL2
```

DAEMON_SHUTDOWN Starting with Condor version 6.9.3, whenever a daemon is about to publish a ClassAd update to the *condor_collector*, it will evaluate this expression. If it evaluates to True, the daemon will gracefully shut itself down, exit with the exit code 99, and will not be restarted by the *condor_master* (as if it sent itself a *condor_off* command). The expression is evaluated in the context of the ClassAd that is being sent to the *condor_collector*, so it can reference any attributes that can be seen with `condor_status -long [-daemon_type]` (for example, `condor_status -long [-master]` for the *condor_master*). Since each daemon's ClassAd will contain different attributes, administrators should define these shutdown expressions specific to each daemon, for example:

```
STARTD.DAEMON_SHUTDOWN = when to shutdown the startd
MASTER.DAEMON_SHUTDOWN = when to shutdown the master
```

Normally, these expressions would not be necessary, so if not defined, they default to FALSE. One possible use case is for Condor glide-in, to have the *condor_startd* shut itself down if it has not been claimed by a job after a certain period of time.

NOTE: This functionality does not work in conjunction with Condor's high-availability support (see section 3.10 on page 354 for more information). If you enable high-availability for a particular daemon, you should not define this expression.

DAEMON_SHUTDOWNFAST Identical to **DAEMON_SHUTDOWN** (defined above), except the daemon will use the fast shutdown mode (as if it sent itself a *condor_off* command using the **-fast** option).

USE_CLONE_TO_CREATE_PROCESSES This setting controls how a Condor daemon creates a new process under certain versions of Linux. If set to True (the default value), the `clone` system call is used. Otherwise, the `fork` system call is used. `clone` provides scalability improvements for daemons using a large amount of memory (e.g. a *condor_schedd* with a lot of jobs in the queue). Currently, the use of `clone` is available on Linux systems other than IA-64, but not when GCB is enabled.

NOT_RESPONDING_TIMEOUT When a Condor daemon's parent process is another Condor daemon, the child daemon will periodically send a short message to its parent stating that it is alive and well. If the parent does not hear from the child for a while, the parent assumes that the child is hung, kills the child, and restarts the child. This parameter controls how long the parent waits before killing the child. It is defined in terms of seconds and defaults to 3600 (1 hour). The child sends its alive and well messages at an interval of one third of this value.

<SUBSYS>NOT_RESPONDING_TIMEOUT Identical to **NOT_RESPONDING_TIMEOUT**, but controls the timeout for a specific type of daemon. For example, **SCHEDD_NOT_RESPONDING_TIMEOUT** controls how long the *condor_schedd*'s parent daemon will wait without receiving an alive and well message from the *condor_schedd* before killing it.

NOT_RESPONDING_WANT_CORE A boolean parameter with a default value of false. This parameter is for debugging purposes on UNIX systems, and controls the behavior of the parent process when it determines that a child process is not responding (see 164). If NOT_RESPONDING_WANT_CORE is true, the parent will send a SIGABRT instead of SIGKILL to the child process. If the child process is configured with CREATE_CORE_FILES enabled, the child process will then generate a core dump. See NOT_RESPONDING_TIMEOUT on page 164, and CREATE_CORE_FILES on page 156 for related details.

LOCK_FILE_UPDATE_INTERVAL An integer value representing seconds, controlling how often valid lock files should have their on disk timestamps updated. Updating the timestamps prevents administrative programs, such as *tmpwatch*, from deleting long lived lock files. If set to a value less than 60, the update time will be 60 seconds. The default value is 28800, which is 8 hours. This variable only takes effect at the start or restart of a daemon.

3.3.6 Network-Related Configuration File Entries

More information about networking in Condor can be found in section 3.7 on page 325.

BIND_ALL_INTERFACES For systems with multiple network interfaces, if this configuration setting is False, Condor will only bind network sockets to the IP address specified with NETWORK_INTERFACE (described below). If set to True, the default value, Condor will listen on all interfaces. However, currently Condor is still only able to advertise a single IP address, even if it is listening on multiple interfaces. By default, it will advertise the IP address of the network interface used to contact the collector, since this is the most likely to be accessible to other processes which query information from the same collector. More information about using this setting can be found in section 3.7.2 on page 329.

NETWORK_INTERFACE For systems with multiple network interfaces, if this configuration setting is not defined, Condor binds all network sockets to first interface found. To bind to a specific network interface other than the first one, this NETWORK_INTERFACE should be set to the IP address to use. When BIND_ALL_INTERFACES is set to True (the default), this setting simply controls what IP address a given Condor host will advertise. More information about configuring Condor on machines with multiple network interfaces can be found in section 3.7.2 on page 329.

PRIVATE_NETWORK_NAME If two Condor daemons are trying to communicate with each other, and they both belong to the same private network, this setting will allow them to communicate directly using the private network interface, instead of having to use the Generic Connection Broker (GCB) or to go through a public IP address. Each private network should be assigned a unique network name. This string can have any form, but it must be unique for a particular private network. If another Condor daemon or tool is configured with the same PRIVATE_NETWORK_NAME, it will attempt to contact this daemon using the PrivateIpAddress attribute from the classified ad. Even for sites using GCB, this is an important optimization, since it means that two daemons on the same network can

communicate directly, without having to go through the GCB broker. If GCB is enabled, and the `PRIVATE_NETWORK_NAME` is defined, the `PrivateIpAddr` will be defined automatically. Otherwise, you can specify a particular private IP address to use by defining the `PRIVATE_NETWORK_INTERFACE` setting (described below). There is no default for this setting.

PRIVATE_NETWORK_INTERFACE For systems with multiple network interfaces, if this configuration setting and `PRIVATE_NETWORK_NAME` are both defined, Condor daemons will advertise some additional attributes in their ClassAds to help other Condor daemons and tools in the same private network to communicate directly. The `PRIVATE_NETWORK_INTERFACE` defines what IP address a given multi-homed machine should use for the private network. If another Condor daemon or tool is configured with the same `PRIVATE_NETWORK_NAME`, it will attempt to contact this daemon using the IP address specified here. Sites using the Generic Connection Broker (GCB) only need to define the `PRIVATE_NETWORK_NAME`, and the `PRIVATE_NETWORK_INTERFACE` will be defined automatically. Unless GCB is enabled, there is no default for this setting.

HIGHPORT Specifies an upper limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both `HIGHPORT` and `LOWPORT` (given below) must be defined.

LOWPORT Specifies a lower limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both `HIGHPORT` (given above) and `LOWPORT` must be defined.

IN_LOWPORT An integer value that specifies a lower limit of given port numbers for Condor to use on incoming connections (ports for listening), such that Condor is restricted to a range of port numbers. This range implies the use of both `IN_LOWPORT` and `IN_HIGHPORT`. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify `IN_LOWPORT` in combination with `IN_HIGHPORT` such that the range crosses the port 1024 boundary. Applies only to Unix machine configuration. Use of `IN_LOWPORT` and `IN_HIGHPORT` overrides any definition of `LOWPORT` and `HIGHPORT`.

IN_HIGHPORT An integer value that specifies an upper limit of given port numbers for Condor to use on incoming connections (ports for listening), such that Condor is restricted to a range of port numbers. This range implies the use of both `IN_LOWPORT` and `IN_HIGHPORT`. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify `IN_LOWPORT` in combination with `IN_HIGHPORT` such that the range crosses the port 1024 boundary. Applies only to Unix machine configuration. Use of `IN_LOWPORT` and `IN_HIGHPORT` overrides any definition of `LOWPORT` and `HIGHPORT`.

OUT_LOWPORT An integer value that specifies a lower limit of given port numbers for Condor to use on outgoing connections, such that Condor is restricted to a range of port numbers. This range implies the use of both `OUT_LOWPORT` and `OUT_HIGHPORT`. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of `OUT_LOWPORT` and `OUT_HIGHPORT` overrides any definition of `LOWPORT` and `HIGHPORT`.

OUT_HIGHPORT An integer value that specifies an upper limit of given port numbers for Condor to use on outgoing connections, such that Condor is restricted to a range of port numbers. This range implies the use of both `OUT_LOWPORT` and `OUT_HIGHPORT`. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of `OUT_LOWPORT` and `OUT_HIGHPORT` overrides any definition of `LOWPORT` and `HIGHPORT`.

UPDATE_COLLECTOR_WITH_TCP If your site needs to use TCP connections to send ClassAd updates to your collector (which it almost certainly does NOT), set to `True` to enable this feature. Please read section 3.7.4 on “Using TCP to Send Collector Updates” on page 345 for more details and a discussion of when this functionality is needed. At this time, this setting only affects the main *condor_collector* for the site, not any sites that a *condor_schedd* might flock to. If enabled, also define `COLLECTOR_SOCKET_CACHE_SIZE` at the central manager, so that the collector will accept TCP connections for updates, and will keep them open for reuse. Defaults to `False`.

TCP_UPDATE_COLLECTORS The list of collectors which will be updated with TCP instead of UDP. Please read section 3.7.4 on “Using TCP to Send Collector Updates” on page 345 for more details and a discussion of when a site needs this functionality. If not defined, no collectors use TCP instead of UDP.

<SUBSYS>TIMEOUT_MULTIPLIER An integer value that defaults to 1. This value multiplies configured timeout values for all targeted subsystem communications, thereby increasing the time until a timeout occurs. This configuration variable is intended for use by developers for debugging purposes, where communication timeouts interfere.

NONBLOCKING_COLLECTOR_UPDATE A boolean value that defaults to `True`. When `True`, the establishment of TCP connections to the *condor_collector* daemon for a security-enabled pool are done in a nonblocking manner.

NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT A boolean value that defaults to `True`. When `True`, the establishment of TCP connections from the *condor_negotiator* daemon to the *condor_startd* daemon for a security-enabled pool are done in a nonblocking manner.

The following settings are specific to enabling Generic Connection Brokering or GCB in your Condor pool. More information about GCB and how to configure it can be found in section 3.7.3 on page 332.

NET_REMAP_ENABLE A boolean variable, that when defined to `True`, enables a network remapping service for Condor. The service to use is controlled by `NET_REMAP_SERVICE`. This boolean value defaults to `False`.

NET_REMAP_SERVICE If `NET_REMAP_ENABLE` is defined to `True`, this setting controls what network remapping service should be used. Currently, the only value supported is GCB. The default is undefined.

NET_REMAP_INAGENT A comma or space-separated list of IP addresses for GCB brokers. Upon start up, the *condor_master* chooses one at random from among the working brokers in the list. There is no default if not defined.

NET_REMAP_ROUTE Hosts with the GCB network remapping service enabled that would like to use a GCB routing table GCB broker specify the full path to their routing table with this setting. There is no default value if undefined.

MASTER_WAITS_FOR_GCB_BROKER A boolean value that defaults to `True`. This variable determines the behavior of the *condor_master* with GCB enabled. With no GCB broker working upon either the start up of the *condor_master*, or once the *condor_master* has successfully communicated with a GCB broker, but the communication fails, if `MASTER_WAITS_FOR_GCB_BROKER` is `True`, the *condor_master* waits while attempting to find a working GCB broker. With no GCB broker working upon the start up of the *condor_master*, if `MASTER_WAITS_FOR_GCB_BROKER` is `False`, the *condor_master* fails and exits, without restarting. Once the *condor_master* has successfully communicated with a GCB broker, but the communication fails, if `MASTER_WAITS_FOR_GCB_BROKER` is `False`, the *condor_master* kills all its children, exits, and restarts.

The set up task of *condor_glidein* explicitly sets `MASTER_WAITS_FOR_GCB_BROKER` to `False` in the configuration file it produces.

3.3.7 Shared File System Configuration File Macros

These macros control how Condor interacts with various shared and network file systems. If you are using AFS as your shared file system, be sure to read section 3.12.1 on Using Condor with AFS. For information on submitting jobs under shared file systems, see section 2.5.3.

UID_DOMAIN The `UID_DOMAIN` macro is used to decide under which user to run jobs. If the `$(UID_DOMAIN)` on the submitting machine is different than the `$(UID_DOMAIN)` on the machine that runs a job, then Condor runs the job as the user *nobody*. For example, if the submit machine has a `$(UID_DOMAIN)` of `flippy.cs.wisc.edu`, and the machine where the job will execute has a `$(UID_DOMAIN)` of `cs.wisc.edu`, the job will run as user *nobody*, because the two `$(UID_DOMAIN)`s are not the same. If the `$(UID_DOMAIN)` is the same on both the submit and execute machines, then Condor will run the job as the user that submitted the job.

A further check attempts to assure that the submitting machine can not lie about its `UID_DOMAIN`. Condor compares the submit machine's claimed value for `UID_DOMAIN` to its fully qualified name. If the two do not end the same, then the submit machine is presumed to be lying about its `UID_DOMAIN`. In this case, Condor will run the job as user *nobody*. For example, a job submission to the Condor pool at the UW Madison from `flippy.example.com`, claiming a `UID_DOMAIN` of `cs.wisc.edu`, will run the job as the user *nobody*.

Because of this verification, `$(UID_DOMAIN)` must be a real domain name. At the Computer Sciences department at the UW Madison, we set the `$(UID_DOMAIN)` to be `cs.wisc.edu` to indicate that whenever someone submits from a department machine, we will run the job as the user who submits it.

Also see `SOFT_UID_DOMAIN` below for information about one more check that Condor performs before running a job as a given user.

A few details:

An administrator could set `UID_DOMAIN` to `*`. This will match all domains, but it is a gaping security hole. It is not recommended.

An administrator can also leave `UID_DOMAIN` undefined. This will force Condor to always run jobs as user `nobody`. Running standard universe jobs as user `nobody` enhances security and should cause no problems, because the jobs use remote I/O to access all of their files. However, if vanilla jobs are run as user `nobody`, then files that need to be accessed by the job will need to be marked as world readable/writable so the user `nobody` can access them.

When Condor sends e-mail about a job, Condor sends the e-mail to `user@$ (UID_DOMAIN)`. If `UID_DOMAIN` is undefined, the e-mail is sent to `user@submitmachinename`.

TRUST_UID_DOMAIN As an added security precaution when Condor is about to spawn a job, it ensures that the `UID_DOMAIN` of a given submit machine is a substring of that machine's fully-qualified host name. However, at some sites, there may be multiple UID spaces that do not clearly correspond to Internet domain names. In these cases, administrators may wish to use names to describe the UID domains which are not substrings of the host names of the machines. For this to work, Condor must not do this regular security check. If the `TRUST_UID_DOMAIN` setting is defined to `True`, Condor will not perform this test, and will trust whatever `UID_DOMAIN` is presented by the submit machine when trying to spawn a job, instead of making sure the submit machine's host name matches the `UID_DOMAIN`. When not defined, the default is `False`, since it is more secure to perform this test.

SOFT_UID_DOMAIN A boolean variable that defaults to `False` when not defined. When Condor is about to run a job as a particular user (instead of as user `nobody`), it verifies that the UID given for the user is in the password file and actually matches the given user name. However, under installations that do not have every user in every machine's password file, this check will fail and the execution attempt will be aborted. To cause Condor not to do this check, set this configuration variable to `True`. Condor will then run the job under the user's UID.

SLOTx_USER The name of a user for Condor to use instead of user `nobody`, as part of a solution that plugs a security hole whereby a lurker process can prey on a subsequent job run as user name `nobody`. `x` is an integer associated with slots. On Windows, `SLOTx_USER` will only work if the credential of the specified user is stored on the execute machine using `condor_store_cred`. See Section 3.6.11 for more information.

STARTER_ALLOW_RUNAS_OWNER This is a boolean expression (evaluated with the job ad as the target) that determines whether the job may run under the job owner's account (`true`) or whether it will run as `SLOTx_USER` or `nobody` (`false`). In Unix, this defaults to `true`. In windows, it defaults to `false`. The job ClassAd may also contain an attribute `RunAsOwner` which is logically ANDed with the starter's boolean value. Under Unix, if the job does not specify it, this attribute defaults to `true`. Under windows, it defaults to `false`. In Unix, if the `UidDomain` of the machine and job do not match, then there is no possibility to run the job as the owner anyway, so, in that case, this setting has no effect. See Section 3.6.11 for more information.

DEDICATEDEXECUTEACCOUNT_REGEX This is a regular expression (i.e. a string matching pattern) that matches the account name(s) that are dedicated to running condor jobs on the

execute machine and which will never be used for more than one job at a time. The default matches no account name. If you have configured `SLOTx_USER` to be a *different* account for each Condor slot, and no non-condor processes will ever be run by these accounts, then this pattern should match the names of all `SLOTx_USER` accounts. Jobs run under a dedicated execute account are reliably tracked by Condor, whereas other jobs, may spawn processes that Condor fails to detect. Therefore, a dedicated execution account provides more reliable tracking of CPU usage by the job and it also guarantees that when the job exits, no “lurker” processes are left behind. When the job exits, condor will attempt to kill all processes owned by the dedicated execution account. Example:

```
SLOT1_USER = cndrusr1
SLOT2_USER = cndrusr2
STARTER_ALLOW_RUNAS_OWNER = False
DEDICATED_EXECUTE_ACCOUNT_REGEX = cndrusr[0-9]+
```

You can tell if the starter is in fact treating the account as a dedicated account, because it will print a line such as the following in its log file:

```
Tracking process family by login "cndrusr1"
```

EXECUTELOGIN_IS_DEDICATED This configuration setting is deprecated because it cannot handle the case where some jobs run as dedicated accounts and some do not. Use `DEDICATED_EXECUTE_ACCOUNT_REGEX` instead.

A boolean value that defaults to `False`. When `True`, Condor knows that all jobs are being run by dedicated execution accounts (whether they are running as the job owner or as nobody or as `SLOTx_USER`). Therefore, when the job exits, all processes running under the same account will be killed.

FILESYSTEMDOMAIN The `FILESYSTEMDOMAIN` macro is an arbitrary string that is used to decide if two machines (a submitting machine and an execute machine) share a file system. Although the macro name contains the word “DOMAIN”, the macro is not required to be a domain name. It often is a domain name.

Note that this implementation is not ideal: machines may share some file systems but not others. Condor currently has no way to express this automatically. You can express the need to use a particular file system by adding additional attributes to your machines and submit files, similar to the example given in Frequently Asked Questions, section 7 on how to run jobs only on machines that have certain software packages.

Note that if you do not set `$(FILESYSTEMDOMAIN)`, Condor defaults to setting the macro’s value to be the fully qualified host name of the local machine. Since each machine will have a different `$(FILESYSTEMDOMAIN)`, they will not be considered to have shared file systems.

RESERVEAFSCACHE If your machine is running AFS and the AFS cache lives on the same partition as the other Condor directories, and you want Condor to reserve the space that your AFS cache is configured to use, set this macro to `True`. It defaults to `False`.

USE_NFS This macro influences how Condor jobs running in the standard universe access their files. Condor will redirect the file I/O requests of standard universe jobs to be executed on the machine which submitted the job. Because of this, as a Condor job migrates around the network, the file system always appears to be identical to the file system where the job was submitted. However, consider the case where a user's data files are sitting on an NFS server. The machine running the user's program will send all I/O over the network to the machine which submitted the job, which in turn sends all the I/O over the network a second time back to the NFS file server. Thus, all of the program's I/O is being sent over the network twice.

If this macro to `True`, then Condor will attempt to read/write files without redirecting I/O back to the submitting machine if both the submitting machine and the machine running the job are both accessing the same NFS servers (*if* they are both in the same `$(FILESYSTEM_DOMAIN)` and in the same `$(UID_DOMAIN)`, as described above). The result is I/O performed by Condor standard universe jobs is only sent over the network once. While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth is at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_NFS)` to `False` is always safe. It may result in slightly more network traffic, but Condor jobs are most often heavy on CPU and light on I/O. It also ensures that a remote standard universe Condor job will always use Condor's remote system calls mechanism to reroute I/O and therefore see the exact same file system that the user sees on the machine where she/he submitted the job.

Some gritty details for folks who want to know: If the you set `$(USE_NFS)` to `True`, and the `$(FILESYSTEM_DOMAIN)` of both the submitting machine and the remote machine about to execute the job match, and the `$(FILESYSTEM_DOMAIN)` claimed by the submit machine is indeed found to be a subset of what an inverse look up to a DNS (domain name server) reports as the fully qualified domain name for the submit machine's IP address (this security measure safeguards against the submit machine from lying), *then* the job will access files using a local system call, without redirecting them to the submitting machine (with NFS). Otherwise, the system call will get routed back to the submitting machine using Condor's remote system call mechanism. **NOTE:** When submitting a vanilla job, `condor_submit` will, by default, append requirements to the Job ClassAd that specify the machine to run the job must be in the same `$(FILESYSTEM_DOMAIN)` and the same `$(UID_DOMAIN)`.

IGNORE_NFS_LOCK_ERRORS When set to `True`, all errors related to file locking errors from NFS are ignored. Defaults to `False`, not ignoring errors.

USE_AFS If your machines have AFS, this macro determines whether Condor will use remote system calls for standard universe jobs to send I/O requests to the submit machine, or if it should use local file access on the execute machine (which will then use AFS to get to the submitter's files). Read the setting above on `$(USE_NFS)` for a discussion of why you might want to use AFS access instead of remote system calls.

One important difference between `$(USE_NFS)` and `$(USE_AFS)` is the AFS cache. With `$(USE_AFS)` set to `True`, the remote Condor job executing on some machine will start modifying the AFS cache, possibly evicting the machine owner's files from the cache to make

room for its own. Generally speaking, since we try to minimize the impact of having a Condor job run on a given machine, we do not recommend using this setting.

While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth is at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_AFS)` to `False` is always safe. It may result in slightly more network traffic, but Condor jobs are usually heavy on CPU and light on I/O. `False` ensures that a remote standard universe Condor job will always see the exact same file system that the user sees on the machine where he/she submitted the job. Plus, it will ensure that the machine where the job executes does not have its AFS cache modified as a result of the Condor job being there.

However, things may be different at your site, which is why the setting is there.

3.3.8 Checkpoint Server Configuration File Macros

These macros control whether or not Condor uses a checkpoint server. If you are using a checkpoint server, this section describes the settings that the checkpoint server itself needs defined. A checkpoint server is installed separately. It is not included in the main Condor binary distribution or installation procedure. See section 3.8 on Installing a Checkpoint Server for details on installing and running a checkpoint server for your pool.

NOTE: If you are setting up a machine to join the UW-Madison CS Department Condor pool, you *should* configure the machine to use a checkpoint server, and use “condor-ckpt.cs.wisc.edu” as the checkpoint server host (see below).

CKPT_SERVER_HOST The host name of a checkpoint server.

STARTER_CHOOSES_CKPT_SERVER If this parameter is `True` or undefined on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the execute machine is used. If it is `False` on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the submit machine is used.

CKPT_SERVER_DIR The checkpoint server needs this macro defined to the full path of the directory the server should use to store checkpoint files. Depending on the size of your pool and the size of the jobs your users are submitting, this directory (and its subdirectories) might need to store many Mbytes of data.

USE_CKPT_SERVER A boolean which determines if you want a given submit machine to use a checkpoint server if one is available. If a checkpoint server isn’t available or `USE_CKPT_SERVER` is set to `False`, checkpoints will be written to the local `$(SPOOL)` directory on the submission machine.

MAX_DISCARDED_RUN_TIME If the shadow is unable to read a checkpoint file from the checkpoint server, it keeps trying only if the job has accumulated more than this many seconds of CPU usage. Otherwise, the job is started from scratch. Defaults to 3600 (1 hour). This setting is only used if `$(USE_CKPT_SERVER)` is `True`.

CKPT_SERVER_CHECK_PARENT_INTERVAL This is the number of seconds between checks to see whether the parent of the checkpoint server (i.e. the *condor_master*) has died. If the parent has died, the checkpoint server shuts itself down. The default is 120 seconds. A setting of 0 disables this check.

3.3.9 condor_master Configuration File Macros

These macros control the *condor_master*.

DAEMONLIST This macro determines what daemons the *condor_master* will start and keep its watchful eyes on. The list is a comma or space separated list of subsystem names (listed in section 3.3.1). For example,

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

NOTE: This configuration variable cannot be changed by using *condor_reconfig* or by sending a SIGHUP. To change this configuration variable, restart the *condor_master* daemon by using *condor_restart*. Only then will the change take effect.

NOTE: On your central manager, your \$(DAEMON_LIST) will be different from your regular pool, since it will include entries for the *condor_collector* and *condor_negotiator*.

DC_DAEMONLIST This macro lists the daemons in DAEMON_LIST which use the Condor DaemonCore library. The *condor_master* must differentiate between daemons that use DaemonCore and those that don't so it uses the appropriate inter-process communication mechanisms. This list currently includes all Condor daemons except the checkpoint server by default.

As of 7.2.1, you can append to the default DC_DAEMON_LIST list by specifying a "+" as the first character in the DC_DAEMON_LIST definition. For example:

```
DAEMON_LIST = NEW_DAEMON
DC_DAEMON_LIST = +NEW_DAEMON
```

<SUBSYS> Once you have defined which subsystems you want the *condor_master* to start, you must provide it with the full path to each of these binaries. For example:

```
MASTER          = $(SBIN)/condor_master
STARTD          = $(SBIN)/condor_startd
SCHEDD          = $(SBIN)/condor_schedd
```

These are most often defined relative to the \$(SBIN) macro.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

DAEMONNAME_ENVIRONMENT For each subsystem defined in `DAEMON_LIST`, you may specify changes to the environment that daemon is started with by setting `DAEMONNAME_ENVIRONMENT`, where `DAEMONNAME` is the name of a daemon listed in `DAEMON_LIST`. It should use the same syntax for specifying the environment as the environment specification in a *condor_submit* file (see page 797). For example, if you wish to redefine the `TMP` and `CONDOR_CONFIG` environment variables seen by the *condor_schedd*, you could place the following in the config file:

```
SCHEDD_ENVIRONMENT = "TMP=/new/value CONDOR_CONFIG=/special/config"
```

When the *condor_schedd* was started by the *condor_master*, it would see the specified values of `TMP` and `CONDOR_CONFIG`.

<SUBSYS>ARGS This macro allows the specification of additional command line arguments for any process spawned by the *condor_master*. List the desired arguments using the same syntax as the arguments specification in a *condor_submit* submit file (see page 796), with one exception: do not escape double-quotes when using the old-style syntax (this is for backward compatibility). Set the arguments for a specific daemon with this macro, and the macro will affect only that daemon. Define one of these for each daemon the *condor_master* is controlling. For example, set `$(STARTD_ARGS)` to specify any extra command line arguments to the *condor_startd*.

The macro is named by substituting `<SUBSYS>` with the appropriate subsystem string as defined in section 3.3.1.

PREEN In addition to the daemons defined in `$(DAEMON_LIST)`, the *condor_master* also starts up a special process, *condor_preen* to clean out junk files that have been left laying around by Condor. This macro determines where the *condor_master* finds the *condor_preen* binary. Comment out this macro, and *condor_preen* will not run.

PREEN_ARGS Controls how *condor_preen* behaves by allowing the specification of command-line arguments. This macro works as `$(<SUBSYS>_ARGS)` does. The difference is that you must specify this macro for *condor_preen* if you want it to do anything. *condor_preen* takes action only because of command line arguments. **-m** means you want e-mail about files *condor_preen* finds that it thinks it should remove. **-r** means you want *condor_preen* to actually remove these files.

PREEN_INTERVAL This macro determines how often *condor_preen* should be started. It is defined in terms of seconds and defaults to 86400 (once a day).

PUBLISH_OBITUARIES When a daemon crashes, the *condor_master* can send e-mail to the address specified by `$(CONDOR_ADMIN)` with an obituary letting the administrator know that the daemon died, the cause of death (which signal or exit status it exited with), and (optionally) the last few entries from that daemon's log file. If you want obituaries, set this macro to `True`.

OBITUARY_LOG_LENGTH This macro controls how many lines of the log file are part of obituaries. This macro has a default value of 20 lines.

START_MASTER If this setting is defined and set to `False` when the *condor_master* starts up, the first thing it will do is exit. This appears strange, but perhaps you do not want Condor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized This is an entry you would most likely find in a local configuration file, not a global configuration file.

START_DAEMONS This macro is similar to the `$(START_MASTER)` macro described above. However, the *condor_master* does not exit; it does not start any of the daemons listed in the `$(DAEMON_LIST)`. The daemons may be started at a later time with a *condor_on* command.

MASTER_UPDATE_INTERVAL This macro determines how often the *condor_master* sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_CHECK_NEW_EXEC_INTERVAL This macro controls how often the *condor_master* checks the timestamps of the running daemons. If any daemons have been modified, the master restarts them. It is defined in seconds and defaults to 300 (every 5 minutes).

MASTER_NEW_BINARY_DELAY Once the *condor_master* has discovered a new binary, this macro controls how long it waits before attempting to execute the new binary. This delay exists because the *condor_master* might notice a new binary while it is in the process of being copied, in which case trying to execute it yields unpredictable results. The entry is defined in seconds and defaults to 120 (2 minutes).

SHUTDOWN_FAST_TIMEOUT This macro determines the maximum amount of time daemons are given to perform their fast shutdown procedure before the *condor_master* kills them outright. It is defined in seconds and defaults to 300 (5 minutes).

MASTER_SHUTDOWN_<Name> A full path and file name of a program that the *condor_master* is to execute via the Unix `exec1()` call, or the similar Win32 `_exec1()` call, instead of the normal call to `exit()`. Multiple programs to execute may be defined with multiple entries, each with a unique Name. These macros have no affect on a *condor_master* unless *condor_set_shutdown* is run. The Name specified as an argument to the *condor_set_shutdown* program must match the Name portion of one of these `MASTER_SHUTDOWN_<Name>` macros; if not, the *condor_master* will log an error and ignore the command. If a match is found, the *condor_master* will attempt to verify the program, and it will store the path and program name. When the *condor_master* shuts down, the program is then executed as described above. The manual page for *condor_set_shutdown* on page 781 contains details on the use of this program.

MASTER_BACKOFF_CONSTANT and MASTER_<name>_BACKOFF_CONSTANT When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. These settings define the constant value of the expression used to determine how long to wait before starting the daemon again (and, effectively becomes the initial backoff time). It is an integer in units of seconds, and defaults to 9 seconds.

`$(MASTER_<name>_BACKOFF_CONSTANT)` is the daemon-specific form of `MASTER_BACKOFF_CONSTANT`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will be used.

MASTER_BACKOFF_FACTOR and MASTER<name>_BACKOFF_FACTOR When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This setting is the base of the exponent used to determine how long to wait before starting the daemon again. It defaults to 2 seconds.

`$(MASTER_<name>_BACKOFF_FACTOR)` is the daemon-specific form of `MASTER_BACKOFF_FACTOR`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will be used.

MASTER_BACKOFF_CEILING and MASTER<name>_BACKOFF_CEILING When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This entry determines the maximum amount of time you want the master to wait between attempts to start a given daemon. (With 2.0 as the `$(MASTER_BACKOFF_FACTOR)`, 1 hour is obtained in 12 restarts). It is defined in terms of seconds and defaults to 3600 (1 hour).

`$(MASTER_<name>_BACKOFF_CEILING)` is the daemon-specific form of `MASTER_BACKOFF_CEILING`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will be used.

MASTER_RECOVER_FACTOR and MASTER<name>_RECOVER_FACTOR A macro to set how long a daemon needs to run without crashing before it is considered *recovered*. Once a daemon has recovered, the number of restarts is reset, so the exponential back off returns to its initial state. The macro is defined in terms of seconds and defaults to 300 (5 minutes).

`$(MASTER_<name>_RECOVER_FACTOR)` is the daemon-specific form of `MASTER_RECOVER_FACTOR`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will be used.

When a daemon crashes, *condor_master* will restart the daemon after a delay (a back off). The length of this delay is based on how many times it has been restarted, and gets larger after each crash. The equation for calculating this backoff time is given by:

$$t = c + k^n$$

where t is the calculated time, c is the constant defined by `$(MASTER_BACKOFF_CONSTANT)`, k is the “factor” defined by `$(MASTER_BACKOFF_FACTOR)`, and n is the number of restarts already attempted (0 for the first restart, 1 for the next, etc.).

With default values, after the first crash, the delay would be $t = 9 + 2.0^0$, giving 10 seconds (remember, $n = 0$). If the daemon keeps crashing, the delay increases.

For example, take the `$(MASTER_BACKOFF_FACTOR)` (which defaults to 2.0) to the power the number of times the daemon has restarted, and add `$(MASTER_BACKOFF_CONSTANT)` (which defaults to 9). Thus:

1st crash: $n = 0$, so: $t = 9 + 2^0 = 9 + 1 = 10$ seconds

2nd crash: $n = 1$, so: $t = 9 + 2^1 = 9 + 2 = 11$ seconds

3rd crash: $n = 2$, so: $t = 9 + 2^2 = 9 + 4 = 13$ seconds

...

6th crash: $n = 5$, so: $t = 9 + 2^5 = 9 + 32 = 41$ seconds

...

9th crash: $n = 8$, so: $t = 9 + 2^8 = 9 + 256 = 265$ seconds

And, after the 13 crashes, it would be:

13th crash: $n = 12$, so: $t = 9 + 2^{12} = 9 + 4096 = 4105$ seconds

This is bigger than the $\$(MASTER_BACKOFF_CEILING)$, which defaults to 3600, so the daemon would really be restarted after only 3600 seconds, not 4105. The *condor_master* tries again every hour (since the numbers would get larger and would always be capped by the ceiling). Eventually, imagine that daemon finally started and did not crash. This might happen if, for example, an administrator reinstalled an accidentally deleted binary after receiving e-mail about the daemon crashing. If it stayed alive for $\$(MASTER_RECOVER_FACTOR)$ seconds (defaults to 5 minutes), the count of how many restarts this daemon has performed is reset to 0.

The moral of the example is that the defaults work quite well, and you probably will not want to change them for any reason.

MASTER_NAME Defines a unique name given for a *condor_master* daemon on a machine. For a *condor_master* running as *root*, it defaults to the fully qualified host name. When *not* running as *root*, it defaults to the user that instantiates the *condor_master*, concatenated with an at symbol (@), concatenated with the fully qualified host name. If more than one *condor_master* is running on the same host, then the *MASTER_NAME* for each *condor_master* must be defined to uniquely identify the separate daemons.

A defined *MASTER_NAME* is presumed to be of the form *identifying-string@full.host.name*. If the string does not include an @ sign, Condor appends one, followed by the fully qualified host name of the local machine. The *identifying-string* portion may contain any alphanumeric ASCII characters or punctuation marks, except the @ sign. We recommend that the string does not contain the : (colon) character, since that might cause problems with certain tools. Previous to Condor 7.1.1, when the string included an @ sign, Condor replaced whatever followed the @ sign with the fully qualified host name of the local machine. Condor does not modify any portion of the string, if it contains an @ sign. This is useful for remote job submissions under the high availability of the job queue.

If the *MASTER_NAME* setting is used, and the *condor_master* is configured to spawn a *condor_schedd*, the name defined with *MASTER_NAME* takes precedence over the *SCHEDD_NAME* setting (see section 3.3.11 on page 199). Since Condor makes the assumption that there is only one instance of the *condor_startd* running on a machine, the *MASTER_NAME* is not automatically propagated to the *condor_startd*. However, in situations where multiple *condor_startd* daemons are running on the same host (for example, when using *condor_glidein*), the *STARTD_NAME* should be set to uniquely identify the *condor_startd* daemons (this is done

automatically in the case of *condor_glidein*).

If a Condor daemon (master, schedd or startd) has been given a unique name, all Condor tools that need to contact that daemon can be told what name to use via the **-name** command-line option.

MASTER_ATTRS This macro is described in section 3.3.5 as `<SUBSYS>_ATTRS`.

MASTER_DEBUG This macro is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

MASTER_ADDRESS_FILE This macro is described in section 3.3.5 as `<SUBSYS>_ADDRESS_FILE`.

SECONDARY_COLLECTOR_LIST This macro has been removed as of Condor version 6.9.3. Use the `COLLECTOR_HOST` configuration variable, which may define a list of *condor_collector* daemons.

ALLOW_ADMIN_COMMANDS If set to NO for a given host, this macro disables administrative commands, such as *condor_restart*, *condor_on*, and *condor_off*, to that host.

MASTER_INSTANCELOCK Defines the name of a file for the *condor_master* daemon to lock in order to prevent multiple *condor_masters* from starting. This is useful when using shared file systems like NFS which do not technically support locking in the case where the lock files reside on a local disk. If this macro is not defined, the default file name will be `$(LOCK)/InstanceLock`. `$(LOCK)` can instead be defined to specify the location of all lock files, not just the *condor_master*'s `InstanceLock`. If `$(LOCK)` is undefined, then the master log itself is locked.

ADD_WINDOWS_FIREWALL_EXCEPTION When set to False, the *condor_master* will not automatically add Condor to the Windows Firewall list of trusted applications. Such trusted applications can accept incoming connections without interference from the firewall. This only affects machines running Windows XP SP2 or higher. The default is True.

WINDOWS_FIREWALL_FAILURE_RETRY An integer value (default value is 60) that represents the number of times the *condor_master* will retry to add firewall exceptions. When a Windows machine boots up, Condor starts up by default as well. Under certain conditions, the *condor_master* may have difficulty adding exceptions to the Windows Firewall because of a delay in other services starting up. Examples of services that may possibly be slow are the SharedAccess service, the Netman service, or the Workstation service. This configuration variable allows administrators to set the number of times (once every 10 seconds) that the *condor_master* will retry to add firewall exceptions. A value of 0 means that Condor will retry indefinitely.

USE_PROCESS_GROUPS A boolean value that defaults to True. When False, Condor daemons on Unix machines will *not* create new sessions or process groups. Condor uses process groups to help it track the descendants of processes it creates. This can cause problems when Condor is run under another job execution system (e.g. Condor Glidein).

3.3.10 condor_startd Configuration File Macros

NOTE: If you are running Condor on a multi-CPU machine, be sure to also read section 3.12.7 on page 399 which describes how to set up and configure Condor on SMP machines.

These settings control general operation of the *condor_startd*. Examples using these configuration macros, as well as further explanation is found in section 3.5 on Configuring The Startd Policy.

START A boolean expression that, when `True`, indicates that the machine is willing to start running a Condor job. `START` is considered when the *condor_negotiator* daemon is considering evicting the job to replace it with one that will generate a better rank for the *condor_startd* daemon, or a user with a higher priority.

SUSPEND A boolean expression that, when `True`, causes Condor to suspend running a Condor job. The machine may still be claimed, but the job makes no further progress, and Condor does not generate a load on the machine.

PREEMPT A boolean expression that, when `True`, causes Condor to stop a currently running job.

CONTINUE A boolean expression that, when `True`, causes Condor to continue the execution of a suspended job.

KILL A boolean expression that, when `True`, causes Condor to immediately stop the execution of a currently running job, without delay, and without taking the time to produce a checkpoint (for a standard universe job).

PERIODICCHECKPOINT A boolean expression that, when `True`, causes Condor to initiate a checkpoint of the currently running job. This setting applies to all standard universe jobs and to vm universe jobs that have set **vm_checkpoint** to `True` in the submit description file.

RANK A floating point value that Condor uses to compare potential jobs. A larger value for a specific job ranks that job above others with lower values for `RANK`.

IS_VALID_CHECKPOINT_PLATFORM A boolean expression that is logically ANDed with the `START` expression to limit which machines a standard universe job may continue execution on once they have produced a checkpoint. The default expression is

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  ( (TARGET.JobUniverse == 1) == FALSE) ||
  (
    (MY.CheckpointPlatform != UNDEFINED) &&
    (
      (TARGET.LastCheckpointPlatform == MY.CheckpointPlatform) ||
      (TARGET.NumCkpts == 0)
    )
  )
)
```

WANT_SUSPEND A boolean expression that, when `True`, tells Condor to evaluate the `SUSPEND` expression.

WANT_VACATE A boolean expression that, when `True`, defines that a preempted Condor job is to be vacated, instead of killed.

IS_OWNER A boolean expression that defaults to being defined as

```
IS_OWNER = ( START == FALSE )
```

Used to describe the state of the machine with respect to its use by its owner. Job ClassAd attributes are not used in defining `IS_OWNER`, as they would be `Undefined`.

STARTER This macro holds the full path to the *condor_starter* binary that the *condor_startd* should spawn. It is normally defined relative to `$(SBIN)`.

POLLING_INTERVAL When a *condor_startd* enters the claimed state, this macro determines how often the state of the machine is polled to check the need to suspend, resume, vacate or kill the job. It is defined in terms of seconds and defaults to 5.

UPDATE_INTERVAL Determines how often the *condor_startd* should send a ClassAd update to the *condor_collector*. The *condor_startd* also sends update on any state or activity change, or if the value of its `START` expression changes. See section 3.5.5 on *condor_startd* states, section 3.5.6 on *condor_startd* Activities, and section 3.5.2 on *condor_startd* `START` expression for details on states, activities, and the `START` expression. This macro is defined in terms of seconds and defaults to 300 (5 minutes).

MAXJOBRETIREMENTTIME An integer value representing the number of seconds a preempted job will be allowed to run before being evicted. The default value of 0 (when the configuration variable is not present) implements the expected policy that there is no retirement time. See `MAXJOBRETIREMENTTIME` in section 3.5.8 for further explanation.

CLAIMWORKLIFE If provided, this expression specifies the number of seconds after which a claim will stop accepting additional jobs. By default, once the negotiator gives a schedd a claim to a slot, the schedd will keep running jobs on that slot as long as it has more jobs with matching requirements, without returning the slot to the unclaimed state and renegotiating for machines. Once `CLAIMWORKLIFE` expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. This may be useful if you want to force periodic renegotiation of resources without preemption having to occur. For example, if you have some low-priority jobs which should never be interrupted with kill signals, you could prevent them from being killed with `MaxJobRetirementTime`, but now high-priority jobs may have to wait in line when they match to a machine that is busy running one of these uninterruptible jobs. You can prevent the high-priority jobs from ever matching to such a machine by using a rank expression in the job or in the negotiator's rank expressions, but then the low-priority claim will never be interrupted; it can keep running more jobs. The solution is to use `CLAIMWORKLIFE` to force the claim to stop running additional jobs after a certain amount of time. The default value for `CLAIMWORKLIFE` is -1, which is treated as an infinite

claim worklife, so claims may be held indefinitely (as long as they are not preempted and the schedd does not relinquish them, of course). A value of 0 has the effect of not allowing more than one job to run per claim, since it immediately expires after the first job starts running.

MAX_CLAIM_ALIVES_MISSED The *condor_schedd* sends periodic updates to each *condor_startd* as a keep alive (see the description of `ALIVE_INTERVAL` on page 197). If the *condor_startd* does not receive any keep alive messages, it assumes that something has gone wrong with the *condor_schedd* and that the resource is not being effectively used. Once this happens, the *condor_startd* considers the claim to have timed out, it releases the claim, and starts advertising itself as available for other jobs. Because these keep alive messages are sent via UDP, they are sometimes dropped by the network. Therefore, the *condor_startd* has some tolerance for missed keep alive messages, so that in case a few keep alives are lost, the *condor_startd* will not immediately release the claim. This setting controls how many keep alive messages can be missed before the *condor_startd* considers the claim no longer valid. The default is 6.

STARTD_HAS_BAD_UTMP When the *condor_startd* is computing the idle time of all the users of the machine (both local and remote), it checks the utmp file to find all the currently active ttys, and only checks access time of the devices associated with active logins. Unfortunately, on some systems, utmp is unreliable, and the *condor_startd* might miss keyboard activity by doing this. So, if your utmp is unreliable, set this macro to `True` and the *condor_startd* will check the access time on all tty and pty devices.

CONSOLEDEVICES This macro allows the *condor_startd* to monitor console (keyboard and mouse) activity by checking the access times on special files in `/dev`. Activity on these files shows up as `ConsoleIdle` time in the *condor_startd*'s ClassAd. Give a comma-separated list of the names of devices considered the console, without the `/dev/` portion of the path name. The defaults vary from platform to platform, and are usually correct.

One possible exception to this is on Linux, where we use “mouse” as one of the entries. Most Linux installations put in a soft link from `/dev/mouse` that points to the appropriate device (for example, `/dev/psaux` for a PS/2 bus mouse, or `/dev/tty00` for a serial mouse connected to com1). However, if your installation does not have this soft link, you will either need to put it in (you will be glad you did), or change this macro to point to the right device.

Unfortunately, there are no such devices on Digital Unix (don't be fooled by `/dev/keyboard0`; the kernel does not update the access times on these devices), so this macro is not useful in these cases, and we must use the *condor_kbdd* to get this information by connecting to the X server.

STARTD_JOB_EXPRS When the machine is claimed by a remote user, the *condor_startd* can also advertise arbitrary attributes from the job ClassAd in the machine ClassAd. List the attribute names to be advertised. **NOTE:** Since these are already ClassAd expressions, do not do anything unusual with strings. This setting defaults to “JobUniverse”.

STARTD_ATTRS This macro is described in section 3.3.5 as `<SUBSYS>_ATTRS`.

STARTD_DEBUG This macro (and other settings related to debug logging in the *condor_startd*) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

STARTD_ADDRESS_FILE This macro is described in section 3.3.5 as `<SUBSYS>_ADDRESS_FILE`

STARTD_SENDS_ALIVES A boolean value that defaults to `False`, such that the *condor_schedd* daemon sends keep alive signals to the *condor_startd* daemon. When `True`, the *condor_startd* daemon sends keep alive signals to the *condor_schedd* daemon, reversing the direction. This may be useful if the *condor_startd* daemon is on a private network or behind a firewall.

STARTD_SHOULD_WRITE_CLAIM_ID_FILE The *condor_startd* can be configured to write out the `ClaimId` for the next available claim on all slots to separate files. This boolean attribute controls whether the *condor_startd* should write these files. The default value is `True`.

STARTD_CLAIM_ID_FILE This macro controls what file names are used if the above `STARTD_SHOULD_WRITE_CLAIM_ID_FILE` is true. By default, Condor will write the `ClaimId` into a file in the `$(LOG)` directory called `.startd.claim_id.slotX`, where `X` is the value of `SlotID`, the integer that identifies a given slot on the system, or 1 on a single-slot machine. If you define your own value for this setting, you should provide a full path, and Condor will automatically append the `.slotX` portion of the file name.

NUM_CPUS An integer value, which can be used to lie to the *condor_startd* daemon about how many CPUs a machine has. When set, it overrides the value determined with Condor's automatic computation of the number of CPUs in the machine. Lying in this way can allow multiple Condor jobs to run on a single-CPU machine, by having that machine treated like an SMP machine with multiple CPUs, which could have different Condor jobs running on each one. Or, an SMP machine may advertise more slots than it has CPUs. However, lying in this manner will hurt the performance of the jobs, since now multiple jobs will run on the same CPU, and the jobs will compete with each other. The option is only meant for people who specifically want this behavior and know what they are doing. It is disabled by default.

Note that this setting cannot be changed with a simple reconfigure, either by sending a `SIGHUP` or by using the *condor_reconfig* command. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
condor_restart -startd
```

If lying about a given machine, this fact should probably be advertised in the machine's `ClassAd` by using the `STARTD_ATTRS` setting. This way, jobs submitted in the pool could specify that they did or did not want to be matched with machines that were only really offering these fractional CPUs.

MAX_NUM_CPUS An integer value used as a ceiling for the number of CPUs detected by Condor on a machine. This value is ignored if `NUM_CPUS` is set. If set to zero, there is no ceiling. If not defined, the default value is zero, and thus there is no ceiling.

Note that this setting cannot be changed with a simple reconfigure, either by sending a `SIGHUP` or by using the *condor_reconfig* command. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
condor_restart -startd
```

COUNT_HYPERTHREAD_CPUS This macro controls how Condor sees hyper threaded processors. When set to `True` (the default), it includes virtual CPUs in the default value of `NUM_CPUS`. On dedicated cluster nodes, counting virtual CPUs can sometimes improve total throughput at the expense of individual job speed. However, counting them on desktop workstations can interfere with interactive job performance.

MEMORY Normally, Condor will automatically detect the amount of physical memory available on your machine. Define `MEMORY` to tell Condor how much physical memory (in MB) your machine has, overriding the value Condor computes automatically.

RESERVED_MEMORY How much memory would you like reserved from Condor? By default, Condor considers all the physical memory of your machine as available to be used by Condor jobs. If `RESERVED_MEMORY` is defined, Condor subtracts it from the amount of memory it advertises as available.

STARTD_NAME Used to give an alternative value to the `Name` attribute in the *condor_startd*'s ClassAd. This esoteric configuration macro might be used in the situation where there are two *condor_startd* daemons running on one machine, and each reports to the same *condor_collector*. Different names will distinguish the two daemons. See the description of `MASTER_NAME` in section 3.3.9 on page 177 for defaults and composition of valid Condor daemon names.

RUNBENCHMARKS Specifies when to run benchmarks. When the machine is in the Unclaimed state and this expression evaluates to `True`, benchmarks will be run. If `RunBenchmarks` is specified and set to anything other than `False`, additional benchmarks will be run when the *condor_startd* initially starts. To disable start up benchmarks, set `RunBenchmarks` to `False`, or comment it out of the configuration file.

DedicatedScheduler A string that identifies the dedicated scheduler this machine is managed by. Section 3.12.8 on page 409 details the use of a dedicated scheduler.

STARTD_NOCLAIMSHUTDOWN The number of seconds to run without receiving a claim before shutting Condor down on this machine. Defaults to unset, which means to never shut down. This is primarily intended for *condor_glidein*. Use in other situations is not recommended.

These macros control if the *condor_startd* daemon should perform backfill computations whenever resources would otherwise be idle. See section 3.12.9 on page 412 on Configuring Condor for Running Backfill Jobs for details.

ENABLE_BACKFILL A boolean value that, when `True`, indicates that the machine is willing to perform backfill computations when it would otherwise be idle. This is not a policy expression that is evaluated, it is a simple `True` or `False`. This setting controls if any of the other backfill-related expressions should be evaluated. The default is `False`.

BACKFILLSYSTEM A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported value for this is `"BOINC"`, which stands for the *Berkeley Open Infrastructure for Network Computing*. See <http://boinc.berkeley.edu> for more information about BOINC. There is no default value, administrators must define this.

START_BACKFILL A boolean expression that is evaluated whenever a Condor resource is in the Unclaimed/Idle state and the `ENABLE_BACKFILL` expression is `True`. If `START_BACKFILL` evaluates to `True`, the machine will enter the Backfill state and attempt to spawn a backfill computation. This expression is analogous to the `START` expression that controls when a Condor resource is available to run normal Condor jobs. The default value is `False` (which means do not spawn a backfill job even if the machine is idle and `ENABLE_BACKFILL` expression is `True`). For more information about policy expressions and the Backfill state, see section 3.5 beginning on page 252, especially sections 3.5.5, 3.5.6, and 3.5.7.

EVICT_BACKFILL A boolean expression that is evaluated whenever a Condor resource is in the Backfill state which, when `True`, indicates the machine should immediately kill the currently running backfill computation and return to the Owner state. This expression is a way for administrators to define a policy where interactive users on a machine will cause backfill jobs to be removed. The default value is `False`. For more information about policy expressions and the Backfill state, see section 3.5 beginning on page 252, especially sections 3.5.5, 3.5.6, and 3.5.7.

These macros only apply to the `condor_startd` daemon when it is running on an SMP machine. See section 3.12.7 on page 399 on Configuring The Startd for SMP Machines for details.

STARTD_RESOURCE_PREFIX A string which specifies what prefix to give the unique Condor resources that are advertised on SMP machines. Previously, Condor used the term *virtual machine* to describe these resources, so the default value for this setting was “vm”. However, to avoid confusion with other kinds of virtual machines (the ones created using tools like VMware or Xen), the old *virtual machine* terminology has been changed, and we now use the term *slot*. Therefore, the default value of this prefix is now “slot”. If sites want to keep using “vm”, or prefer something other “slot”, this setting enables sites to define what string the `condor_startd` will use to name the individual resources on an SMP machine.

SLOTS_CONNECTED_TO_CONSOLE An integer which indicates how many of the machine slots the `condor_startd` is representing should be “connected” to the console (in other words, notice when there’s console activity). This defaults to all slots (N in a machine with N CPUs).

SLOTS_CONNECTED_TO_KEYBOARD An integer which indicates how many of the machine slots the `condor_startd` is representing should be “connected” to the keyboard (for remote tty activity, as well as console activity). Defaults to 1.

DISCONNECTED_KEYBOARD_IDLE_BOOST If there are slots not connected to either the keyboard or the console, the corresponding idle time reported will be the time since the `condor_startd` was spawned, plus the value of this macro. It defaults to 1200 seconds (20 minutes). We do this because if the slot is configured not to care about keyboard activity, we want it to be available to Condor jobs as soon as the `condor_startd` starts up, instead of having to wait for 15 minutes or more (which is the default time a machine must be idle before Condor will start a job). If you do not want this boost, set the value to 0. If you change your `START` expression to require more than 15 minutes before a job starts, but you still want jobs to start right away on some of your SMP nodes, increase this macro’s value.

STARTD_SLOT_ATTRS The list of ClassAd attribute names that should be shared across all slots on the same machine. This setting was formerly know as `STARTD_VM_ATTRS` or `STARTD_VM_EXPRS` (before version 6.9.3). For each attribute in the list, the attribute's value is taken from each slot's machine ClassAd and placed into the machine ClassAd of all the other slots within the machine. For example, if the configuration file for a 2-slot machine contains

```
STARTD_SLOT_ATTRS = State, Activity, EnteredCurrentActivity
```

then the machine ClassAd for both slots will contain attributes that will be of the form:

```
slot1_State = "Claimed"
slot1_Activity = "Busy"
slot1_EnteredCurrentActivity = 1075249233
slot2_State = "Unclaimed"
slot2_Activity = "Idle"
slot2_EnteredCurrentActivity = 1075240035
```

The following settings control the number of slots reported for a given SMP host, and what attributes each one has. They are only needed if you do not want to have an SMP machine report to Condor with a separate slot for each CPU, with all shared system resources evenly divided among them. Please read section 3.12.7 on page 400 for details on how to properly configure these settings to suit your needs.

NOTE: You can only change the number of each type of slot the *condor_startd* is reporting with a simple reconfig (such as sending a SIGHUP signal, or using the *condor_reconfig* command). You cannot change the definition of the different slot types with a reconfig. If you change them, you must restart the *condor_startd* for the change to take effect (for example, using *condor_restart_startd*).

NOTE: Prior to version 6.9.3, any settings that included the term “slot” used to use “virtual machine” or “vm”. If you’re looking for information about one of these older settings, search for the corresponding attribute names using “slot”, instead.

MAX_SLOT_TYPES The maximum number of different slot types. Note: this is the maximum number of different *types*, not of actual slots. Defaults to 10. (You should only need to change this setting if you define more than 10 separate slot types, which would be pretty rare.)

SLOT_TYPE<N> This setting defines a given slot type, by specifying what part of each shared system resource (like RAM, swap space, etc) this kind of slot gets. This setting has *no* effect unless you also define `NUM_SLOTS_TYPE<N>`. N can be any integer from 1 to the value of `$(MAX_SLOT_TYPES)`, such as `SLOT_TYPE_1`. The format of this entry can be somewhat complex, so please refer to section 3.12.7 on page 400 for details on the different possibilities.

SLOT_TYPE<N>_PARTITIONABLE A boolean variable that defaults to `False`. When `True`, this slot permits dynamic provisioning, as specified in section 3.12.7.

NUM_SLOTS_TYPE_<N> This macro controls how many of a given slot type are actually reported to Condor. There is no default.

NUM_SLOTS If your SMP machine is being evenly divided, and the slot type settings described above are not being used, this macro controls how many slots will be reported. The default is one slot for each CPU. This setting can be used to reserve some CPUs on an SMP which would not be reported to the Condor pool. You cannot use this parameter to make Condor advertise more slots than there are CPUs on the machine. To do that, use `NUM_CPUS`.

ALLOW_VM_CRUFT A boolean value that Condor sets and uses internally, currently defaulting to `True`. When `True`, Condor looks for configuration variables named with the previously used string `VM` after searching unsuccessfully for variables named with the currently used string `SLOT`. When `False`, Condor does *not* look for variables named with the previously used string `VM` after searching unsuccessfully for the string `SLOT`.

The following macros describe the *cron* capabilities of Condor. The *cron* mechanism is used to run executables (called modules) directly from the *condor_startd* daemon. The output from modules is incorporated into the machine ClassAd generated by the *condor_startd*. These capabilities are used in Hawkeye, but can be used in other situations as well.

These configuration macros are divided into three sets. The three sets occurred as the functionality and usage of Condor's *cron* capabilities evolved. The first set applies to both new and older macros and syntax. The second set applies to the new macros and syntax. The third set applies only to the older (and outdated) macros and syntax.

This first set of configuration macros applies to both new and older macros and syntax.

STARTD_CRON_NAME Defines a logical name to be used in the formation of related configuration macro names. While not required, this macro makes other macros more readable and maintainable. A common example is

```
STARTD_CRON_NAME = HAWKEYE
```

This example allows the naming of other related macros to contain the string "HAWKEYE" in their name.

STARTD_CRON_CONFIG_VAL This configuration variable can be used to specify the *condor_config_val* program which the modules (jobs) should use to get configuration information from the daemon. If this is provided, a environment variable by the same name with the same value will be passed to all modules.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_CONFIG_VAL` to `$(STARTD_CRON_NAME)_CONFIG_VAL`. Example:

```
HAWKEYE_CONFIG_VAL = /usr/local/condor/bin/condor_config_val
```

STARTD_CRON_AUTOPUBLISH Optional setting that determines if the *condor_startd* should automatically publish a new update to the *condor_collector* after any of the cron modules produce output. Beware that enabling this setting can greatly increase the network traffic in a Condor pool, especially when many modules are executed, or if the period in which they run is short. There are three possible (case insensitive) values for this setting:

Never This default value causes the *condor_startd* to not automatically publish updates based on any cron modules. Instead, updates rely on the usual behavior for sending updates, which is periodic, based on the `UPDATE_INTERVAL` configuration setting, or whenever a given slot changes state.

Always Causes the *condor_startd* to always send a new update to the *condor_collector* whenever any module exits.

If_Changed Causes the *condor_startd* to only send a new update to the *condor_collector* if the output produced by a given module is different than the previous output of the same module. The only exception is the `LastUpdate` attribute (automatically set for all cron modules to be the timestamp when the module last ran), which is ignored when `STARTD_CRON_AUTOPUBLISH` is set to `If_Changed`.

Beware that `STARTD_CRON_AUTOPUBLISH` does not honor the `STARTD_CRON_NAME` setting described above. Even if `STARTD_CRON_NAME` is defined, `STARTD_CRON_AUTOPUBLISH` will have the same name.

The following second set of configuration macros applies only to the new macros and syntax. This set is to be used for all new applications.

STARTD_CRON_JOBLIST This configuration variable is defined by a white space separated list of job names (called modules) to run. Each of these is the logical name of the module. This name must be unique (no two modules may have the same name).

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_JOBLIST` to `$(STARTD_CRON_NAME)_JOBLIST`.

STARTD_CRON_<ModuleName>_PREFIX Specifies a string which is prepended by Condor to all attribute names that the module generates. For example, if a prefix is “xyz_”, and an individual attribute is named “abc”, the resulting attribute would be “xyz_abc”. Although it can be quoted, the prefix can contain only alpha-numeric characters.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_PREFIX` to `$(STARTD_CRON_NAME)_<ModuleName>_PREFIX`.

STARTD_CRON_<ModuleName>_EXECUTABLE Used to specify the full path to the executable to run for this module. Note that multiple modules may specify the same executable (although they need to have different names).

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_EXECUTABLE` to `$(STARTD_CRON_NAME)_<ModuleName>_EXECUTABLE`.

STARTD_CRON_<ModuleName>PERIOD The period specifies time intervals at which the module should be run. For periodic modules, this is the time interval that passes between starting the execution of the module. The value may be specified in seconds (append value with the character 's'), in minutes (append value with the character 'm'), or in hours (append value with the character 'h'). As an example, 5m starts the execution of the module every five minutes. If no character is appended to the value, seconds are used as a default. For “Wait For Exit” mode, the value has a different meaning; in this case the period specifies the length of time after the module ceases execution before it is restarted. The minimum valid value of the period is 1 second.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_PERIOD` to `$(STARTD_CRON_NAME)_<ModuleName>_PERIOD`.

STARTD_CRON_<ModuleName>MODE Used to specify the “Mode” in which the module operates. Legal values are “WaitForExit” and “Periodic” (the default).

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_MODE` to `$(STARTD_CRON_NAME)_<ModuleName>_MODE`.

The default “Periodic” mode is used for most modules. In this mode, the module is expected to be started by the *condor_startd* daemon, gather and publish its data, and then exit.

The “WaitForExit” mode is used to specify a module which runs in the “Wait For Exit” mode. In this mode, the *condor_startd* daemon interprets the “period” differently. In this case, it refers to the amount of time to wait after the module exits before restarting it. With a value of 1, the module is kept running nearly continuously.

In general, “Wait For Exit” mode is for modules that produce a periodic stream of updated data, but it can be used for other purposes, as well.

STARTD_CRON_<ModuleName>RECONFIG The “ReConfig” macro is used to specify whether a module can handle HUP signals, and should be sent a HUP signal when the *condor_startd* daemon is reconfigured. The module is expected to reread its configuration at that time. A value of “True” enables this setting, and “False” disables it.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_RECONFIG` to `$(STARTD_CRON_NAME)_<ModuleName>_RECONFIG`.

STARTD_CRON_<ModuleName>KILL The “Kill” macro is applicable on for modules running in the “Periodic” mode. Possible values are “True” and “False” (the default).

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_KILL` to `$(STARTD_CRON_NAME)_<ModuleName>_KILL`.

This macro controls the behavior of the *condor_startd* when it detects that the module’s executable is still running when it is time to start the module for a run. If enabled, the *condor_startd* will kill and restart the process in this condition. If not enabled, the existing process is allowed to continue running.

STARTD_CRON_<ModuleName>_ARGS The command line arguments to pass to the module to be executed.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_ARGS` to `$(STARTD_CRON_NAME)_<ModuleName>_ARGS`.

STARTD_CRON_<ModuleName>_ENV The environment string to pass to the module. The syntax is the same as that of `DAEMONNAME_ENVIRONMENT` in 3.3.9.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_ENV` to `$(STARTD_CRON_NAME)_<ModuleName>_ENV`.

STARTD_CRON_<ModuleName>_CWD The working directory in which to start the module.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_CWD` to `$(STARTD_CRON_NAME)_<ModuleName>_CWD`.

STARTD_CRON_<ModuleName>_OPTIONS A colon separated list of options. Not all combinations of options make sense; when a nonsense combination is listed, the last one in the list is followed.

If `STARTD_CRON_NAME` is defined, then this configuration macro name is changed from `STARTD_CRON_<ModuleName>_OPTIONS` to `$(STARTD_CRON_NAME)_<ModuleName>_OPTIONS`.

- The “WaitForExit” option enables the “Wait For Exit” mode (see above).
- The “ReConfig” option enables the “Reconfig” setting (see above).
- The “NoReConfig” option disables the “Reconfig” setting (see above).
- The “Kill” option enables the “Kill” setting (see above).
- The “NoKill” option disables the “Kill” setting (see above).

Here is a complete configuration example that uses Hawkeye.

```
# Hawkeye Job Definitions
STARTD_CRON_NAME = HAWKEYE

# Job 1
HAWKEYE_JOBLIST = job1
HAWKEYE_job1_PREFIX = prefix_
HAWKEYE_job1_EXECUTABLE = $(MODULES)/job1
HAWKEYE_job1_PERIOD = 5m
HAWKEYE_job1_MODE = WaitForExit
HAWKEYE_job1_KILL = false
HAWKEYE_job1_ARGS = --foo -bar
HAWKEYE_job1_ENV = xyzzy=somevalue
```

```
# Job 2
HAWKEYE_JOBLIST = $(HAWKEYE_JOBLIST) job2
HAWKEYE_job2_PREFIX = prefix_
HAWKEYE_job2_EXECUTABLE = $(MODULES)/job2
HAWKEYE_job2_PERIOD = 1h
HAWKEYE_job2_ENV = lwp=somevalue
```

The following third set of configuration macros applies only to older macros and syntax. This set is documented for completeness and backwards compatibility. Do not use these configuration macros for any new application. Future releases of Condor may disable the use of this set.

STARTD_CRON_JOBS The list of the modules to execute. In Hawkeye, this is usually named **HAWKEYE_JOBS**. This configuration variable is defined by a white space or newline separated list of jobs (called modules) to run, where each module is specified using the format

```
modulename:prefix:executable:period[:options]
```

Each of these fields can be surrounded by matching quote characters (single quote or double quote, but they must match). This allows colon and white space characters to be specified. For example, the following specifies an executable name with a colon and a space in it:

```
foo:foo_:"c:/some dir/foo.exe":10m
```

These individual fields are described below:

- **modulename** The logical name of the module. This must be unique (no two modules may have the same name). See **STARTD_CRON_JOBLIST**
- **prefix** See **STARTD_CRON_<ModuleName>_PREFIX**
- **executable** See **STARTD_CRON_<ModuleName>_EXECUTABLE**
- **period** See **STARTD_CRON_<ModuleName>_PERIOD**
- Several options are available. Using more than one of these options for one module does not make sense. If this happens, the last one in the list is followed. See **STARTD_CRON_<ModuleName>_OPTIONS**
 - The “Continuous” option is used to specify a module which runs in continuous mode (as described above). See the “WaitForExit” and “ReConfig” options which replace “Continuous”.
This option is now deprecated, and its functionality has been replaced by the new “WaitForExit” and “ReConfig” options, which together implement the capabilities of “Continuous”. This option will be removed from a future version of Condor.
 - The “WaitForExit” option
See the discussion of “WaitForExit” in **STARTD_CRON_<ModuleName>_OPTIONS** above.

- The “ReConfig” option
See the discussion of “ReConfig in STARTD_CRON_<ModuleName>_OPTIONS above.
- The “NoReConfig” option
See the discussion of “NoReConfig in STARTD_CRON_<ModuleName>_OPTIONS above.
- The “Kill” option
See the discussion of “Kill” in STARTD_CRON_<ModuleName>_OPTIONS above.
- The “NoKill” option
See the discussion of “NoKill” in STARTD_CRON_<ModuleName>_OPTIONS above.

NOTE: The configuration file parsing logic will strip white space from the beginning and end of continuation lines. Thus, a job list like below will be misinterpreted and will not work as expected:

```
# Hawkeye Job Definitions
HAWKEYE_JOBS =\
    JOB1:prefix:${MODULES}/job1:5m:nokill\
    JOB2:prefix:${MODULES}/job1_co:1h
HAWKEYE_JOB1_ARGS =-foo -bar
HAWKEYE_JOB1_ENV = xyzzy=somevalue
HAWKEYE_JOB2_ENV = lwpi=somevalue
```

Instead, write this as below:

```
# Hawkeye Job Definitions
HAWKEYE_JOBS =

# Job 1
HAWKEYE_JOBS = $(HAWKEYE_JOBS) JOB1:prefix:${MODULES}/job1:5m:nokill
HAWKEYE_JOB1_ARGS =-foo -bar
HAWKEYE_JOB1_ENV = xyzzy=somevalue

# Job 2
HAWKEYE_JOBS = $(HAWKEYE_JOBS) JOB2:prefix:${MODULES}/job2:1h
HAWKEYE_JOB2_ENV = lwpi=somevalue
```

The following macros control the optional computation of resource availability statistics in the *condor_startd*.

STARTD_COMPUTE_AVAIL_STATS A boolean that determines if the *condor_startd* computes resource availability statistics. The default is False.

If `STARTD_COMPUTE_AVAIL_STATS = True`, the *condor_startd* will define the following ClassAd attributes for resources:

AvailTime The proportion of the time (between 0.0 and 1.0) that this resource has been in a state other than Owner.

LastAvailInterval The duration (in seconds) of the last period between Owner states.

The following attributes will also be included if the resource is not in the Owner state:

AvailSince The time at which the resource last left the Owner state. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

AvailTimeEstimate Based on past history, an estimate of how long the current period between Owner states will last.

STARTDAVAILCONFIDENCE A floating point number representing the confidence level of the *condor_startd* daemon's AvailTime estimate. By default, the estimate is based on the 80th percentile of past values (that is, the value is initially set to 0.8).

STARTD_MAX_AVAIL_PERIOD_SAMPLES An integer that limits the number of samples of past available intervals stored by the *condor_startd* to limit memory and disk consumption. Each sample requires 4 bytes of memory and approximately 10 bytes of disk space.

The following configuration variables support java universe jobs.

JAVA The full path to the Java interpreter (the Java Virtual Machine).

JAVA_MAXHEAP_ARGUMENT An incomplete command line argument to the Java interpreter (the Java Virtual Machine) to specify the switch name for the Maxheap Argument. Condor uses it to construct the maximum heap size for the Java Virtual Machine. For example, the value for the Sun JVM is *-Xmx*.

JAVA_CLASSPATH_ARGUMENT The command line argument to the Java interpreter (the Java Virtual Machine) that specifies the Java Classpath. Classpath is a Java-specific term that denotes the list of locations (*.jar* files and/or directories) where the Java interpreter can look for the Java class files that a Java program requires.

JAVA_CLASSPATH_SEPARATOR The single character used to delimit constructed entries in the Classpath for the given operating system and Java Virtual Machine. If not defined, the operating system is queried for its default Classpath separator.

JAVA_CLASSPATH_DEFAULT A list of path names to *.jar* files to be added to the Java Classpath by default. The comma and/or space character delimits list entries.

JAVA_EXTRA_ARGUMENTS A list of additional arguments to be passed to the Java executable.

These macros control the system of job hooks invoked by the *condor_startd* to optionally fetch work. See section 4.4 on page 460 on "Job Hooks" for more details.

SLOTN_JOB_HOOK_KEYWORD The keyword used to define which set of hooks a particular compute slot should invoke. Note that the “N” in “SLOTN” should be replaced with the slot identification number, for example, on slot1, this setting would be called [SLOT1_JOB_HOOK_KEYWORD. There is no default keyword. Sites that wish to use these job hooks must explicitly define the keyword (and the corresponding hook paths).

STARTD_JOB_HOOK_KEYWORD The keyword used to define which set of hooks a particular *condor_startd* should invoke. This setting is only used if a slot-specific keyword is not defined for a given compute slot. There is no default keyword. Sites that wish to use these job hooks must explicitly define the keyword (and the corresponding hook paths).

HOOK_FETCH_WORK The full path to the program to invoke whenever the *condor_startd* wants to fetch work. The actual configuration setting must be prefixed with a hook keyword. There is no default.

HOOK_REPLY_CLAIM The full path to the program to invoke whenever the *condor_startd* finishes fetching a job and decides what to do with it. The actual configuration setting must be prefixed with a hook keyword. There is no default.

HOOK_EVICT_CLAIM The full path to the program to invoke whenever the *condor_startd* needs to evict a fetched claim. The actual configuration setting must be prefixed with a hook keyword. There is no default.

FetchWorkDelay An expression that defines the number of seconds that the *condor_startd* should wait after an invocation of **HOOK_FETCH_WORK** completes before the hook should be invoked again. The expression is evaluated in the context of the slot ClassAd, and the ClassAd of the currently running job (if any). The expression must evaluate to an integer. If not defined, the *condor_startd* will wait 300 seconds (five minutes) between attempts to fetch work. For more information about this expression, see section 4.4.1 on page 465.

These macros control the power management capabilities of the *condor_startd* to optionally put the machine in to a low power state. See section 3.15 on page 426 on Power Management for more details.

HIBERNATE_CHECK_INTERVAL An integer number of seconds that determines how often the *condor_startd* checks to see if the machine is ready to enter a low power state. The default value is 0, which disables the check. If not 0, the **HIBERNATE** expression is evaluated within the context of each slot at the given interval. If used, a value 300 (5 minutes) is recommended.

As a special case, the interval is ignored when the machine has just returned from a low power state (excluding shutdown (5)). In order to avoid machines from volleying between a running state and a low power state, an hour of uptime is enforced after a machine has been woken. After the hour has passed, regular checks resume.

HIBERNATE An string expression that represents lower power state. When this state name evaluates to a valid non-“NONE” state (see below), causes Condor to put the machine into a low power state given by the evaluation of the expression. The following names are supported (and are not case sensitive):

"NONE", "0": No-op: do not enter a low power state
 "S1", "1", "STANDBY", "SLEEP": On Windows, Sleep (standby)
 "S2", "2": On Windows, Sleep (standby)
 "S3", "3", "RAM", "MEM": Sleep (standby)
 "S4", "4", "DISK", "HIBERNATE": Hibernate
 "S5", "5", "SHUTDOWN": Shutdown (soft-off)

The HIBERNATE expression is written in terms of the S-states as defined in the Advanced Configuration and Power Interface (ACPI) specification. The S-states take the form S_n , where n is an integer in the range 0 to 5, inclusive. The number that results from evaluating the expression determines which S-state to enter. The n from S_n notation was adopted because at this junction in time it appears to be the standard naming scheme for power states on several popular Operating Systems, including various flavors of Windows and Linux distributions. The other strings ("RAM", "DISK", etc.) are provided for ease of configuration.

Since this expression is evaluated in the context of each slot on the machine, any one slot has veto power over the other slots. If the evaluation of HIBERNATE in one slot evaluates to "NONE" or "0", then the machine will not be placed into a low power state. On the other hand, if all slots evaluate to a non-zero value, but differ in value, then the largest value is used as the representative power state.

Strings that do not match any in the table above are treated as "NONE".

LINUX_HIBERNATION_METHOD A string that can be used to override the default search used by Condor on Linux platforms to detect the hibernation method to use. The default behavior orders its search with:

1. Detect and use the *pm-utils* command line tools. The corresponding string is defined with "pm-utils".
2. Detect and use the directory in the virtual file system `/sys/power`. The corresponding string is defined with "`/sys`".
3. Detect and use the directory in the virtual file system `/proc/acpi`. The corresponding string is defined with "`/proc`".

To override this ordered search behavior, and force the use of one particular method, set LINUX_HIBERNATION_METHOD to one of the defined strings.

OFFLINE_LOG The full path and file name of a file that stores machine ClassAds for every hibernating machine. This forms a persistent storage of these ClassAds, in case the *condor_collector* daemon crashes.

To avoid *condor_preem* removing this log, place it in a directory other than the directory defined by `$(SPOOL)`. Alternatively, if this log file is to go in the directory defined by `$(SPOOL)`, add the file to the list given by `VALID_SPOOL_FILES`.

OFFLINE_EXPIRE_ADS_AFTER An integer number of seconds specifying the lifetime of the persistent machine ClassAd representing a hibernating machine. Defaults to the largest 32-bit integer.

3.3.11 condor_schedd Configuration File Entries

These macros control the *condor_schedd*.

SHADOW This macro determines the full path of the *condor_shadow* binary that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

START_LOCAL_UNIVERSE A boolean value that defaults to `True`. The *condor_schedd* uses this macro to determine whether to start a **local** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **local** universe job that it has. For each job, if the `START_LOCAL_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **local** universe jobs to execute concurrently. The attribute `TotalLocalJobsRunning` is supplied by *condor_schedd*'s ClassAd:

```
START_LOCAL_UNIVERSE = TotalLocalJobsRunning < 10
```

STARTER_LOCAL The complete path and executable name of the *condor_starter* to run for **local** universe jobs. This variable's value is defined in the initial configuration provided with Condor as

```
STARTER_LOCAL = $(SBIN)/condor_starter
```

This variable would only be modified or hand added into the configuration for a pool to be upgraded from one running a version of Condor that existed before the **local** universe to one that includes the **local** universe, but without utilizing the newer, provided configuration files.

START_SCHEDULER_UNIVERSE A boolean value that defaults to `True`. The *condor_schedd* uses this macro to determine whether to start a **scheduler** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **scheduler** universe job that it has. For each job, if the `START_SCHEDULER_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **scheduler** universe jobs to execute concurrently. The attribute `TotalSchedulerJobsRunning` is supplied by *condor_schedd*'s ClassAd:

```
START_SCHEDULER_UNIVERSE = TotalSchedulerJobsRunning < 10
```

MAX_JOBS_RUNNING This macro limits the number of processes spawned by a given *condor_schedd*, for all job universes except the grid universe. See section 2.4.1. This includes, but is not limited to *condor_shadow* processes, and scheduler universe processes, including *condor_dagman*. The actual number of *condor_shadows* may be less if you have reached your `$(RESERVED_SWAP)` limit. This macro has a default value of 200.

MAX_JOBS_SUBMITTED This integer value limits the number of jobs permitted in a *condor_schedd* daemon's queue. Submission of a new cluster of jobs fails, if the total number of jobs would exceed this limit. The default value for this variable is the largest positive integer value.

MAX_SHADOW_EXCEPTIONS This macro controls the maximum number of times that *condor_shadow* processes can have a fatal error (exception) before the *condor_schedd* will relinquish the match associated with the dying shadow. Defaults to 5.

MAX_PENDING_STARTD_CONTACTS An integer value that limits the number of simultaneous connection attempts by the *condor_schedd* when it is requesting claims from one or more *condor_startd* daemons. The intention is to protect the *condor_schedd* from being overloaded by authentication operations. The default value is 0. The special value 0 indicates no limit.

MAX_CONCURRENT_DOWNLOADS This specifies the maximum number of simultaneous transfers of output files from execute machines to the submit machine. The limit applies to all jobs submitted from the same *condor_schedd*. The default is 10. A setting of 0 means unlimited transfers. This limit currently does not apply to grid universe jobs or standard universe jobs, and it also does not apply to streaming output files. When the limit is reached, additional transfers will queue up and wait before proceeding.

MAX_CONCURRENT_UPLOADS This specifies the maximum number of simultaneous transfers of input files from the submit machine to execute machines. The limit applies to all jobs submitted from the same *condor_schedd*. The default is 10. A setting of 0 means unlimited transfers. This limit currently does not apply to grid universe jobs or standard universe jobs. When the limit is reached, additional transfers will queue up and wait before proceeding.

SCHEDD_QUERY_WORKERS This specifies the maximum number of concurrent sub-processes that the *condor_schedd* will spawn to handle queries. The setting is ignored in Windows. In Unix, the default is 3. If the limit is reached, the next query will be handled in the *condor_schedd*'s main process.

SCHEDD_INTERVAL This macro determines the maximum interval for both how often the *condor_schedd* sends a ClassAd update to the *condor_collector* and how often the *condor_schedd* daemon evaluates jobs. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

SCHEDD_INTERVAL_TIMESLICE The bookkeeping done by the *condor_schedd* takes more time when there are large numbers of jobs in the job queue. However, when it is not too expensive to do this bookkeeping, it is best to keep the collector up to date with the latest state of the job queue. Therefore, this macro is used to adjust the bookkeeping interval so that it is done more frequently when the cost of doing so is relatively small, and less frequently when the cost is high. The default is 0.05, which means the schedd will adapt its bookkeeping interval to consume no more than 5% of the total time available to the schedd. The lower bound is configured by `SCHEDD_MIN_INTERVAL` (default 5 seconds), and the upper bound is configured by `SCHEDD_INTERVAL` (default 300 seconds).

JOB_START_COUNT This macro works together with the `JOB_START_DELAY` macro to throttle job starts. The default and minimum values for this integer configuration variable are both 1.

JOB_START_DELAY This integer-valued macro works together with the `JOB_START_COUNT` macro to throttle job starts. The *condor_schedd* daemon starts $\$(JOB_START_COUNT)$ jobs at a time, then delays for $\$(JOB_START_DELAY)$ seconds before starting the next set of jobs. This delay prevents a sudden, large load on resources required by the jobs during their start up phase. The resulting job start rate averages as fast as $(\$(JOB_START_COUNT))/\(JOB_START_DELAY) jobs/second. This configuration variable is also used during the graceful shutdown of the *condor_schedd* daemon. During graceful shutdown, this macro determines the wait time in between requesting each *condor_shadow* daemon to gracefully shut down. It is defined in terms of seconds and defaults to 0, which means jobs will be started as fast as possible. If you wish to throttle the rate of specific types of jobs, you can use the job attribute `NextJobStartDelay`.

MAX_NEXT_JOB_START_DELAY An integer number of seconds representing the maximum allowed value of the job ClassAd attribute `NextJobStartDelay`. It defaults to 600, which is 10 minutes.

JOB_IS_FINISHED_INTERVAL The *condor_schedd* maintains a list of jobs that are ready to permanently leave the job queue, e.g. they have completed or been removed. This integer-valued macro specifies a delay in seconds to place between the taking jobs permanently out of the queue. The default value is 0, which tells the *condor_schedd* to not impose any delay.

ALIVE_INTERVAL An initial value for an integer number of seconds defining how often the *condor_schedd* sends a UDP keep alive message to any *condor_startd* it has claimed. When the *condor_schedd* claims a *condor_startd*, the *condor_schedd* tells the *condor_startd* how often it is going to send these messages. The utilized interval for sending keep alive messages is the smallest of the two values `ALIVE_INTERVAL` and the expression `JobLeaseDuration/3`, formed with the job ClassAd attribute `JobLeaseDuration`. The value of the interval is further constrained by the floor value of 10 seconds. If the *condor_startd* does not receive any of these keep alive messages during a certain period of time (defined via `MAX_CLAIM_ALIVES_MISSED`, described on page 181) the *condor_startd* releases the claim, and the *condor_schedd* no longer pays for the resource (in terms of user priority in the system). The macro is defined in terms of seconds and defaults to 300, which is 5 minutes.

REQUEST_CLAIM_TIMEOUT This macro sets the time (in seconds) that the *condor_schedd* will wait for a claim to be granted by the *condor_startd*. The default is 30 minutes. This is only likely to matter if the *condor_startd* has an existing claim and it takes a long time for the existing claim to be preempted due to `MaxJobRetirementTime`. Once a request times out, the *condor_schedd* will simply begin the process of finding a machine for the job all over again.

Normally, it is not a good idea to set this to be very small (e.g. a few minutes). Doing so can lead to failure to preempt, because the preempting job will spend a significant fraction of its time waiting to be re-matched. During that time, it would miss out on any opportunity to run if the job it is trying to preempt gets out of the way.

SHADOW_SIZE_ESTIMATE This macro sets the estimated virtual memory size of each *condor_shadow* process. Specified in kilobytes. The default varies from platform to platform.

SHADOW_RENICE_INCREMENT When the *condor_schedd* spawns a new *condor_shadow*, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority so that the processes run with less priority than other tasks on the machine. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 0.

SCHED_UNIV_RENICE_INCREMENT Analogous to `JOB_RENICE_INCREMENT` and `SHADOW_RENICE_INCREMENT`, scheduler universe jobs can be given a nice-level. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 0.

QUEUE_CLEAN_INTERVAL The *condor_schedd* maintains the job queue on a given machine. It does so in a persistent way such that if the *condor_schedd* crashes, it can recover a valid state of the job queue. The mechanism it uses is a transaction-based log file (the `job_queue.log` file, not the `SchedLog` file). This file contains an initial state of the job queue, and a series of transactions that were performed on the queue (such as new jobs submitted, jobs completing, and checkpointing). Periodically, the *condor_schedd* will go through this log, truncate all the transactions and create a new file with containing only the new initial state of the log. This is a somewhat expensive operation, but it speeds up when the *condor_schedd* restarts since there are fewer transactions it has to play to figure out what state the job queue is really in. This macro determines how often the *condor_schedd* should rework this queue to cleaning it up. It is defined in terms of seconds and defaults to 86400 (once a day).

WALL_CLOCK_CKPT_INTERVAL The job queue contains a counter for each job's "wall clock" run time, i.e., how long each job has executed so far. This counter is displayed by *condor_q*. The counter is updated when the job is evicted or when the job completes. When the *condor_schedd* crashes, the run time for jobs that are currently running will not be added to the counter (and so, the run time counter may become smaller than the CPU time counter). The *condor_schedd* saves run time "checkpoints" periodically for running jobs so if the *condor_schedd* crashes, only run time since the last checkpoint is lost. This macro controls how often the *condor_schedd* saves run time checkpoints. It is defined in terms of seconds and defaults to 3600 (one hour). A value of 0 will disable wall clock checkpoints.

QUEUE_ALL_USERS_TRUSTED Defaults to False. If set to True, then unauthenticated users are allowed to write to the queue, and also we always trust whatever the `Owner` value is set to be by the client in the job ad. This was added so users can continue to use the SOAP web-services interface over HTTP (w/o authenticating) to submit jobs in a secure, controlled environment – for instance, in a portal setting.

QUEUE_SUPER_USERS This macro determines what user names on a given machine have *super-user access* to the job queue, meaning that they can modify or delete the job ClassAds of other users. (Normally, you can only modify or delete ClassAds from the job queue that you own). Whatever user name corresponds with the UID that Condor is running as (usually the Unix user `condor`) will automatically be included in this list because that is needed for Condor's proper functioning. See section 3.6.11 on UIDs in Condor for more details on this. By default, we give root the ability to remove other user's jobs, in addition to user `condor`.

SCHEDDLOCK This macro specifies what lock file should be used for access to the `SchedLog` file. It must be a separate file from the `SchedLog`, since the `SchedLog` may be rotated

and synchronization across log file rotations is desired. This macro is defined relative to the `$(LOCK)` macro.

SCHEDD_NAME Used to give an alternative value to the Name attribute in the *condor_schedd*'s ClassAd.

See the description of `MASTER_NAME` in section 3.3.9 on page 177 for defaults and composition of valid Condor daemon names. Also, note that if the `MASTER_NAME` setting is defined for the *condor_master* that spawned a given *condor_schedd*, that name will take precedence over whatever is defined in `SCHEDD_NAME`.

SCHEDDATTRS This macro is described in section 3.3.5 as `<SUBSYS>_ATTRS`.

SCHEDD_DEBUG This macro (and other settings related to debug logging in the *condor_schedd*) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

SCHEDD_ADDRESS_FILE This macro is described in section 3.3.5 as `<SUBSYS>_ADDRESS_FILE`.

SCHEDD_EXECUTE A directory to use as a temporary sandbox for local universe jobs. Defaults to `$(SPOOL)/execute`.

FLOCK_NEGOTIATOR_HOSTS This macro defines a list of negotiator host names (not including the local `$(NEGOTIATOR_HOST)` machine) for pools in which the *condor_schedd* should attempt to run jobs. Hosts in the list should be in order of preference. The *condor_schedd* will only send a request to a central manager in the list if the local pool and pools earlier in the list are not satisfying all the job requests. `$(HOSTALLOW_NEGOTIATOR_SCHEDD)` (see section 3.3.5) must also be configured to allow negotiators from all of the `$(FLOCK_NEGOTIATOR_HOSTS)` to contact the *condor_schedd*. Please make sure the `$(NEGOTIATOR_HOST)` is first in the `$(HOSTALLOW_NEGOTIATOR_SCHEDD)` list. Similarly, the central managers of the remote pools must be configured to listen to requests from this *condor_schedd*.

FLOCK_COLLECTOR_HOSTS This macro defines a list of collector host names for pools in which the *condor_schedd* should attempt to run jobs. The collectors must be specified in order, corresponding to the `$(FLOCK_NEGOTIATOR_HOSTS)` list. In the typical case, where each pool has the collector and negotiator running on the same machine, `$(FLOCK_COLLECTOR_HOSTS)` should have the same definition as `$(FLOCK_NEGOTIATOR_HOSTS)`.

NEGOTIATE_ALL_JOBS_IN_CLUSTER If this macro is set to False (the default), when the *condor_schedd* fails to start an idle job, it will not try to start any other idle jobs in the same cluster during that negotiation cycle. This makes negotiation much more efficient for large job clusters. However, in some cases other jobs in the cluster can be started even though an earlier job can't. For example, the jobs' requirements may differ, because of different disk space, memory, or operating system requirements. Or, machines may be willing to run only some jobs in the cluster, because their requirements reference the jobs' virtual memory size or other attribute. Setting this macro to True will force the *condor_schedd* to try to start all idle jobs in each negotiation cycle. This will make negotiation cycles last longer, but it will ensure that all jobs that can be started will be started.

PERIODIC_EXPR_INTERVAL This macro determines the minimum period, in seconds, between evaluation of periodic job control expressions, such as `periodic_hold`, `periodic_release`, and `periodic_remove`, given by the user in a Condor submit file. By default, this value is 60 seconds. A value of 0 prevents the *condor_schedd* from performing the periodic evaluations.

PERIODIC_EXPR_TIMESLICE This macro is used to adapt the frequency with which the *condor_schedd* evaluates periodic job control expressions. When the job queue is very large, the cost of evaluating all of the ClassAds is high, so in order for the *condor_schedd* to continue to perform well, it makes sense to evaluate these expressions less frequently. The default time slice is 0.01, so the *condor_schedd* will set the interval between evaluations so that it spends only 1% of its time in this activity. The lower bound for the interval is configured by `PERIODIC_EXPR_INTERVAL` (default 60 seconds).

SYSTEM_PERIODIC_HOLD This expression behaves identically to the job expression `periodic_hold`, but it is evaluated by the *condor_schedd* daemon individually for each job in the queue. It defaults to `False`. When `True`, it causes the job to stop running and go on hold. Here is an example that puts jobs on hold if they have been restarted too many times, have an unreasonably large virtual memory `ImageSize`, or have unreasonably large disk usage for an invented environment.

```
SYSTEM_PERIODIC_HOLD = \
  (JobStatus == 1 || JobStatus == 2) && \
  (JobRunCount > 10 || ImageSize > 3000000 || DiskUsage > 10000000)
```

SYSTEM_PERIODIC_RELEASE This expression behaves identically to the job expression `periodic_release`, but it is evaluated by the *condor_schedd* daemon individually for each job in the queue. It defaults to `False`. When `True`, it causes a held job to return to the idle state. Here is an example that releases jobs from hold if they have tried to run less than 20 times, have most recently been on hold for over 20 minutes, and have gone on hold due to “Connection timed out” when trying to execute the job, because the file system containing the job’s executable is temporarily unavailable.

```
SYSTEM_PERIODIC_RELEASE = \
  (JobRunCount < 20 && CurrentTime - EnteredCurrentStatus > 1200 ) && ( \
    (HoldReasonCode == 6 && HoldReasonSubCode == 110) \
  )
```

SYSTEM_PERIODIC_REMOVE This expression behaves identically to the job expression `periodic_remove`, but it is evaluated by the *condor_schedd* daemon individually for each job in the queue. It defaults to `False`. When `True`, it causes the job to be removed from the queue. Here is an example that removes jobs which have been on hold for 30 days:

```
SYSTEM_PERIODIC_REMOVE = \
  (JobStatus == 5 && CurrentTime - EnteredCurrentStatus > 3600*24*30)
```

SCHEDD_ASSUME_NEGOTIATOR_GONE This macro determines the period, in seconds, that the *condor_schedd* will wait for the *condor_negotiator* to initiate a negotiation cycle before the schedd will simply try to claim any local *condor_startd*. This allows for a machine that is acting as both a submit and execute node to run jobs locally if it cannot communicate with the central manager. The default value, if not specified, is 4 x \$(NEGOTIATOR_INTERVAL). If \$(NEGOTIATOR_INTERVAL) is not defined, then SCHEDD_ASSUME_NEGOTIATOR_GONE will default to 1200 (20 minutes).

SCHEDD_ROUND_ATTR_<xxxx> This is used to round off attributes in the job ClassAd so that similar jobs may be grouped together for negotiation purposes. There are two cases. One is that a percentage such as 25% is specified. In this case, the value of the attribute named <xxxx> in the job ClassAd will be rounded up to the next multiple of the specified percentage of the values order of magnitude. For example, a setting of 25% will cause a value near 100 to be rounded up to the next multiple of 25 and a value near 1000 will be rounded up to the next multiple of 250. The other case is that an integer, such as 4, is specified instead of a percentage. In this case, the job attribute is rounded up to the specified number of decimal places. Replace <xxxx> with the name of the attribute to round, and set this macro equal to the number of decimal places to round up. For example, to round the value of job ClassAd attribute foo up to the nearest 100, set

```
SCHEDD_ROUND_ATTR_foo = 2
```

When the schedd rounds up an attribute value, it will save the raw (un-rounded) actual value in an attribute with the same name appended with “_RAW”. So in the above example, the raw value will be stored in attribute foo_RAW in the job ClassAd. The following are set by default:

```
SCHEDD_ROUND_ATTR_ImageSize = 25%
SCHEDD_ROUND_ATTR_ExecutableSize = 25%
SCHEDD_ROUND_ATTR_DiskUsage = 25%
SCHEDD_ROUND_ATTR_NumCkpts = 4
```

Thus, an ImageSize near 100MB will be rounded up to the next multiple of 25MB. If your batch slots have less memory or disk than the rounded values, it may be necessary to reduce the amount of rounding, because the job requirements will not be met.

SCHEDD_BACKUP_SPOOL This macro is used to enable the *condor_schedd* to make a backup of the job queue as it starts. If set to “True”, the *condor_schedd* will create host specific a backup of the current spool file to the spool directory. This backup file will be overwritten each time the *condor_schedd* starts. SCHEDD_BACKUP_SPOOL defaults to “False”.

MPI_CONDOR_RSH_PATH The complete path to the special version of *rsh* that is required to spawn MPI jobs under Condor. \$(LIBEXEC) is the proper value for this configuration variable, required when running MPI dedicated jobs.

SCHEDD_PREEMPTION_REQUIREMENTS This boolean expression is utilized only for machines allocated by a dedicated scheduler. When True, a machine becomes a candidate for job preemption. This configuration variable has no default; when not defined, preemption will never be considered.

SCHEDD_PREEMPTIONRANK This floating point value is utilized only for machines allocated by a dedicated scheduler. It is evaluated in context of a job ClassAd, and it represents a machine's preference for running a job. This configuration variable has no default; when not defined, preemption will never be considered.

ParallelSchedulingGroup For parallel jobs which must be assigned within a group of machines (and not cross group boundaries), this configuration variable identifies members of a group. Each machine within a group sets this configuration variable with a string that identifies the group.

PER_JOB_HISTORY_DIR If set to a directory writable by the Condor user, when a job leaves the *condor_schedd*'s queue, a copy of its ClassAd will be written in that directory. The files are named "history." with the job's cluster and process number appended. For example, job 35.2 will result in a file named "history.35.2". Condor does not rotate or delete the files, so without an external entity to clean the directory it can grow very large. This option defaults to being unset. When not set, no such files are written.

DEDICATED_SCHEDULER_USE_FIFO When this parameter is set to true (the default), parallel and mpi universe jobs will be scheduled in a first-in, first-out manner. When set to false, parallel and mpi jobs are scheduled using a best-fit algorithm. Using the best-fit algorithm is not recommended, as it can cause starvation.

SCHEDD_SEND_VACATE_VIA_TCP A boolean value that defaults to False. When True, the *condor_schedd* daemon sends vacate signals via TCP, instead of the default UDP.

SCHEDD_CLUSTER_INITIAL_VALUE An integer that specifies the initial cluster number value to use within a job id when a job is first submitted. The default value is 1.

SCHEDD_CLUSTER_INCREMENT_VALUE A positive integer that defaults to 1, representing a stride used for assignment of cluster numbers within a job id. When a job is submitted, the job will be assigned a job id. The cluster number of the job id will be equal to the previous cluster number used plus the value of this setting.

3.3.12 condor_shadow Configuration File Entries

These settings affect the *condor_shadow*.

SHADOW_LOCK This macro specifies the lock file to be used for access to the ShadowLog file. It must be a separate file from the ShadowLog, since the ShadowLog may be rotated and you want to synchronize access across log file rotations. This macro is defined relative to the `$(LOCK)` macro.

SHADOW_DEBUG This macro (and other settings related to debug logging in the shadow) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

SHADOW_QUEUE_UPDATE_INTERVAL The amount of time (in seconds) between ClassAd updates that the *condor_shadow* daemon sends to the *condor_schedd* daemon. Defaults to 900 (15 minutes).

SHADOW_LAZY_QUEUE_UPDATE This boolean macro specifies if the *condor_shadow* should immediately update the job queue for certain attributes (at this time, it only effects the NumJobStarts and NumJobReconnects counters) or if it should wait and only update the job queue on the next periodic update. There is a trade-off between performance and the semantics of these attributes, which is why the behavior is controlled by a configuration macro. If the *condor_shadow* do not use a lazy update, and immediately ensures the changes to the job attributes are written to the job queue on disk, the semantics for the attributes are very solid (there's only a tiny chance that the counters will be out of sync with reality), but this introduces a potentially large performance and scalability problem for a busy *condor_schedd*. If the *condor_shadow* uses a lazy update, there's no additional cost to the *condor_schedd*, but it means that *condor_q* and Quill won't immediately see the changes to the job attributes, and if the *condor_shadow* happens to crash or be killed during that time, the attributes are never incremented. Given that the most obvious usage of these counter attributes is for the periodic user policy expressions (which are evaluated directly by the *condor_shadow* using its own copy of the job's classified ad, which is immediately updated in either case), and since the additional cost for aggressive updates to a busy *condor_schedd* could potentially cause major problems, the default is True to do lazy, periodic updates.

COMPRESS_PERIODIC_CHKPT This boolean macro specifies whether the shadow should instruct applications to compress periodic checkpoints (when possible). The default is False.

COMPRESS_VACATE_CHKPT This boolean macro specifies whether the shadow should instruct applications to compress vacate checkpoints (when possible). The default is False.

PERIODIC_MEMORY_SYNC This boolean value specifies whether the *condor_shadow* should instruct applications to commit dirty memory pages to swap space during a periodic checkpoint. The default is False. This potentially reduces the number of dirty memory pages at vacate time, thereby reducing swapping activity on the remote machine.

SLOW_CHKPT_SPEED This macro specifies the speed at which vacate checkpoints should be written, in kilobytes per second. If zero (the default), vacate checkpoints are written as fast as possible. Writing vacate checkpoints slowly can avoid overwhelming the remote machine with swapping activity.

SHADOW_JOB_CLEANUP_RETRY_DELAY This integer specifies the number of seconds to wait between tries to commit the final update to the job ClassAd in the *condor_schedd*'s job queue. The default is 30.

SHADOW_MAX_JOB_CLEANUP_RETRIES This integer specifies the number of times to try committing the final update to the job ClassAd in the *condor_schedd*'s job queue. The default is 5.

3.3.13 condor_starter Configuration File Entries

These settings affect the *condor_starter*.

EXEC_TRANSFERATTEMPTS Sometimes due to a router misconfiguration, kernel bug, or other network problem, the transfer of the initial checkpoint from the submit machine to the execute machine will fail midway through. This parameter allows a retry of the transfer a certain number of times that must be equal to or greater than 1. If this parameter is not specified, or specified incorrectly, then it will default to three. If the transfer of the initial executable fails every attempt, then the job goes back into the idle state until the next renegotiation cycle.

NOTE: : This parameter does not exist in the NT starter.

JOB_NICE_INCREMENT When the *condor_starter* spawns a Condor job, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority, such that these processes do not interfere with interactive use of the machine. For machines with lots of real memory and swap space, such that the only scarce resource is CPU time, use this macro in conjunction with a policy that allows Condor to always start jobs on the machines. Condor jobs would always run, but interactive response on the machines would never suffer. A user most likely will not notice Condor is running jobs. See section 3.5 on Startd Policy Configuration for more details on setting up a policy for starting and stopping jobs on a given machine.

The integer value is set by the *condor_starter* daemon for each job just before the job runs. The range of allowable values are integers in the range of 0 to 19 (inclusive), with a value of 19 being the lowest priority. If the integer value is outside this range, then on a Unix machine, a value greater than 19 is auto-decreased to 19; a value less than 0 is treated as 0. For values outside this range, a Windows machine ignores the value and uses the default instead. The default value is 10, which maps to the idle priority class on a Windows machine.

STARTER_LOCAL_LOGGING This macro determines whether the starter should do local logging to its own log file, or send debug information back to the *condor_shadow* where it will end up in the ShadowLog. It defaults to True.

STARTER_DEBUG This setting (and other settings related to debug logging in the starter) is described above in section 3.3.4 as \$(<SUBSYS>_DEBUG).

STARTER_UPDATE_INTERVAL An integer value representing the number of seconds between ClassAd updates that the *condor_starter* daemon sends to the *condor_shadow* and *condor_startd* daemons. Defaults to 300 (5 minutes).

STARTER_UPDATE_INTERVAL_TIMESLICE A floating point value, specifying the highest fraction of time that the *condor_starter* daemon should spend collecting monitoring information about the job, such as disk usage. The default value is 0.1. If monitoring, such as checking disk usage takes a long time, the *condor_starter* will monitor less frequently than specified by STARTER_UPDATE_INTERVAL.

USER_JOB_WRAPPER The full path to an executable or script. This macro allows an administrator to specify a wrapper script to handle the execution of all user jobs. If specified, Condor never directly executes a job, but instead invokes the program specified by this macro. The command-line arguments passed to this program will include the full-path to the actual user job which should be executed, followed by all the command-line parameters to pass to the user job. This wrapper program must ultimately replace its image with the user job; in other

words, it must `exec ()` the user job, not `fork ()` it. For instance, if the wrapper program is a C/Korn shell script, the last line of execution should be:

```
exec $*
```

This can potentially lose information about the arguments. Any argument with embedded white space will be split into multiple arguments. For example the argument "argument one" will become the two arguments "argument" and "one". For Bourne type shells (sh, bash, ksh), the following preserves the arguments:

```
exec "$@"
```

For the C type shells (csh, tcsh), the following preserves the arguments:

```
exec $*:q
```

For Windows machines, the wrapper will either be a batch script (with a file extension of `.bat` or `.cmd`) or an executable (with a file extension of `.exe` or `.com`).

USE_VISIBLE_DESKTOP This setting is only meaningful on Windows machines. If True, Condor will allow the job to create windows on the desktop of the execute machine and interact with the job. This is particularly useful for debugging why an application will not run under Condor. If False, Condor uses the default behavior of creating a new, non-visible desktop to run the job on. See section 6.2 for details on how Condor interacts with the desktop.

STARTER_JOB_ENVIRONMENT This macro sets the default environment inherited by jobs. The syntax is the same as the syntax for environment settings in the job submit file (see page 797). If the same environment variable is assigned by this macro and by the user in the submit file, the user's setting takes precedence.

JOB_INHERITS_STARTER_ENVIRONMENT A boolean value that defaults to False. When True, it causes jobs to inherit all environment variables from the *condor_starter*. This is useful for glidein jobs that need to access environment variables from the batch system running the glidein daemons. When both the user job and **STARTER_JOB_ENVIRONMENT** define an environment variable that is in the *condor_starter*'s environment, the user job's definition takes precedence. This variable does not apply to standard universe jobs.

STARTER_UPLOAD_TIMEOUT An integer value that specifies the network communication time-out to use when transferring files back to the submit machine. The default value is set by the *condor_shadow* daemon to 300. Increase this value if the disk on the submit machine cannot keep up with large bursts of activity, such as many jobs all completing at the same time.

3.3.14 condor_submit Configuration File Entries

DEFAULT_UNIVERSE The universe under which a job is executed may be specified in the submit description file. If it is not specified in the submit description file, then this variable specifies the universe (when defined). If the universe is not specified in the submit description file, and if this variable is not defined, then the default universe for a job will be the vanilla universe.

If you want *condor_submit* to automatically append an expression to the Requirements expression or Rank expression of jobs at your site use the following macros:

APPEND_REQ_VANILLA Expression to be appended to vanilla job requirements.

APPEND_REQ_STANDARD Expression to be appended to standard job requirements.

APPEND_REQUIREMENTS Expression to be appended to any type of universe jobs. However, if **APPEND_REQ_VANILLA** or **APPEND_REQ_STANDARD** is defined, then ignore the **APPEND_REQUIREMENTS** for those universes.

APPEND_RANK Expression to be appended to job rank. **APPEND_RANK_STANDARD** or **APPEND_RANK_VANILLA** will override this setting if defined.

APPEND_RANK_STANDARD Expression to be appended to standard job rank.

APPEND_RANK_VANILLA Expression to append to vanilla job rank.

NOTE: The **APPEND_RANK_STANDARD** and **APPEND_RANK_VANILLA** macros were called **APPEND_PREF_STANDARD** and **APPEND_PREF_VANILLA** in previous versions of Condor.

In addition, you may provide default Rank expressions if your users do not specify their own with:

DEFAULT_RANK Default rank expression for any job that does not specify its own rank expression in the submit description file. There is no default value, such that when undefined, the value used will be 0.0.

DEFAULT_RANK_VANILLA Default rank for vanilla universe jobs. There is no default value, such that when undefined, the value used will be 0.0. When both **DEFAULT_RANK** and **DEFAULT_RANK_VANILLA** are defined, the value for **DEFAULT_RANK_VANILLA** is used for vanilla universe jobs.

DEFAULT_RANK_STANDARD Default rank for standard universe jobs. There is no default value, such that when undefined, the value used will be 0.0. When both **DEFAULT_RANK** and **DEFAULT_RANK_STANDARD** are defined, the value for **DEFAULT_RANK_STANDARD** is used for standard universe jobs.

DEFAULT_IO_BUFFER_SIZE Condor keeps a buffer of recently-used data for each file an application opens. This macro specifies the default maximum number of bytes to be buffered for each open file at the executing machine. The *condor_status* *buffer_size* command will override this default. If this macro is undefined, a default size of 512 KB will be used.

DEFAULT_IO_BUFFER_BLOCK_SIZE When buffering is enabled, Condor will attempt to consolidate small read and write operations into large blocks. This macro specifies the default block size Condor will use. The *condor_status* *buffer_block_size* command will override this default. If this macro is undefined, a default size of 32 KB will be used.

SUBMIT_SKIP_FILECHECKS If True, *condor_submit* behaves as if the **-d** command-line option is used. This tells *condor_submit* to disable file permission checks when submitting a job. This can significantly decrease the amount of time required to submit a large group of jobs. The default value is False.

WARN_ON_UNUSED_SUBMIT_FILE_MACROS A boolean variable that defaults to True. When True, *condor_submit* performs checks on the job's submit description file contents for commands that define a macro, but do not use the macro within the file. A warning is issued, but job submission continues. A definition of a new macro occurs when the lhs of a command is not a known submit command. This check may help spot spelling errors of known submit commands.

SUBMIT_SEND_RESCHEDULE A boolean expression that when False, prevents *condor_submit* from automatically sending a *condor_reschedule* command as it completes. The *condor_reschedule* command causes the *condor_schedd* daemon to start searching for machines with which to match the submitted jobs. When True, this step always occurs. In the case that the machine where the job(s) are submitted is managing a huge number of jobs (thousands or tens of thousands), this step would hurt performance in such a way that it became an obstacle to scalability. The default value is True.

SUBMIT_EXPRS The given comma-separated, named expressions are inserted into all the job ClassAds that *condor_submit* creates. This is equivalent to the "+" syntax in submit files. See the *condor_submit* manual page on page 795 for details on using the "+" syntax to add attributes to the job ClassAd. Attributes defined in the submit description file with "+" will override attributes defined in the config file with **SUBMIT_EXPRS**.

LOG_ON_NFS_IS_ERROR A boolean value that controls whether *condor_submit* prohibits job submit files with user log files on NFS. If **LOG_ON_NFS_IS_ERROR** is set to True, such submit files will be rejected. If **LOG_ON_NFS_IS_ERROR** is set to False, the job will be submitted. If not defined, **LOG_ON_NFS_IS_ERROR** defaults to False.

SUBMIT_MAX_PROCS_IN_CLUSTER An integer value that limits the maximum number of jobs that would be assigned within a single cluster. Job submissions that would exceed the defined value fail, issuing an error message, and with no jobs submitted. The default value is 0, which does not limit the number of jobs assigned a single cluster number.

3.3.15 condor_preen Configuration File Entries

These macros affect *condor_preen*.

PREENADMIN This macro sets the e-mail address where *condor_preen* will send e-mail (if it is configured to send email at all; see the entry for **PREEN**). Defaults to \$(CONDOR_ADMIN).

VALID_SPOOL_FILES This macro contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the \$(SPOOL) directory. There is no default value. *condor_preen* will add to the list files and directories that are normally present in the \$(SPOOL) directory.

INVALID_LOG_FILES This macro contains a (comma or space separated) list of files that *condor_preen* considers invalid files to find in the \$(LOG) directory. There is no default value.

3.3.16 condor_collector Configuration File Entries

These macros affect the *condor_collector*.

CLASSADLIFETIME This macro determines the default maximum age for ClassAds collected by the *condor_collector*. ClassAd older than the maximum age are discarded by the *condor_collector* as stale.

If present, the ClassAd attribute “ClassAdLifetime” specifies the ad’s lifetime in seconds. If “ClassAdLifetime” is not present in the ad, the *condor_collector* will use the value of \$(CLASSADLIFETIME). The macro is defined in terms of seconds, and defaults to 900 (15 minutes).

MASTER_CHECK_INTERVAL This macro defines how often the collector should check for machines that have ClassAds from some daemons, but not from the *condor_master* (*orphaned daemons*) and send e-mail about it. It is defined in seconds and defaults to 10800 (3 hours).

COLLECTOR_REQUIREMENTS A boolean expression that filters out unwanted ClassAd updates. The expression is evaluated for ClassAd updates that have passed through enabled security authorization checks. The default behavior when this expression is not defined is to allow all ClassAd updates to take place. If `False`, a ClassAd update will be rejected.

Stronger security mechanisms are the better way to authorize or deny updates to the *condor_collector*. This configuration variable exists to help those that use host-based security, and do not trust all processes that run on the hosts in the pool. This configuration variable may be used to throw out ClassAds that should not be allowed. For example, for *condor_startd* daemons that run on a fixed port, configure this expression to ensure that only machine ClassAds advertising the expected fixed port are accepted. As a convenience, before evaluating the expression, some basic sanity checks are performed on the ClassAd to ensure that all of the ClassAd attributes used by Condor to contain IP:port information are consistent. To validate this information, the attribute to check is `TARGET.MyAddress`.

CLIENT_TIMEOUT Network timeout that the *condor_collector* uses when talking to any daemons or tools that are sending it a ClassAd update. It is defined in seconds and defaults to 30.

QUERY_TIMEOUT Network timeout when talking to anyone doing a query. It is defined in seconds and defaults to 60.

CONDOR_DEVELOPERS By default, Condor will send e-mail once per week to this address with the output of the *condor_status* command, which lists how many machines are in the pool and how many are running jobs. The default value of `condor-admin@cs.wisc.edu` will send this report to the Condor Team developers at the University of Wisconsin-Madison. The Condor Team uses these weekly status messages in order to have some idea as to how many Condor pools exist in the world. We appreciate getting the reports, as this is one way we can convince funding agencies that Condor is being used in the real world. If you do not wish this

information to be sent to the Condor Team, explicitly set the value to `NONE` to disable this feature, or replace the address with a desired location. If undefined (commented out) in the configuration file, Condor follows its default behavior.

COLLECTORNAME This macro is used to specify a short description of your pool. It should be about 20 characters long. For example, the name of the UW-Madison Computer Science Condor Pool is `"UW-Madison CS"`. While this macro might seem similar to `MASTER_NAME` or `SCHEDD_NAME`, it is unrelated. Those settings are used to uniquely identify (and locate) a specific set of Condor daemons, if there are more than one running on the same machine. The `COLLECTOR_NAME` setting is just used as a human-readable string to describe the pool, which is included in the updates sent to the `CONDOR_DEVELOPERS_COLLECTOR` (see below).

CONDOR_DEVELOPERS_COLLECTOR By default, every pool sends periodic updates to a central *condor_collector* at UW-Madison with basic information about the status of your pool. This includes only the number of total machines, the number of jobs submitted, the number of machines running jobs, the host name of your central manager, and the `$(COLLECTOR_NAME)` specified above. These updates help the Condor Team see how Condor is being used around the world. By default, they will be sent to `condor.cs.wisc.edu`. If you do not want these updates to be sent from your pool, explicitly set this macro to `NONE`. If undefined (commented out) in the configuration file, Condor follows its default behavior.

COLLECTOR_SOCKET_BUFSIZE This specifies the buffer size, in bytes, reserved for *condor_collector* network UDP sockets. The default is 10240000, or a ten megabyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 256000 or 128000).

NOTE: For some Linux distributions, it may be necessary to raise the OS's system-wide limit for network buffer sizes. The parameter that controls this limit is `/proc/sys/net/core/rmem_max`. You can see the values that the *condor_collector* actually uses by enabling `D_FULLDEBUG` for the collector and looking at the log line that looks like this:
Reset OS socket buffer size to 2048k (UDP), 255k (TCP).

For changes to this parameter to take effect, *condor_collector* must be restarted.

COLLECTOR_TCP_SOCKET_BUFSIZE This specifies the TCP buffer size, in bytes, reserved for *condor_collector* network sockets. The default is 131072, or a 128 kilobyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 65536 or 32768).

NOTE: See the note for `COLLECTOR_SOCKET_BUFSIZE`.

COLLECTOR_SOCKET_CACHE_SIZE If your site wants to use TCP connections to send ClassAd updates to the collector, you must use this setting to enable a cache of TCP sockets (in addition to enabling `UPDATE_COLLECTOR_WITH_TCP`). Please read section 3.7.4 on "Using TCP to Send Collector Updates" on page 345 for more details and a discussion of when you would

need this functionality. If you do not enable a socket cache, TCP updates will be refused by the collector. The default value for this setting is 0, with no cache enabled. If you lower this number, you must run *condor_restart* and not just *condor_reconfig* for the change to take effect.

KEEP_POOL_HISTORY This boolean macro is used to decide if the collector will write out statistical information about the pool to history files. The default is `False`. The location, size, and frequency of history logging is controlled by the other macros.

POOL_HISTORY_DIR This macro sets the name of the directory where the history files reside (if history logging is enabled). The default is the `SPOOL` directory.

POOL_HISTORY_MAX_STORAGE This macro sets the maximum combined size of the history files. When the size of the history files is close to this limit, the oldest information will be discarded. Thus, the larger this parameter's value is, the larger the time range for which history will be available. The default value is 10000000 (10 Mbytes).

POOL_HISTORY_SAMPLING_INTERVAL This macro sets the interval, in seconds, between samples for history logging purposes. When a sample is taken, the collector goes through the information it holds, and summarizes it. The information is written to the history file once for each 4 samples. The default (and recommended) value is 60 seconds. Setting this macro's value too low will increase the load on the collector, while setting it to high will produce less precise statistical information.

COLLECTOR_DAEMON_STATS This macro controls whether or not the Collector keeps update statistics on incoming updates. The default value is `True`. If this option is enabled, the collector will insert several attributes into ClassAds that it stores and sends. ClassAds without the "UpdateSequenceNumber" and "DaemonStartTime" attributes will not be counted, and will not have attributes inserted (all modern Condor daemons which publish ClassAds publish these attributes).

The attributes inserted are "UpdatesTotal", "UpdatesSequenced", and "UpdatesLost". "UpdatesTotal" is the total number of updates (of this ad type) the Collector has received from this host. "UpdatesSequenced" is the number of updates that the Collector could have as lost. In particular, for the first update from a daemon it is impossible to tell if any previous ones have been lost or not. "UpdatesLost" is the number of updates that the Collector has detected as being lost. See page 894 for more information on the added attributes.

COLLECTOR_STATS_SWEEP This value specifies the number of seconds between sweeps of the *condor_collector*'s per-daemon update statistics. Records for daemons which have not reported in this amount of time are purged in order to save memory. The default is two days. It is unlikely that you would ever need to adjust this.

COLLECTOR_DAEMON_HISTORY_SIZE This macro controls the size of the published update history that the Collector inserts into the ClassAds it stores and sends. The default value is 128, which means that history is stored and published for the latest 128 updates. This macro is ignored if `$(COLLECTOR_DAEMON_STATS)` is not enabled.

If this has a non-zero value, the Collector will insert "UpdatesHistory" into the ClassAd (similar to "UpdatesTotal" above). "UpdatesHistory" is a hexadecimal string which represents a

bitmap of the last `COLLECTOR_DAEMON_HISTORY_SIZE` updates. The most significant bit (MSB) of the bitmap represents the most recent update, and the least significant bit (LSB) represents the least recent. A value of zero means that the update was not lost, and a value of 1 indicates that the update was detected as lost.

For example, if the last update was not lost, the previous lost, and the previous two not, the bitmap would be 0100, and the matching hex digit would be “4”. Note that the MSB can never be marked as lost because its loss can only be detected by a non-lost update (a “gap” is found in the sequence numbers). Thus, `UpdatesHistory = "0x40"` would be the history for the last 8 updates. If the next updates are all successful, the values published, after each update, would be: 0x20, 0x10, 0x08, 0x04, 0x02, 0x01, 0x00.

See page 894 for more information on the added attribute.

COLLECTORCLASS_HISTORY_SIZE This macro controls the size of the published update history that the Collector inserts into the Collector ClassAds it produces. The default value is zero.

If this has a non-zero value, the Collector will insert “UpdatesClassHistory” into the Collector ClassAd (similar to “UpdatesHistory” above). These are added “per class” of ClassAd, however. The classes refer to the “type” of ClassAds (i.e. “Start”). Additionally, there is a “Total” class created which represents the history of all ClassAds that this Collector receives.

Note that the collector always publishes Lost, Total and Sequenced counts for all ClassAd “classes”. This is similar to the statistics gathered if `$(COLLECTOR_DAEMON_STATS)` is enabled.

COLLECTORQUERY_WORKERS This macro sets the maximum number of “worker” processes that the Collector can have. When receiving a query request, the UNIX Collector will “fork” a new process to handle the query, freeing the main process to handle other requests. When the number of outstanding “worker” processes reaches this maximum, the request is handled by the main process. This macro is ignored on Windows, and its default value is zero. The default configuration, however, has this set to 16.

COLLECTORDEBUG This macro (and other macros related to debug logging in the collector) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

3.3.17 condor_negotiator Configuration File Entries

These macros affect the *condor_negotiator*.

NEGOTIATOR_INTERVAL Sets how often the negotiator starts a negotiation cycle. It is defined in seconds and defaults to 300 (5 minutes).

NEGOTIATOR_CYCLE_DELAY An integer value that represents the minimum number of seconds that must pass before a new negotiation cycle may start. The default value is 20. `NEGOTIATOR_CYCLE_DELAY` is intended only for use by Condor experts.

NEGOTIATOR_TIMEOUT Sets the timeout that the negotiator uses on its network connections to the *condor_schedd* and *condor_startds*. It is defined in seconds and defaults to 30.

PRIORITY_HALFLIFE This macro defines the half-life of the user priorities. See section 2.7.2 on User Priorities for details. It is defined in seconds and defaults to 86400 (1 day).

DEFAULT_PRIO_FACTOR This macro sets the priority factor for local users. See section 2.7.2 on User Priorities for details. Defaults to 1.

NICE_USER_PRIO_FACTOR This macro sets the priority factor for nice users. See section 2.7.2 on User Priorities for details. Defaults to 1000000.

REMOTE_PRIO_FACTOR This macro defines the priority factor for remote users (users who do not belong to the accountant's local domain - see below). See section 2.7.2 on User Priorities for details. Defaults to 10000.

ACCOUNTANT_LOCAL_DOMAIN This macro is used to decide if a user is local or remote. A user is considered to be in the local domain if the `UID_DOMAIN` matches the value of this macro. Usually, this macro is set to the local `UID_DOMAIN`. If it is not defined, all users are considered local.

MAX_ACCOUNTANT_DATABASE_SIZE This macro defines the maximum size (in bytes) that the accountant database log file can reach before it is truncated (which re-writes the file in a more compact format). If, after truncating, the file is larger than one half the maximum size specified with this macro, the maximum size will be automatically expanded. The default is 1 megabyte (1000000).

NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES This macro tells the negotiator to not count resources that are suspended when calculating the number of resources a user is using. Defaults to false, that is, a user is still charged for a resource even when that resource has suspended the job.

NEGOTIATOR_SOCKET_CACHE_SIZE This macro defines the maximum number of sockets that the *condor_negotiator* keeps in its open socket cache. Caching open sockets makes the negotiation protocol more efficient by eliminating the need for socket connection establishment for each negotiation cycle. The default is currently 16. To be effective, this parameter should be set to a value greater than the number of *condor_schedds* submitting jobs to the negotiator at any time. If you lower this number, you must run *condor_restart* and not just *condor_reconfig* for the change to take effect.

NEGOTIATOR_INFORM_STARTD Boolean setting that controls if the *condor_negotiator* should inform the *condor_startd* when it has been matched with a job. The default is `True`. When this is set to `False`, the *condor_startd* will never enter the Matched state, and will go directly from Unclaimed to Claimed. Because this notification is done via UDP, if a pool is configured so that the execute hosts do not create UDP command sockets (see the `WANT_UDP_COMMAND_SOCKET` setting described in section 3.3.3 on page 157 for details), the *condor_negotiator* should be configured not to attempt to contact these *condor_startds* by configuring this setting to `False`.

NEGOTIATOR_PRE_JOB_RANK Resources that match a request are first sorted by this expression. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by `NEGOTIATOR_POST_JOB_RANK`, then by `PREEMPTION_RANK` (if the match would cause preemption and there are still any ties in the top choice). `MY` refers to attributes of the machine ClassAd and `TARGET` refers to the job ClassAd. The purpose of the pre job rank is to allow the pool administrator to override any other rankings, in order to optimize overall throughput. For example, it is commonly used to minimize preemption, even if the job rank prefers a machine that is busy. If undefined, this expression has no effect on the ranking of matches. The standard configuration file shipped with Condor specifies an expression to steer jobs away from busy resources:

```
NEGOTIATOR_PRE_JOB_RANK = RemoteOwner != UNDEFINED
```

NEGOTIATOR_POST_JOB_RANK Resources that match a request are first sorted by `NEGOTIATOR_PRE_JOB_RANK`. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by `NEGOTIATOR_POST_JOB_RANK`, then by `PREEMPTION_RANK` (if the match would cause preemption and there are still any ties in the top choice). `MY` refers to attributes of the machine ClassAd and `TARGET` refers to the job ClassAd. The purpose of the post job rank is to allow the pool administrator to choose between machines that the job ranks equally. The default value is undefined, which causes this rank to have no effect on the ranking of matches. The following example expression steers jobs toward faster machines and tends to fill a cluster of multi-processors by spreading across all machines before filling up individual machines. In this example, the expression is chosen to have no effect when preemption would take place, allowing control to pass on to `PREEMPTION_RANK`.

```
UWCS_NEGOTIATOR_POST_JOB_RANK = \
  (RemoteOwner != UNDEFINED) * (KFlops - VirtualMachineID)
```

PREEMPTION_REQUIREMENTS When considering user priorities, the negotiator will not preempt a job running on a given machine unless the `PREEMPTION_REQUIREMENTS` expression evaluates to `True` and the owner of the idle job has a better priority than the owner of the running job. The `PREEMPTION_REQUIREMENTS` expression is evaluated within the context of the candidate machine ClassAd and the candidate idle job ClassAd; thus the `MY` scope prefix refers to the machine ClassAd, and the `TARGET` scope prefix refers to the ClassAd of the idle (candidate) job. There is no direct access to the currently running job, but attributes of the currently running job that need to be accessed in `PREEMPTION_REQUIREMENTS` can be placed in the machine ClassAd using `STARTD_JOB_EXPRS`. If not explicitly set in the Condor configuration file, the default value for this expression is `True`. Note that this setting does not influence other potential causes of preemption, such as `startd RANK`, or `PREEMPT` expressions. See section 3.5.9 for a general discussion of limiting preemption.

PREEMPTION_REQUIREMENTS_STABLE A boolean value that defaults to `True`, implying that all attributes utilized to define the `PREEMPTION_REQUIREMENTS` variable will not change within a negotiation period time interval. If utilized attributes will change during the negotiation period time interval, then set this variable to `False`.

PREEMPTIONRANK Resources that match a request are first sorted by `NEGOTIATOR_PRE_JOB_RANK`. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by `NEGOTIATOR_POST_JOB_RANK`, then by `PREEMPTION_RANK` (if the match would cause preemption and there are still any ties in the top choice). `MY` refers to attributes of the machine ClassAd and `TARGET` refers to the job ClassAd. This expression is used to rank machines that the job and the other negotiation expressions rank the same. For example, if the job has no preference, it is usually preferable to preempt a job with a small `ImageSize` instead of a job with a large `ImageSize`. The default is to rank all preemptable matches the same. However, the negotiator will always prefer to match the job with an idle machine over a preemptable machine, if none of the other ranks express a preference between them.

PREEMPTIONRANK_STABLE A boolean value that defaults to `True`, implying that all attributes utilized to define the `PREEMPTION_RANK` variable will not change within a negotiation period time interval. If utilized attributes will change during the negotiation period time interval, then set this variable to `False`.

NEGOTIATOR_DEBUG This macro (and other settings related to debug logging in the negotiator) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

NEGOTIATOR_MAX_TIME_PER_SUBMITTER The maximum number of seconds the *condor_negotiator* will spend with a submitter during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will still finish its current pie spin, but it will skip over the submitter if subsequent pie spins are needed to dish out all of the available machines. It defaults to one year. See `NEGOTIATOR_MAX_TIME_PER_PIESPIN` for more information.

NEGOTIATOR_MAX_TIME_PER_PIESPIN The maximum number of seconds the *condor_negotiator* will spend with a submitter in one pie spin. A negotiation cycle is composed of at least one pie spin, possibly more, depending on whether there are still machines left over after computing fair shares and negotiating with each submitter. By limiting the maximum length of a pie spin or the maximum time per submitter per negotiation cycle, the *condor_negotiator* is protected against spending a long time talking to one submitter, for example someone with a very slow *condor_schedd* daemon. But, this can result in unfair allocation of machines or some machines not being allocated at all. See section 3.4.6 on page 248 for a description of a pie slice.

NEGOTIATOR_MATCH_EXPRS This macro specifies a list of macro names that are inserted as ClassAd attributes into matched job ClassAds. The attribute name in the ClassAd will be given the prefix `NegotiatorMatchExpr` if the macro name doesn't already begin with that. Example:

```
NegotiatorName = "My Negotiator"
NEGOTIATOR_MATCH_EXPRS = NegotiatorName
```

As a result of the above configuration, jobs that are matched by this negotiator will contain the following attribute when they are sent to the startd:

```
NegotiatorMatchExprNegotiatorName = "My Negotiator"
```

The expressions inserted by the negotiator may be useful in startd policy expressions when the startd belongs to multiple Condor pools.

The following configuration macros affect negotiation for group users.

GROUP_NAMES A comma-separated list of the recognized group names, case insensitive. If undefined (the default), group support is disabled. Group names must not conflict with any user names. That is, if there is a `physics` group, there may not be a `physics` user. Any group that is defined here must also have a quota, or the group will be ignored. Example:

```
GROUP_NAMES = group_physics, group_chemistry
```

GROUP_QUOTA<groupname> A positive integer to represent a static quota specifying the exact number of machines owned by this group. Note that Condor does not verify or check consistency of quota values. Example:

```
GROUP_QUOTA_group_physics = 20
GROUP_QUOTA_group_chemistry = 10
```

GROUP_PRIO_FACTOR<groupname> A floating point value greater than or equal to 1.0 to specify the default user priority factor for <groupname>. The group name must also be specified in the `GROUP_NAMES` list. `GROUP_PRIO_FACTOR.<groupname>` is evaluated when the negotiator first negotiates for the user as a member of the group. All members of the group inherit the default priority factor when no other value is present. For example, the following setting specifies that all members of the group named `group_physics` inherit a default user priority factor of 2.0:

```
GROUP_PRIO_FACTOR_group_physics = 2.0
```

GROUP_AUTOREGROUP A boolean value (defaults to `False`) that when `True`, causes users who submitted to a specific group to also negotiate a second time with the `none` group, to be considered with the independent job submitters. This allows group submitted jobs to be matched with idle machines even if the group is over its quota.

GROUP_AUTOREGROUP.<groupname> This is the same as `GROUP_AUTOREGROUP`, but it is settable on a per-group basis. If no value is specified for a given group, the default behavior is determined by `GROUP_AUTOREGROUP`, which in turn defaults to `False`.

NEGOTIATORCONSIDERPREEMPTION For expert users only. A boolean value (defaults to `True`), that when `False`, can cause the negotiator to run faster and also have better spinning pie accuracy. *Only set this to `False` if `PREEMPTION_REQUIREMENTS` is `False`, and if all `condor_startd` rank expressions are `False`.*

3.3.18 condor_procdd Configuration File Macros

USE_PROCDD This boolean parameter is used to determine whether the *condor_procdd* will be used for managing process families. If the *condor_procdd* is not used, each daemon will run the process family tracking logic on its own. Use of the *condor_procdd* results in improved scalability because only one instance of this logic is required. The *condor_procdd* is required when using privilege separation (see Section 3.6.12) or group ID-based process tracking (see Section 3.12.10). In either of these cases, the `USE_PROCDD` setting will be ignored and a *condor_procdd* will always be used. By default, the *condor_master* will not use a *condor_procdd* but all other daemons that need process family tracking will. A daemon that uses the *condor_procdd* will start a *condor_procdd* for use by itself and all of its child daemons.

PROCDD_MAX_SNAPSHOT_INTERVAL This setting determines the maximum time that the *condor_procdd* will wait between probes of the system for information about the process families it is tracking.

PROCDD_LOG Specifies a log file for the ProcD to use. Note that by design, the *condor_procdd* does not include most of the other logic that is shared amongst the various Condor daemons. This is because the *condor_procdd* is a component of the PrivSep Kernel (see Section 3.6.12 for more information regarding privilege separation). This means that the *condor_procdd* does not include the normal Condor logging subsystem, and thus things like multiple debug levels and log rotation are not supported. Therefore, `PROCDD_LOG` is not set by default and is only intended to debug problems should they arise. Note, however, that enabling `D_PROCFAMILY` in the debug level for any other daemon will cause it to log all interactions with the *condor_procdd*.

PROCDD_ADDRESS This specifies the address that the *condor_procdd* will use to receive requests from other Condor daemons. On Unix, this should point to a file system location that can be used for a named pipe. On Windows, named pipes are also used but they do not exist in the file system. The default setting therefore depends on the platform: `$(LOCK)/procdd_pipe` on Unix and `\\.\pipe\procdd_pipe` on Windows.

3.3.19 condor_creddd Configuration File Macros

These macros affect the *condor_creddd*.

CREDD_HOST The host name of the machine running the *condor_creddd* daemon.

CREDD_CACHE_LOCALLY A boolean value that defaults to `False`. When `True`, the first successful password fetch operation to the *condor_creddd* daemon causes the password to be stashed in a local, secure password store. Subsequent uses of that password do not require communication with the *condor_creddd* daemon.

3.3.20 condor_gridmanager Configuration File Entries

These macros affect the *condor_gridmanager*.

GRIDMANAGERLOG Defines the path and file name for the log of the *condor_gridmanager*. The owner of the file is the *condor* user.

GRIDMANAGERCHECKPROXYINTERVAL The number of seconds between checks for an updated X509 proxy credential. The default is 10 minutes (600 seconds).

GRIDMANAGERMINIMUMPROXYTIME The minimum number of seconds before expiration of the X509 proxy credential for the *gridmanager* to continue operation. If seconds until expiration is less than this number, the *gridmanager* will shutdown and wait for a refreshed proxy credential. The default is 3 minutes (180 seconds).

HOLD_JOB_IF_CREDENTIAL_EXPIRES True or False. Defaults to True. If True, and for grid universe jobs only, Condor-G will place a job on hold *GRIDMANAGER_MINIMUM_PROXY_TIME* seconds before the proxy expires. If False, the job will stay in the last known state, and Condor-G will periodically check to see if the job's proxy has been refreshed, at which point management of the job will resume.

GRIDMANAGERCONTACT_SCHEDD_DELAY The minimum number of seconds between connections to the *condor_schedd*. The default is 5 seconds.

GRIDMANAGERJOB_PROBE_INTERVAL The number of seconds between active probes of the status of a submitted job. The default is 5 minutes (300 seconds).

CONDOR_JOB_POLL_INTERVAL After a condor grid type job is submitted, how often (in seconds) the *condor_gridmanager* should probe the remote *condor_schedd* to check the jobs status. This defaults to 300 seconds (5 minutes). Setting this to a lower number will decrease latency (Condor will discover that a job has finished more quickly), but will increase network traffic.

GRIDMANAGERRESOURCEPROBE_INTERVAL When a resource appears to be down, how often (in seconds) the *condor_gridmanager* should ping it to test if it is up again.

GRIDMANAGERRESOURCEPROBE_DELAY The number of seconds between pings of a remote resource that is currently down. The default is 5 minutes (300 seconds).

GRIDMANAGEREMPTY_RESOURCE_DELAY The number of seconds that the *condor_gridmanager* retains information about a grid resource, once the *condor_gridmanager* has no active jobs on that resource. An active job is a grid universe job that is in the queue, but is not in the HELD state. Defaults to 300 seconds.

GRIDMANAGERMAX_SUBMITTED_JOBS_PER_RESOURCE Limits the number of jobs that a *condor_gridmanager* daemon will submit to a resource. It is useful for controlling the number of *jobmanager* processes running on the front-end node of a cluster. This number may be exceeded if it is reduced through the use of *condor_reconfig* while the *condor_gridmanager* is running or if the *condor_gridmanager* receives new jobs from the *condor_schedd* that were already submitted (that is, their *GridJobId* is not undefined). In these cases, submitted jobs will not be killed, but no new jobs can be submitted until the number of submitted jobs falls below the current limit. Defaults to 100.

GRIDMANAGERMAX_PENDING_SUBMITS_PER_RESOURCE The maximum number of jobs that can be in the process of being submitted at any time (that is, how many `globus_gram_client_job_request()` calls are pending). It is useful for controlling the number of new connections/processes created at a given time. The default value is 5. This variable allows you to set different limits for each resource. After the first integer in the value comes a list of resourcename/number pairs, where each number is the limit for that resource. If a resource is not in the list, Condor uses the first integer. An example usage:

```
GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE=20,nostos,5,beak,50
```

GRIDMANAGERMAX_PENDING_SUBMITS Configuration variable still recognized, but the name has changed to be `GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE`.

GRIDMANAGERMAX_JOBMANAGERS_PER_RESOURCE For grid jobs of type **gt2**, limits the number of globus-job-manager processes that the *condor_gridmanager* lets run at a time on the remote head node. Allowing too many globus-job-managers to run causes severe load on the headnode, possibly making it non-functional. This number may be exceeded if it is reduced through the use of *condor_reconfig* while the *condor_gridmanager* is running or if some globus-job-managers take a few extra seconds to exit. The value 0 means there is no limit. The default value is 10.

GRIDMANAGERMAX_WS_DESTROY_PER_RESOURCE For grid jobs of type **gt4**, limits the number of destroy commands that the *condor_gridmanager* will issue at a time to each WS GRAM server. Too many destroy commands can have severe effects on the server. The default value is 5.

GAHP The full path to the binary of the GAHP server. This configuration variable is no longer used. Use `GT2_GAHP` at section 3.3.20 instead.

GAHP_ARGS Arguments to be passed to the GAHP server. This configuration variable is no longer used.

GRIDMANAGERGAHP_CALL_TIMEOUT The number of seconds after which a pending GAHP command should time out. The default is 5 minutes (300 seconds).

GRIDMANAGERMAX_PENDING_REQUESTS The maximum number of GAHP commands that can be pending at any time. The default is 50.

GRIDMANAGERCONNECT_FAILURE_RETRY_COUNT The number of times to retry a command that failed due to a timeout or a failed connection. The default is 3.

GRIDMANAGERGLOBUS_COMMIT_TIMEOUT The duration, in seconds, of the two phase commit timeout to Globus for **gt2** jobs only. This maps directly to the `two_phase` setting in the Globus RSL.

GLOBUS_GATEKEEPER_TIMEOUT The number of seconds after which if a **gt2** grid universe job fails to ping the gatekeeper, the job will be put on hold. Defaults to 5 days (in seconds).

GRIDFTP_URL_BASE Specifies an existing *GridFTP* server on the local system to be used for file transfers for gt4 grid universe jobs. The value is given as the base of a URL, such as `gsiftp://mycomp.foo.edu:2118`. The default is for Condor to launch temporary *GridFTP* servers as needed for file transfer.

C_GAHP_LOG The complete path and file name of the Condor GAHP server's log. There is no default value. The expected location as defined in the example configuration is `/temp/CGAHPLog. $(USERNAME)`.

MAX_C_GAHP_LOG The maximum size of the `C_GAHP_LOG`.

C_GAHP_WORKER_THREAD_LOG The complete path and file name of the Condor GAHP worker process' log. There is no default value. The expected location as defined in the example configuration is `/temp/CGAHPWorkerLog. $(USERNAME)`.

GLITE_LOCATION The complete path to the directory containing the Glite software. There is no default value. The expected location as given in the example configuration is `$(LIB)/glite`. The necessary Glite software is included with Condor, and is required for pbs and lsf jobs.

AMAZON_EC2_URL The URL Condor should use when contacting the Amazon EC2 service. The default value is `https://ec2.amazonaws.com/`.

AMAZON_HTTP_PROXY The http proxy that Condor should use when contacting the Amazon EC2 service. The default is to not use a proxy.

CONDOR_GAHP The complete path and file name of the Condor GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/condor_c-gahp`.

AMAZON_GAHP The complete path and file name of the Amazon GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/amazon-gahp`.

GT2_GAHP The complete path and file name of the GT2 GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/gahp_server`.

GT4_GAHP The complete path and file name of the wrapper script that invokes the GT4 GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/gt4_gahp`.

PBS_GAHP The complete path and file name of the PBS GAHP executable. There is no default value. The expected location as given in the example configuration is `$(GLITE_LOCATION)/bin/batch_gahp`.

LSF_GAHP The complete path and file name of the LSF GAHP executable. There is no default value. The expected location as given in the example configuration is `$(GLITE_LOCATION)/bin/batch_gahp`.

UNICORE_GAHP The complete path and file name of the wrapper script that invokes the Unicore GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/unicore_gahp`.

NORDUGRID_GAHP The complete path and file name of the wrapper script that invokes the NorduGrid GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/nordugrid_gahp`.

3.3.21 condor_job_router Configuration File Entries

These macros affect the *condor_job_router* daemon.

JOB_ROUTER_DEFAULTS Defined by a single ClassAd in New ClassAd syntax, used to provide default values for all routes in the *condor_job_router* daemon's routing table. Where an attribute is set outside of these defaults, that attribute value takes precedence.

JOB_ROUTER_ENTRIES Specification of the job routing table. It is a list of ClassAds, in New ClassAd syntax, where each individual ClassAd is surrounded by square brackets, and the ClassAds are separated from each other by spaces. Each ClassAd describes one entry in the routing table, and each describes a site that jobs may be routed to.

A *condor_reconfig* command causes the *condor_job_router* daemon to rebuild the routing table. Routes are distinguished by a routing table entry's ClassAd attribute Name. Therefore, a Name change in an existing route has the potential to cause the inaccurate reporting of routes.

Instead of setting job routes using this configuration variable, they may be read from an external source using the `JOB_ROUTER_ENTRIES_FILE` or be dynamically generated by an external program via the `JOB_ROUTER_ENTRIES_CMD` configuration variable.

JOB_ROUTER_ENTRIES_FILE A path and file name of a file that contains the ClassAds, in New ClassAd syntax, describing the routing table. The specified file is periodically reread to check for new information. This occurs every `$(JOB_ROUTER_ENTRIES_REFRESH)` seconds.

JOB_ROUTER_ENTRIES_CMD Specifies the command line of an external program to run. The output of the program defines or updates the routing table, and the output must be given in New ClassAd syntax. The specified command is periodically rerun to regenerate or update the routing table. This occurs every `$(JOB_ROUTER_ENTRIES_REFRESH)` seconds. Specify the full path and file name of the executable within this command line, as no assumptions may be made about the current working directory upon command invocation. To enter spaces in any command-line arguments or in the command name itself, surround the right hand side of this definition with double quotes, and use single quotes around individual arguments that contain spaces. This is the same as when dealing with spaces within job arguments in a Condor submit description file.

JOB_ROUTER_ENTRIES_REFRESH The number of seconds between updates to the routing table described by `JOB_ROUTER_ENTRIES_FILE` or `JOB_ROUTER_ENTRIES_CMD`. The default value is 0, meaning no periodic updates occur. With the default value of 0, the routing table

can be modified when a *condor_reconfig* command is invoked or when the *condor_job_router* daemon restarts.

JOB_ROUTER_SOURCE_JOB_CONSTRAINT Specifies a global `Requirements` expression that will be appended to all routed jobs, in addition to any `Requirements` specified within a routing table entry.

JOB_ROUTER_MAX_JOBS An integer value representing the maximum number of jobs that may be routed, summed over all routes. The default value is -1, which means an unlimited number of jobs may be routed.

MAX_JOB_MIRROR_UPDATE_LAG An integer value that administrators will rarely consider changing, representing the maximum number of seconds the *condor_job_router* daemon waits, before it decides that routed copies have gone awry, due to the failure of events to appear in the *condor_schedd*'s job queue log file. The default value is 600. As the *condor_job_router* daemon uses the *condor_schedd*'s job queue log file entries for synchronization of routed copies, when an expected log file event fails to appear after this wait period, the *condor_job_router* daemon acts presuming the expected event will never occur.

JOB_ROUTER_POLLING_PERIOD An integer value representing the number of seconds between cycles in the *condor_job_router* daemon's task loop. The default is 10 seconds. A small value makes the *condor_job_router* daemon quick to see new candidate jobs for routing. A large value makes the *condor_job_router* daemon generate less overhead at the cost of being slower to see new candidates for routing. For very large job queues where a few minutes of routing latency is no problem, increasing this value to a few hundred seconds would be reasonable.

JOB_ROUTER_NAME A unique identifier utilized to name multiple instances of the *condor_job_router* daemon within a single Condor pool. Each instance must have a different name, or all but the first to start up will refuse to run.

3.3.22 condor_lease_manager Configuration File Entries

These macros affect the *condor_lease_manager*.

The *condor_lease_manager* expects to use the syntax

```
<subsystem name>.<parameter name>
```

in configuration. This allows multiple instances of the *condor_lease_manager* to be easily configured using the syntax

```
<subsystem name>.<local name>.<parameter name>
```

LeaseManager.GETADSINTERVAL An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* pulls relevant resource `ClassAds` from the *condor_collector*. The default value is 60 seconds, with a minimum value of 2 seconds.

LeaseManager.UPDATEINTERVAL An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* sends its ClassAds to the *condor_collector*. The default value is 60 seconds, with a minimum value of 5 seconds.

LeaseManager.PRUNEINTERVAL An integer value, given in seconds, that controls the frequency with which the *condor_lease_manager* prunes its leases. This involves checking all leases to see if they have expired. The default value is 60 seconds, with no minimum value.

LeaseManager.DEBUGADS A boolean value that defaults to False. When True, it enables extra debugging information about the resource ClassAds that it retrieves from the *condor_collector* and about the search ClassAds that it sends to the *condor_collector*.

LeaseManager.MAXLEASEDURATION An integer value representing seconds which determines the maximum duration of a lease. This can be used to provide a hard limit on lease durations. Normally, the *condor_lease_manager* honors the `MaxLeaseDuration` attribute from the resource ClassAd. If this configuration variable is defined, it limits the effective maximum duration for all resources to this value. The default value is 1800 seconds.

Note that leases can be renewed, and thus can be extended beyond this limit. To provide a limit on the total duration of a lease, use `LeaseManager.MAXTOTALLEASEDURATION`.

LeaseManager.MAXTOTALLEASEDURATION An integer value representing seconds used to limit the *total* duration of leases, over all its renewals. The default value is 3600 seconds.

LeaseManager.DEFAULTMAXLEASEDURATION The *condor_lease_manager* uses the `MaxLeaseDuration` attribute from the resource ClassAd to limit the lease duration. If this attribute is not present in a resource ClassAd, then this configuration variable is used instead. This integer value is given in units of seconds, with a default value of 60 seconds.

LeaseManager.CLASSADLOG This variable defines a full path and file name to the location where the *condor_lease_manager* keeps persistent state information. This variable has no default value.

LeaseManager.QUERYADTYPE This parameter controls the type of the query in the ClassAd sent to the *condor_collector*, which will control the types of ClassAds returned by the *condor_collector*. This parameter must be a valid ClassAd type name, with a default value of "Any".

LeaseManager.QUERYCONSTRAINTS A ClassAd expression that controls the constraint in the query sent to the *condor_collector*. It is used to further constrain the types of ClassAds from the *condor_collector*. There is no default value, resulting in no constraints being placed on query.

3.3.23 grid_monitor Configuration File Entries

These macros affect the *grid_monitor*.

ENABLE_GRID_MONITOR When set to `True` enables the *grid_monitor* tool. The *grid_monitor* tool is used to reduce load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. `GRID_MONITOR` must also be correctly configured. Defaults to `False`. See section 5.3.2 on page 508 for more information.

GRID_MONITOR The complete path name of the *grid_monitor* tool used to reduce load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. This parameter is not referenced unless `ENABLE_GRID_MONITOR` is set to `True`. See section 5.3.2 on page 508 for more information.

GRID_MONITOR_HEARTBEAT_TIMEOUT If this many seconds pass without hearing from a *grid_monitor*, it is assumed to be dead. Defaults to 300 (5 minutes). Increasing this number will improve the ability of the *grid_monitor* to survive in the face of transient problems but will also increase the time before Condor notices a problem.

GRID_MONITOR_RETRY_DURATION If something goes wrong with the *grid_monitor* at a particular site (like `GRID_MONITOR_HEARTBEAT_TIMEOUT` expiring), Condor-G will attempt to restart the *grid_monitor* for this many seconds. Defaults to 900 (15 minutes). If this duration passes without success the *grid_monitor* will be disabled for the site in question until 60 minutes have passed.

GRID_MONITOR_NO_STATUS_TIMEOUT Jobs can disappear from the *grid_monitor*'s status reports for short periods of time under normal circumstances, but a prolonged absence is often a sign of problems on the remote machine. This parameter sets the amount of time (in seconds) that a job can be absent before the *condor_gridmanager* reacts by restarting the GRAM *jobmanager*. The default is 15 minutes.

3.3.24 Configuration File Entries Relating to Grid Usage and Glidein

These macros affect the Condor's usage of grid resources and glidein.

GLIDEIN_SERVER_URLS A comma or space-separated list of URLs that contain the binaries that must be copied by *condor_glidein*. There are no default values, but working URLs that copy from the UW site are provided in the distributed sample configuration files.

GLExec_JOB A boolean value that defaults to `False`. When `True`, it enables the use of *glexec* on the machine.

GLExec The full path and file name of the *glexec* executable.

3.3.25 Configuration File Entries for DAGMan

These macros affect the operation of DAGMan and DAGMan jobs within Condor.

DAGMAN_DEBUG_CACHE_ENABLE A boolean value that determines if log line caching for the *dagman.out* file should be enabled in the *condor_dagman* process to increase performance (potentially by orders of magnitude) when writing the *dagman.out* file to an NFS server. Currently, this cache is only utilized in Recovery Mode. If not defined, it defaults to `False`.

DAGMAN_DEBUG_CACHE_SIZE An integer value in bytes which controls how many bytes of log lines are to be stored in the log line cache. When the cache surpasses this number the entries are written out in one call to the logging subsystem. A value of zero is not recommended since each log line would surpass the cache size and be emitted in addition to bracketing log lines explaining that the flushing was happening. The legal range of values is 0 to `INT_MAX`. If defined with a value less than 0, the value 0 will be used. If not defined, it defaults to 5 Megabytes.

DAGMAN_MAX_SUBMITS_PER_INTERVAL An integer that controls how many individual jobs *condor_dagman* will submit in a row before servicing other requests (such as a *condor_rm*). The legal range of values is 1 to 1000. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 1000, the value 1000 will be used. If not defined, it defaults to 5.

DAGMAN_MAX_SUBMIT_ATTEMPTS An integer that controls how many times in a row *condor_dagman* will attempt to execute *condor_submit* for a given job before giving up. Note that consecutive attempts use an exponential backoff, starting with 1 second. The legal range of values is 1 to 16. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 16, the value 16 will be used. Note that a value of 16 would result in *condor_dagman* trying for approximately 36 hours before giving up. If not defined, it defaults to 6 (approximately two minutes before giving up).

DAGMAN_SUBMIT_DELAY An integer that controls the number of seconds that *condor_dagman* will sleep before submitting consecutive jobs. It can be increased to help reduce the load on the *condor_schedd* daemon. The legal range of values is 0 to 60. If defined with a value less than 0, the value 0 will be used. If defined with a value greater than 60, the value 60 will be used. The default value is 0.

DAGMAN_STARTUP_CYCLE_DETECT A boolean value that when `True` causes *condor_dagman* to check for cycles in the DAG before submitting DAG node jobs, in addition to its run time cycle detection. If not defined, it defaults to `False`.

DAGMAN_RETRY_SUBMIT_FIRST A boolean value that controls whether a failed submit is retried first (before any other submits) or last (after all other ready jobs are submitted). If this value is set to `True`, when a job submit fails, the job is placed at the head of the queue of ready jobs, so that it will be submitted again before any other jobs are submitted (this has been the behavior of *condor_dagman* up to this point). If this value is set to `False`, when a job submit fails, the job is placed at the tail of the queue of ready jobs. If not defined, it defaults to `True`.

DAGMAN_RETRY_NODE_FIRST A boolean value that controls whether a failed node (with retries) is retried first (before any other ready nodes) or last (after all other ready nodes). If this value is set to `True`, when a node with retries fails (after the submit succeeded), the node is placed at the head of the queue of ready nodes, so that it will be tried again before any other jobs are

submitted. If this value is set to `False`, when a node with retries fails, the node is placed at the tail of the queue of ready nodes (this has been the behavior of *condor_dagman* up to this point). If not defined, it defaults to `False`.

DAGMAN_MAX_JOBS_IDLE An integer value that controls the maximum number of idle node jobs allowed within the DAG before *condor_dagman* temporarily stops submitting jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. If both the command-line flag and the configuration parameter are specified, the command-line flag overrides the configuration parameter. Unfortunately, `DAGMAN_MAX_JOBS_IDLE` currently counts each individual process within a cluster as a job, which is inconsistent with `DAGMAN_MAX_JOBS_SUBMITTED`. The default is that there is no limit on the maximum number of idle jobs.

DAGMAN_MAX_JOBS_SUBMITTED An integer value that controls the maximum number of node jobs within the DAG that will be submitted to Condor at one time. Note that this parameter is the same as the `-maxjobs` command-line flag to *condor_submit_dag*. If both the command-line flag and the configuration parameter are specified, the command-line flag overrides the configuration parameter. A single invocation of *condor_submit* counts as one job, even if the submit file produces a multi-job cluster. The default is that there is no limit on the maximum number of jobs run at one time.

DAGMAN_MUNGE_NODE_NAMES A boolean value that controls whether *condor_dagman* automatically renames nodes when running multiple DAGs (the renaming is done to avoid possible name conflicts). If this value is set to `True`, all node names have the "DAG number" prepended to them. For example, the first DAG specified on the *condor_submit_dag* command line is considered DAG number 0, the second is DAG number 1, etc. So if DAG number 2 has a node B, that node will internally be renamed to "2.B". If not defined, `DAGMAN_MUNGE_NODE_NAMES` defaults to `True`.

DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION This macro is no longer used. The improved functionality of the `DAGMAN_ALLOW_EVENTS` macro eliminates the need for this variable.

A boolean value that controls whether *condor_dagman* aborts or continues with a DAG in the rare case that Condor erroneously executes the job within a DAG node more than once. A bug in Condor very occasionally causes a job to run twice. Running a job twice is contrary to the semantics of a DAG. The configuration macro `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION` determines whether *condor_dagman* considers this a fatal error or not. The default value is `False`; *condor_dagman* considers running the job more than once a fatal error, logs this fact, and aborts the DAG. When set to `True`, *condor_dagman* still logs this fact, but continues with the DAG.

This configuration macro is to remain at its default value except in the case where a site encounters the Condor bug in which DAG job nodes are executed twice, and where it is certain that having a DAG job node run twice will not corrupt the DAG. The logged messages within `*.dagman.out` files in the case of that a node job runs twice contain the string "EVENT ERROR."

DAGMAN_ALLOW_EVENTS An integer that controls which "bad" events are considered fatal errors by *condor_dagman*. This macro replaces and expands upon the functionality of the

DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION macro. If DAGMAN_ALLOW_EVENTS is set, it overrides the setting of DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION.

The DAGMAN_ALLOW_EVENTS value is a bitwise-OR of the following values:

- 0 = allow no "bad" events
- 1 = allow almost all "bad" events (all except "job re-run after terminated event")
- 2 = allow terminated/aborted event combination
- 4 = allow "job re-run after terminated event" bug
- 8 = allow garbage/orphan events
- 16 = allow execute or terminate event before job's submit event
- 32 = allow two terminated events per job (sometimes seen with grid jobs)
- 64 = allow duplicated events in general

The default value is 114 (allow terminated/aborted event combination, allow execute and/or terminated event before job's submit event, allow double terminated events, and allow general duplicate events).

For example, a value of 6 instructs *condor_dagman* to allow both the terminated/aborted event combination and the "job re-run after terminated event" bug. A value of 0 means that any "bad" event will be considered a fatal error.

A value of 5 (1 + 4) will never abort the DAG because of a "bad" event – but you should almost never use this setting, because the "job re-run after terminated event" bug breaks the semantics of the DAG.

This macro should almost always remain set to the default value!

DAGMAN_DEBUG This macro is described in section 3.3.4 as <SUBSYS>_DEBUG.

MAX_DAGMAN_LOG This macro is described in section 3.3.4 as MAX_<SUBSYS>_LOG.

DAGMAN_CONDOR_SUBMIT_EXE The executable that *condor_dagman* will use to submit Condor jobs. If not defined, *condor_dagman* looks for *condor_submit* in the PATH.

DAGMAN_STORK_SUBMIT_EXE The executable that *condor_dagman* will use to submit Stork jobs. If not defined, *condor_dagman* looks for *stork_submit* in the PATH.

DAGMAN_CONDOR_RM_EXE The executable that *condor_dagman* will use to remove Condor jobs. If not defined, *condor_dagman* looks for *condor_rm* in the PATH.

DAGMAN_STORK_RM_EXE The executable that *condor_dagman* will use to remove Stork jobs. If not defined, *condor_dagman* looks for *stork_rm* in the PATH.

DAGMAN_PROHIBIT_MULTIJOB A boolean value that controls whether *condor_dagman* prohibits node job submit files that queue multiple job procs (other than parallel universe). If a DAG references such a submit file, the DAG will abort during the initialization process. If not defined, DAGMAN_PROHIBIT_MULTIJOB defaults to False.

DAGMAN_LOG_ON_NFS_IS_ERROR A boolean value that controls whether *condor_dagman* prohibits node job submit files with user log files on NFS. If a DAG references such a submit file and DAGMAN_LOG_ON_NFS_IS_ERROR is True, the DAG will abort during the initialization process. If DAGMAN_LOG_ON_NFS_IS_ERROR is False, a warning will be issued but the DAG will still be submitted. It is *strongly* recommended that DAGMAN_LOG_ON_NFS_IS_ERROR remain set to the default value, because running a DAG with node job log files on NFS will often cause errors. If not defined, DAGMAN_LOG_ON_NFS_IS_ERROR defaults to True.

DAGMAN_ABORT_DUPLICATES A boolean value that controls whether to attempt to abort duplicate instances of *condor_dagman* running the same DAG on the same machine. When *condor_dagman* starts up, if no DAG lock file exists, *condor_dagman* creates the lock file and writes its PID into it. If the lock file does exist, and DAGMAN_ABORT_DUPLICATES is set to True, *condor_dagman* checks whether a process with the given PID exists, and if so, it assumes that there is already another instance of *condor_dagman* running on the same DAG. Note that this test is not foolproof: it is possible that, if *condor_dagman* crashes, the same PID gets reused by another process before *condor_dagman* gets rerun on that DAG. This should be quite rare, however. If not defined, DAGMAN_ABORT_DUPLICATES defaults to True.

DAGMAN_SUBMIT_DEPTH_FIRST A boolean value that controls whether to submit ready DAG node jobs in (more-or-less) depth first order, as opposed to breadth-first order. Setting DAGMAN_SUBMIT_DEPTH_FIRST to True does *not* override dependencies defined in the DAG. Rather, it causes newly-ready nodes to be added to the head, rather than the tail, of the ready node list. If there are no PRE scripts in the DAG, this will cause the ready nodes to be submitted depth-first. If there are PRE scripts, the order will not be strictly depth-first, but it will tend to favor depth rather than breadth in executing the DAG. If you set DAGMAN_SUBMIT_DEPTH_FIRST to True, you may also want to set DAGMAN_RETRY_SUBMIT_FIRST and DAGMAN_RETRY_NODE_FIRST to True. If not defined, DAGMAN_SUBMIT_DEPTH_FIRST defaults to false.

DAGMAN_ON_EXIT_REMOVE The OnExitRemove expression put into the *condor_dagman* submit file by *condor_submit_dag*. The default expression is designed to ensure that *condor_dagman* is automatically re-queued by the schedd if it exits abnormally or is killed (e.g., during a reboot). If this results in *condor_dagman* staying in the queue when it should exit, you may want to change to a less restrictive expression, for example:

```
(ExitBySignal == false || ExitSignal != 9)
```

If not defined, DAGMAN_ON_EXIT_REMOVE defaults to

```
( ExitSignal == 11 || (ExitCode != UNDEFINED && ExitCode >= 0 && ExitCode <=
```

DAGMAN_ABORT_ON_SCARY_SUBMIT A boolean value that controls whether to abort a DAG upon detection of a “scary” submit event (one in which the Condor ID does not match the expected value). Note that in all versions prior to 6.9.3, *condor_dagman* has *not*

aborted a DAG upon detection of a “scary” submit event (this behavior is what now happens if `DAGMAN_ABORT_ON_SCARY_SUBMIT` is set to `false`). If not defined, `DAGMAN_ABORT_ON_SCARY_SUBMIT` defaults to `true`.

DAGMAN_PENDING_REPORT_INTERVAL An integer value (in seconds) that controls how often *condor_dagman* will print a report of pending nodes to the `dagman.out` file. Note that the report will only be printed if *condor_dagman* has been waiting at least `DAGMAN_PENDING_REPORT_INTERVAL` seconds without seeing any node job user log events, in order to avoid cluttering the `dagman.out` file. (This feature is mainly intended to help diagnose “stuck” *condor_dagman* processes that are waiting indefinitely for a job to finish.) If not defined, `DAGMAN_PENDING_REPORT_INTERVAL` defaults to 600 seconds (10 minutes).

DAGMAN_INSERT_SUB_FILE A file name of a file containing submit file commands to be inserted into the `.condor.sub` file created by *condor_submit_dag*. The specified file is inserted into the `.condor.sub` file before the **queue** command and before any commands specified with the **-append** *condor_submit_dag* command-line option. Note that the `DAGMAN_INSERT_SUB_FILE` value can be overridden by the **-insert_sub_file** *condor_submit_dag* command-line option.

DAGMAN_OLD_RESCUE A boolean value that controls whether *condor_dagman* uses “old-style” rescue DAG naming when creating a rescue DAG. (With “old-style” rescue DAG naming, if your DAG file is `my.dag`, the rescue DAG file will be `my.dag.rescue`, and that file will be overwritten if you re-run `my.dag` and it fails again. With “new-style” rescue DAG naming, the first time a rescue DAG is created for `my.dag`, it will be named `my.dag.rescue001`, and subsequent failures of `my.dag` will produce rescue DAGs named `my.dag.rescue002`, `my.dag.rescue003`, etc.) If not defined, `DAGMAN_OLD_RESCUE` defaults to `false`.

DAGMAN_AUTO_RESCUE A boolean value that controls whether *condor_dagman* automatically runs rescue DAGs. If `DAGMAN_AUTO_RESCUE` is `true` and you run the DAG file `my.dag`, if a rescue dag such as `my.dag.rescue001`, `my.dag.rescue002`, etc., exists, the newest (highest-numbered) such rescue DAG will be run. If not defined, `DAGMAN_AUTO_RESCUE` defaults to `true`.

Note: having `DAGMAN_OLD_RESCUE` and `DAGMAN_AUTO_RESCUE` both set to `true` is a fatal error.

DAGMAN_MAX_RESCUE_NUM An integer value that controls the maximum “new-style” rescue DAG number that will be written (if `DAGMAN_OLD_RESCUE` is `false`) or run (if `DAGMAN_AUTO_RESCUE` is `true`). The maximum legal value is 999; the minimum value is 0 (which will prevent a rescue DAG from being written at all, or automatically run). If not defined, `DAGMAN_MAX_RESCUE_NUM` defaults to 100.

DAGMAN_COPY_TO_SPOOL A boolean value that controls whether the *condor_dagman* binary is copied to the spool directory when a DAG is submitted. The main reason for setting this value to `true` is if you have long-running DAGs that should survive a DAGMan version upgrade. If you run large numbers of small DAGs, you should leave this macro unset (or set it to `false`). If not defined, `DAGMAN_COPY_TO_SPOOL` defaults to `false`.

3.3.26 Configuration File Entries Relating to Security

These macros affect the secure operation of Condor. Many of these macros are described in section 3.6 on Security.

SEC*_AUTHENTICATION This section has not yet been written

SEC*_ENCRYPTION This section has not yet been written

SEC*_INTEGRITY This section has not yet been written

SEC*_NEGOTIATION This section has not yet been written

SEC*_AUTHENTICATIONMETHODS This section has not yet been written

SEC*_CRYPTOMETHODS This section has not yet been written

GSI_DAEMONNAME This configuration variable is retired. Instead use `ALLOW_CLIENT` or `DENY_CLIENT` as appropriate. When used, this variable defined a comma separated list of the subject name(s) of the certificate(s) that the daemons use.

GSI_DAEMON_DIRECTORY A directory name used in the construction of complete paths for the configuration variables `GSI_DAEMON_CERT`, `GSI_DAEMON_KEY`, and `GSI_DAEMON_TRUSTED_CA_DIR`, for any of these configuration variables are not explicitly set.

GSI_DAEMON_CERT A complete path and file name to the X.509 certificate to be used in GSI authentication. If this configuration variable is not defined, and `GSI_DAEMON_DIRECTORY` is defined, then Condor uses `GSI_DAEMON_DIRECTORY` to construct the path and file name as

```
GSI_DAEMON_CERT = $(GSI_DAEMON_DIRECTORY)/hostcert.pem
```

GSI_DAEMON_KEY A complete path and file name to the X.509 private key to be used in GSI authentication. If this configuration variable is not defined, and `GSI_DAEMON_DIRECTORY` is defined, then Condor uses `GSI_DAEMON_DIRECTORY` to construct the path and file name as

```
GSI_DAEMON_KEY = $(GSI_DAEMON_DIRECTORY)/hostkey.pem
```

GSI_DAEMON_TRUSTED_CA_DIR The directory that contains the list of trusted certification authorities to be used in GSI authentication. The files in this directory are the public keys and signing policies of the trusted certification authorities. If this configuration variable is not defined, and `GSI_DAEMON_DIRECTORY` is defined, then Condor uses `GSI_DAEMON_DIRECTORY` to construct the directory path as

```
GSI_DAEMON_TRUSTED_CA_DIR = $(GSI_DAEMON_DIRECTORY)/certificates
```

GSI_DAEMON_PROXY A complete path and file name to the X.509 proxy to be used in GSI authentication. When this configuration variable is defined, use of this proxy takes precedence over use of a certificate and key.

DELEGATE_JOB_GSI_CREDENTIALS A boolean value that defaults to `True` for Condor version 6.7.19 and more recent versions. When `True`, a job's GSI X.509 credentials are delegated, instead of being copied. This results in a more secure communication when not encrypted.

GRIDMAP The complete path and file name of the Globus Gridmap file. The Gridmap file is used to map X.509 distinguished names to Condor user ids.

SEC_DEFAULT_SESSION_DURATION The amount of time in seconds before a communication session expires. Defaults to 86400 seconds (1 day). A session is a record of necessary information to do communication between a client and daemon, and is protected by a shared secret key. The session expires to reduce the window of opportunity where the key may be compromised by attack.

SEC_INVALIDATE_SESSIONS_VIA_TCP Use TCP (if `True`) or UDP (if `False`) for responding to attempts to use an invalid security session. This happens, for example, if a daemon restarts and receives incoming commands from other daemons that are still using a previously established security session. The default is `True`.

FS_REMOTEDIR The location of a file visible to both server and client in Remote File System authentication. The default when not defined is the directory `/shared/scratch/tmp`.

ENCRYPT_EXECUTED_DIRECTORY The execute directory for jobs on Windows platforms may be encrypted by setting this configuration variable to `True`. Defaults to `False`. The method of encryption uses the EFS (Encrypted File System) feature of Windows NTFS v5.

SEC_TCP_SESSION_TIMEOUT The length of time in seconds until the timeout on individual network operations when establishing a UDP security session via TCP. The default value is 20 seconds. Scalability issues with a large pool would be the only basis for a change from the default value.

SEC_TCP_SESSION_DEADLINE An integer representing the total length of time in seconds until giving up when establishing a security session. Whereas `SEC_TCP_SESSION_TIMEOUT` specifies the timeout for individual blocking operations (connect, read, write), this setting specifies the total time across all operations, including non-blocking operations that have little cost other than holding open the socket. The default value is 120 seconds. The intention of this setting is to avoid waiting for hours for a response in the rare event that the other side freezes up and the socket remains in a connected state. This problem has been observed in some types of operating system crashes.

SEC_DEFAULT_AUTHENTICATION_TIMEOUT The length of time in seconds that Condor should attempt authenticating network connections before giving up. The default is 20 seconds. Like

other security settings, the portion of the configuration variable name, `DEFAULT`, may be replaced by a different access level to specify the timeout to use for different types of commands, for example `SEC_CLIENT_AUTHENTICATION_TIMEOUT`.

SEC_PASSWORDFILE For Unix machines, the path and file name of the file containing the pool password for password authentication.

AUTH_SSL_SERVER_CAFILE The path and file name of a file containing one or more trusted CA's certificates for the server side of a communication authenticating with SSL.

AUTH_SSL_CLIENT_CAFILE The path and file name of a file containing one or more trusted CA's certificates for the client side of a communication authenticating with SSL.

AUTH_SSL_SERVER_CADIR The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the server side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

AUTH_SSL_CLIENT_CADIR The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the client side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

AUTH_SSL_SERVER_CERTFILE The path and file name of the file containing the public certificate for the server side of a communication authenticating with SSL.

AUTH_SSL_CLIENT_CERTFILE The path and file name of the file containing the public certificate for the client side of a communication authenticating with SSL.

AUTH_SSL_SERVER_KEYFILE The path and file name of the file containing the private key for the server side of a communication authenticating with SSL.

AUTH_SSL_CLIENT_KEYFILE The path and file name of the file containing the private key for the client side of a communication authenticating with SSL.

CERTIFICATEMAPFILE A path and file name of the unified map file.

SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION This is a special authentication mechanism designed to minimize overhead in the *condor_schedd* when communicating with the execute machine. Essentially, matchmaking results in a secret being shared between the *condor_schedd* and *condor_startd*, and this is used to establish a strong security session between the execute and submit daemons without going through the usual security negotiation protocol. This is especially important when operating at large scale over high latency networks (e.g. a glidein pool with one schedd and thousands of startds on a network with 0.1 second round trip times).

The default value for this configuration option is `False`. To have any effect, it must be `True` in the configuration of both the execute side (*startd*) as well as the submit side (*schedd*). When this authentication method is used, all other security negotiation between the submit and execute daemons is bypassed. All inter-daemon communication between the

submit and execute side will use the startd's settings for `SEC_DAEMON_ENCRYPTION` and `SEC_DAEMON_INTEGRITY`; the configuration of these values in the `schedd`, `shadow`, and `starter` are ignored.

Important: For strong security, at least one of the two, integrity or encryption, should be enabled in the startd configuration. Also, some form of strong mutual authentication (e.g. GSI) should be enabled between all daemons and the central manager or the shared secret which is exchanged in matchmaking cannot be safely encrypted when transmitted over the network.

The `schedd` and `shadow` will be authenticated as `submit-side@matchsession` when they talk to the startd and starter. The startd and starter will be authenticated as `execute-side@matchsession` when they talk to the `schedd` and `shadow`. On the submit side, authorization of the execute side happens automatically. On the execute side, it is necessary to explicitly authorize the submit side. Example:

```
ALLOW_DAEMON = submit-side@matchsession/192.168.123.*
```

Replace the example netmask with something suitable for your situation.

KERBEROS_SERVER_KEYTAB The path and file name of the keytab file that holds the necessary Kerberos principals. If not defined, this variable's value is set by the installed Kerberos; it is `/etc/v5srvtab` on most systems.

KERBEROS_SERVER_PRINCIPAL An exact Kerberos principal to use. The default value is `host/<hostname>@<realm>`, as set by the installed Kerberos. Where both `KERBEROS_SERVER_PRINCIPAL` and `KERBEROS_SERVER_SERVICE` are defined, this value takes precedence.

KERBEROS_SERVER_USER The user name that the Kerberos server principal will map to after authentication. The default value is `condor`.

KERBEROS_SERVER_SERVICE A string representing the Kerberos service name. This string is prepended with a slash character (`/`) and the host name in order to form the Kerberos server principal. This value defaults to `host`, resulting in the same default value as specified by using `KERBEROS_SERVER_PRINCIPAL`. Where both `KERBEROS_SERVER_PRINCIPAL` and `KERBEROS_SERVER_SERVICE` are defined, the value of `KERBEROS_SERVER_PRINCIPAL` takes precedence.

KERBEROS_CLIENT_KEYTAB The path and file name of the keytab file for the client in Kerberos authentication. This variable has no default value.

3.3.27 Configuration File Entries Relating to PrivSep

PRIVSEP_ENABLED A boolean variable that, when `True`, enables PrivSep. When `True`, the `condor_proc` is used, ignoring the definition of the configuration variable `USE_PROCD`. The default value when this configuration variable is not defined is `False`.

PRIVSEP_SWITCHBOARD The full (trusted) path and file name of the `condor_root_switchboard` executable.

3.3.28 Configuration File Entries Relating to Virtual Machines

These macros affect how Condor runs **vm** universe jobs on a matched machine within the pool. They specify items related to the *condor_vm-gahp*.

VM_GAHP_SERVER The complete path and file name of the *condor_vm-gahp*. There is no default value for this required configuration variable.

VM_GAHP_LOG The complete path and file name of the *condor_vm-gahp* log. If not specified on a Unix platform, the *condor_starter* log will be used for *condor_vm-gahp* log items. There is no default value for this required configuration variable on Windows platforms.

MAX_VM_GAHP_LOG Controls the maximum length (in bytes) to which the *condor_vm-gahp* log will be allowed to grow.

VM_TYPE Specifies the type of supported virtual machine software. It will be the value `xen` or `vmware`. There is no default value for this required configuration variable.

VM_VERSION Specifies the version of supported virtual machine software defined by `VM_TYPE`. There is no default value for this required configuration variable. This configuration variable does not currently alter the behavior of the *condor_vm-gahp*; instead, it is used in *condor_status* when printing VM-capable hosts and slots.

VM_MEMORY An integer to specify the maximum amount of memory in Mbytes that will be allowed to the virtual machine program.

VM_MAX_NUMBER An integer limit on the number of executing virtual machines. When not defined, the default value is the same `NUM_CPUS`.

VM_STATUS_INTERVAL An integer number of seconds that defaults to 60, representing the interval between job status checks by the *condor_starter* to see if the job has finished. A minimum value of 30 seconds is enforced.

VM_GAHP_REQ_TIMEOUT An integer number of seconds that defaults to 300 (five minutes), representing the amount of time Condor will wait for a command issued from the *condor_starter* to the *condor_vm-gahp* to be completed. When a command times out, an error is reported to the *condor_startd*.

VM_RECHECK_INTERVAL An integer number of seconds that defaults to 600 (ten minutes), representing the amount of time the *condor_startd* waits after a virtual machine error as reported by the *condor_starter*, and before checking a final time on the status of the virtual machine. If the check fails, Condor disables starting any new vm universe jobs by removing the `VM_Type` attribute from the machine ClassAd.

VM_SOFT_SUSPEND A boolean value that defaults to `False`, causing Condor to free the memory of a vm universe job when the job is suspended. When `True`, the memory is not freed.

VM_UNIV_NOBODY_USER Identifies a login name of a user with a home directory that may be used for job owner of a vm universe job. The `nobody` user normally utilized when the job arrives from a different UID domain will not be allowed to invoke a VMware virtual machine.

ALWAYS_VM_UNIV_USE_NOBODY A boolean value that defaults to `False`. When `True`, all vm universe jobs (independent of their UID domain) will run as the user defined in `VM_UNIV_NOBODY_USER`.

VM_NETWORKING A boolean variable describing if networking is supported. When not defined, the default value is `False`.

VM_NETWORKING_TYPE A string describing the type of networking, required and relevant only when `VM_NETWORKING` is `True`. Defined strings are

```
bridge
nat
nat, bridge
```

VM_NETWORKING_DEFAULT_TYPE Where multiple networking types are given in `VM_NETWORKING_TYPE`, this optional configuration variable identifies which to use. Therefore, for

```
VM_NETWORKING_TYPE = nat, bridge
```

this variable may be defined as either `nat` or `bridge`. Where multiple networking types are given in `VM_NETWORKING_TYPE`, and this variable is *not* defined, a default of `nat` is used.

The following configuration variables are specific to the VMware virtual machine software.

VMWARE_PERL The complete path and file name to *Perl*. There is no default value for this required variable.

VMWARE_SCRIPT The complete path and file name of the script that controls VMware. There is no default value for this required variable.

VMWARE_NETWORKING_TYPE An optional string used in networking that the *condor_vm-gahp* inserts into the VMware configuration file to define a networking type. Defined types are `nat` or `bridged`. If a default value is needed, the inserted string will be `nat`.

VMWARE_NAT_NETWORKING_TYPE An optional string used in networking that the *condor_vm-gahp* inserts into the VMware configuration file to define a networking type. If `nat` networking is used, this variable's definition takes precedence over one defined by `VMWARE_NETWORKING_TYPE`.

VMWARE_BRIDGE_NETWORKING_TYPE An optional string used in networking that the *condor_vm-gahp* inserts into the VMware configuration file to define a networking type. If `bridge` networking is used, this variable's definition takes precedence over one defined by `VMWARE_NETWORKING_TYPE`.

VMWARE_LOCAL_SETTINGS_FILE The complete path and file name to a file, whose contents will be inserted into the VMware description file (i.e., the .vmx file) before Condor starts the virtual machine. This parameter is optional.

The following configuration variables are specific to the Xen virtual machine software.

XEN_SCRIPT The complete path and file name of the script that controls Xen. There is no default value for this required variable.

XEN_DEFAULT_KERNEL The complete path and executable name of a Xen kernel to be utilized if the job's submission does not specify its own kernel image.

XEN_DEFAULT_INITRD The complete path and image file name for the initrd image, if used with the default kernel image.

XEN_BOOTLOADER A required full path and executable for the Xen bootloader, if the kernel image includes a disk image.

XEN_BRIDGE_SCRIPT A path, file name, and command-line arguments to specify a script that will be run to set up a bridging network interface for guests. The interface should provide direct access to the host system's LAN, that is, not be NAT'd on the host. An example:

```
XEN_BRIDGE_SCRIPT = vif-bridge bridge=xenbr0
```

XEN_LOCAL_SETTINGS_FILE A complete path and file name. The file's contents will be included in the Xen configuration file that Condor writes to run the virtual machine. This parameter is optional.

The following two macros affect the configuration of Condor where Condor is running on a host machine, the host machine is running an inner virtual machine, and Condor is also running on that inner virtual machine. These two variables have nothing to do with the **vm** universe.

VMP_HOST_MACHINE A configuration variable for the inner virtual machine, which specifies the host name.

VMP_VM_LIST For the host, a comma separated list of the host names or IP addresses for machines running inner virtual machines on a host.

3.3.29 Configuration File Entries Relating to High Availability

These macros affect the high availability operation of Condor.

MASTER_HA_LIST Similar to `DAEMON_LIST`, this macro defines a list of daemons that the *condor_master* starts and keeps its watchful eyes on. However, the `MASTER_HA_LIST` daemons are run in a *High Availability* mode. The list is a comma or space separated list of subsystem names (as listed in section 3.3.1). For example,

```
MASTER_HA_LIST = SCHEDD
```

The *High Availability* feature allows for several *condor_master* daemons (most likely on separate machines) to work together to insure that a particular service stays available. These *condor_master* daemons ensure that one and only one of them will have the listed daemons running.

To use this feature, the lock URL must be set with `HA_LOCK_URL`.

Currently, only file URLs are supported (those with `file:...`). The default value for `MASTER_HA_LIST` is the empty string, which disables the feature.

HA_LOCK_URL This macro specifies the URL that the *condor_master* processes use to synchronize for the *High Availability* service. Currently, only file URLs are supported; for example, `file:/share/spool`. Note that this URL must be identical for all *condor_master* processes sharing this resource. For *condor_schedd* sharing, we recommend setting up `SPOOL` on an NFS share and having all *High Availability condor_schedd* processes sharing it, and setting the `HA_LOCK_URL` to point at this directory as well. For example:

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = SCHEDD.lock
```

A separate lock is created for each *High Availability* daemon.

There is no default value for `HA_LOCK_URL`.

Lock files are in the form `<SUBSYS>.lock`. *condor_preen* is not currently aware of the lock files and will delete them if they are placed in the `SPOOL` directory, so be sure to add `<SUBSYS>.lock` to `VALID_SPOOL_FILES` for each *High Availability* daemon.

HA_<SUBSYS>_LOCK_URL This macro controls the *High Availability* lock URL for a specific subsystem as specified in the configuration variable name, and it overrides the system-wide lock URL specified by `HA_LOCK_URL`. If not defined for each subsystem, `HA_<SUBSYS>_LOCK_URL` is ignored, and the value of `HA_LOCK_URL` is used.

HA_LOCK_HOLD_TIME This macro specifies the number of seconds that the *condor_master* will hold the lock for each *High Availability* daemon. Upon gaining the shared lock, the *condor_master* will hold the lock for this number of seconds. Additionally, the *condor_master* will periodically renew each lock as long as the *condor_master* and the daemon are running. When the daemon dies, or the *condor_master* exists, the *condor_master* will immediately release the lock(s) it holds.

`HA_LOCK_HOLD_TIME` defaults to 3600 seconds (one hour).

HA_<SUBSYS>_LOCK_HOLD_TIME This macro controls the *High Availability* lock hold time for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by `HA_LOCK_HOLD_TIME`. If not defined for each subsystem, `HA_<SUBSYS>_LOCK_HOLD_TIME` is ignored, and the value of `HA_LOCK_HOLD_TIME` is used.

HA_POLL_PERIOD This macro specifies how often the *condor_master* polls the *High Availability* locks to see if any locks are either stale (meaning not updated for `HA_LOCK_HOLD_TIME` seconds), or have been released by the owning *condor_master*. Additionally, the *condor_master* renews any locks that it holds during these polls.

`HA_POLL_PERIOD` defaults to 300 seconds (five minutes).

HA_<SUBSYS>_POLL_PERIOD This macro controls the *High Availability* poll period for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by `HA_POLL_PERIOD`. If not defined for each subsystem, `HA_<SUBSYS>_POLL_PERIOD` is ignored, and the value of `HA_POLL_PERIOD` is used.

MASTER_<SUBSYS>_CONTROLLER Used only in HA configurations involving the *condor_had*.

The *condor_master* has the concept of a controlling and controlled daemon, typically with the *condor_had* daemon serving as the controlling process. In this case, all *condor_on* and *condor_off* commands directed at controlled daemons are given to the controlling daemon, which then handles the command, and, when required, sends appropriate commands to the *condor_master* to do the actual work. This allows the controlling daemon to know the state of the controlled daemon.

As of 6.7.14, this configuration variable must be specified for all configurations using *condor_had*. To configure the *condor_negotiator* controlled by *condor_had*:

```
MASTER_NEGOTIATOR_CONTROLLER = HAD
```

The macro is named by substituting `<SUBSYS>` with the appropriate subsystem string as defined in section 3.3.1.

HAD_LIST A comma-separated list of all *condor_had* daemons in the form `IP:port` or `hostname:port`. Each central manager machine that runs the *condor_had* daemon should appear in this list. If `HAD_USE_PRIMARY` is set to `True`, then the first machine in this list is the primary central manager, and all others in the list are backups.

All central manager machines must be configured with an identical `HAD_LIST`. The machine addresses are identical to the addresses defined in `COLLECTOR_HOST`.

HAD_USE_PRIMARY Boolean value to determine if the first machine in the `HAD_LIST` configuration variable is a primary central manager. Defaults to `False`.

HAD_CONNECTIONTIMEOUT The time (in seconds) that the *condor_had* daemon waits before giving up on the establishment of a TCP connection. The failure of the communication connection is the detection mechanism for the failure of a central manager machine. For a LAN, a recommended value is 2 seconds. The use of authentication (by Condor) increases the connection time. The default value is 5 seconds. If this value is set too low, *condor_had* daemons will incorrectly assume the failure of other machines.

HAD_ARGS Command line arguments passed by the *condor_master* daemon as it invokes the *condor_had* daemon. To make high availability work, the *condor_had* daemon requires the port number it is to use. This argument is of the form

```
-p $(HAD_PORT_NUMBER)
```

where `HAD_PORT_NUMBER` is a helper configuration variable defined with the desired port number. Note that this port number must be the same value here as used in `HAD_LIST`. There is no default value.

HAD The path to the *condor_had* executable. Normally it is defined relative to `$(SBIN)`. This configuration variable has no default value.

MAX_HAD_LOG Controls the maximum length in bytes to which the *condor_had* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix `.old`. The `.old` file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

HAD_DEBUG Logging level for the *condor_had* daemon. See `<SUBSYS>_DEBUG` for values.

HAD_LOG Full path and file name of the log file. There is no default value.

REPLICATION_LIST A comma-separated list of all *condor_replication* daemons in the form `IP:port` or `hostname:port`. Each central manager machine that runs the *condor_had* daemon should appear in this list. All potential central manager machines must be configured with an identical `REPLICATION_LIST`.

STATE_FILE A full path and file name of the file protected by the replication mechanism. When not defined, the default path and file used is

```
$(SPOOL)/Accountantnew.log
```

REPLICATION_INTERVAL Sets how often the *condor_replication* daemon initiates its tasks of replicating the `$(STATE_FILE)`. It is defined in seconds and defaults to 300 (5 minutes). This is the same as the default `NEGOTIATOR_INTERVAL`.

MAX_TRANSFERER_LIFETIME A timeout period within which the process that transfers the state file must complete its transfer. The recommended value is `2 * average size of state file / network rate`. It is defined in seconds and defaults to 300 (5 minutes).

HAD_UPDATE_INTERVAL Like `UPDATE_INTERVAL`, determines how often the *condor_had* is to send a ClassAd update to the *condor_collector*. Updates are also sent at each and every change in state. It is defined in seconds and defaults to 300 (5 minutes).

HAD_USE_REPLICATION A boolean value that defaults to `False`. When `True`, the use of *condor_replication* daemons is enabled.

REPLICATIONARGS Command line arguments passed by the *condor_master* daemon as it invokes the *condor_replication* daemon. To make high availability work, the *condor_replication* daemon requires the port number it is to use. This argument is of the form

```
-p $(REPLICATION_PORT_NUMBER)
```

where `REPLICATION_PORT_NUMBER` is a helper configuration variable defined with the desired port number. Note that this port number must be the same value as used in `REPLICATION_LIST`. There is no default value.

REPLICATION The full path and file name of the *condor_replication* executable. It is normally defined relative to `$(SBIN)`. There is no default value.

MAX_REPLICATIONLOG Controls the maximum length in bytes to which the *condor_replication* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix `.old`. The `.old` file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

REPLICATIONDEBUG Logging level for the *condor_replication* daemon. See `<SUBSYS>_DEBUG` for values.

REPLICATIONLOG Full path and file name to the log file. There is no default value.

TRANSFERER The full path and file name of the *condor_transferer* executable. Versions of Condor previous to 7.2.2 hard coded the location as `$(RELEASE_DIR)/sbin/condor_transferer`. This is now the default value. The future default value is likely to change, and be defined relative to `$(SBIN)`.

TRANSFERERLOG Full path and file name to the log file. There is no default value for this variable; a definition is required if the *condor_replication* daemon does a file transfer.

TRANSFERERDEBUG Logging level for the *condor_transferer* daemon. See `<SUBSYS>_DEBUG` for values.

MAX_TRANSFERERLOG Controls the maximum length in bytes to which the *condor_transferer* daemon log will be allowed to grow. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

3.3.30 Configuration File Entries Relating to Quill

These macros affect the Quill database management and interface to its representation of the job queue.

QUILL The full path name to the *condor_quill* daemon.

QUILLARGS Arguments to be passed to the *condor_quill* daemon upon its invocation.

QUILLLOG Path to the Quill daemon's log file.

QUILLENABLED A boolean variable that defaults to `False`. When `True`, Quill functionality is enabled. When `False`, the Quill daemon writes a message to its log and exits. The *condor_q* and *condor_history* tools then do not use Quill.

QUILLNAME A string that uniquely identifies an instance of the *condor_quill* daemon, as there may be more than *condor_quill* daemon per pool. The string must not be the same as for any *condor_schedd* daemon.

See the description of `MASTER_NAME` in section 3.3.9 on page 177 for defaults and composition of valid Condor daemon names.

QUILLUSESQLLOG In order for Quill to store historical job information or resource information, the Condor daemons must write information to the SQL logfile. By default, this is set to `False`, and the only information Quill stores in the database is the current job queue. This can be set on a per daemon basis. For example, to store information about historical jobs, but not store execute resource information, set `QUILLUSESQLLOG` to `False` and set `SCHEDD.QUILLUSESQLLOG` to `True`.

QUILLDBNAME A string that identifies a database within a database server.

QUILLDBUSER A string that identifies the *PostgreSQL* user that Quill will connect to the database as. We recommend “**quillwriter**” for this setting.

QUILLDBTYPE A string that distinguishes between database system types. Defaults to the only database system currently defined, “`PGSQL`”.

QUILLDBIPADDR The host address of the database server. It can be either an IP address or an IP address. It must match exactly what is used in the `.pgpass` file.

QUILLPOLLINGPERIOD The frequency, in number of seconds, at which the Quill daemon polls the file `job_queue.log` for updates. New information in the log file is sent to the database. The default value is 10.

QUILLNOTRESPONDINGTIMEOUT The length of time, in seconds, before the *condor_master* may decide that the *condor_quill* daemon is hung due to a lack of communication, potentially causing the *condor_master* to kill and restart the *condor_quill* daemon. When the *condor_quill* daemon is processing a very long log file, it may not be able to communicate with the master. The default is 3600 seconds, or one hour. It may be advisable to increase this to several hours.

QUILLMAINTAINDBCONN A boolean variable that defaults to `True`. When `True`, the *condor_quill* daemon maintains an open connection the database server, which speeds up updates to the database. As each open connection consumes resources at the database server, we recommend a setting of `False` for large pools.

DATABASEPURGEINTERVAL The interval, in seconds, between scans of the database to identify and delete records that are beyond their history durations. The default value is 86400, or one day.

DATABASEREINDEXINTERVAL The interval, in seconds, between reindex commands on the database. The default value is 86400, or one day. This is only used when the `QUILL_DB_TYPE` is set to "PGSQL".

QUILLJOBHISTORYDURATION The number of days after entry into the database that a job will remain in the database. After `QUILLJOBHISTORYDURATION` days, the job is deleted. The job history is the final ClassAd, and contains all information necessary for *condor_history* to succeed. The default is 3650, or about 10 years.

QUILLRUNHISTORYDURATION The number of days after entry into the database that extra information about the job will remain in the database. After `QUILLRUNHISTORYDURATION` days, the records are deleted. This data includes matches made for the job, file transfers the job performed, and user log events. The default is 7 days, or one week.

QUILLRESOURCEHISTORYDURATION The number of days after entry into the database that a resource record will remain in the database. After `QUILLRESOURCEHISTORYDURATION` days, the record is deleted. The resource history data includes the ClassAd of a compute slot, submitter ClassAds, and daemon ClassAds. The default is 7 days, or one week.

QUILLDBSIZELIMIT After each purge, the *condor_quill* daemon estimates the size of the database. If the size of the database exceeds this limit, the *condor_quill* daemon will e-mail the administrator a warning. This size is given in gigabytes, and defaults to 20.

QUILLMANAGEVACUUM A boolean value that defaults to `False`. When `True`, the *condor_quill* daemon takes on the maintenance task of vacuuming the database. As of *PostgreSQL* version 8.1, the database can perform this task automatically; therefore, having the *condor_quill* daemon vacuum is not necessary. A value of `True` causes warnings to be written to the log file.

QUILLSHOULOREINDEX A boolean value that defaults to `True`. When `True`, the *condor_quill* daemon will re-index the database tables when the history file is purged of old data. So, if Quill is configured to never delete history data, the tables are never re-indexed.

QUILLISREMOTELYQUERYABLE A boolean value that defaults to `True`. When `False`, the remote database tables may not be remotely queryable.

QUILLDBQUERYPASSWORD Defines the password string needed by *condor_q* to gain read access for remotely querying the Quill database.

QUILLADDRESSFILE When defined, it specifies the path and file name of a local file containing the IP address and port number of the Quill daemon. By using the file, tools executed on the local machine do not need to query the central manager in order to find the *condor_quill* daemon.

DBMSD The full path name to the *condor_dbmsd* daemon. The default location is `$(SBIN)/condor_dbmsd`.

DBMSD_ARGS Arguments to be passed to the *condor_dbmsd* daemon upon its invocation. The default arguments are `-f`.

DBMSD_LOG Path to the *condor_dbmsd* daemon's log file. The default log location is `$(LOG)/DbmsdLog`.

DBMSD_NOT_RESPONDING_TIMEOUT The length of time, in seconds, before the *condor_master* may decide that the *condor_dbmsd* is hung due to a lack of communication, potentially causing the *condor_master* to kill and restart the *condor_dbmsd* daemon. When the *condor_dbmsd* is purging or reindexing a very large database, it may not be able to communicate with the master. The default is 3600 seconds, or one hour. It may be advisable to increase this to several hours.

3.3.31 MyProxy Configuration File Macros

In some cases, Condor can autonomously refresh GSI certificate proxies via *MyProxy*, available from <http://myproxy.ncsa.uiuc.edu/>.

MYPROXY_GET_DELEGATION The full path name to the *myproxy-get-delegation* executable, installed as part of the *MyProxy* software. Often, it is necessary to wrap the actual executable with a script that sets the environment, such as the `LD_LIBRARY_PATH`, correctly. If this macro is defined, Condor-G and *condor_credd* will have the capability to autonomously refresh proxy certificates. By default, this macro is undefined.

3.3.32 Configuration File Macros Affecting APIs

ENABLE_SOAP A boolean value that defaults to `False`. When `True`, Condor daemons will respond to HTTP PUT commands as if they were SOAP calls. When `False`, all HTTP PUT commands are denied.

ENABLE_WEB_SERVER A boolean value that defaults to `False`. When `True`, Condor daemons will respond to HTTP GET commands, and send the static files sitting in the subdirectory defined by the configuration variable `WEB_ROOT_DIR`. In addition, web commands are considered a `READ` command, so the client will be checked by host-based security.

SOAP_LEAVE_IN_QUEUE A boolean value that when `True`, causes a job in the completed state to remain in the queue, instead of being removed based on the completion of file transfer. There is no default value.

WEB_ROOT_DIR A complete path to the directory containing all the files served by the web server.

<SUBSYS>_ENABLE_SOAP_SSL A boolean value that defaults to `False`. When `True`, enables SOAP over SSL for the specified `<SUBSYS>`. Any specific `<SUBSYS>_ENABLE_SOAP_SSL` setting overrides the value of `ENABLE_SOAP_SSL`.

ENABLE_SOAP_SSL A boolean value that defaults to `False`. When `True`, enables SOAP over SSL for all daemons.

<SUBSYS>_SOAP_SSL_PORT A required port number on which SOAP over SSL messages are accepted, when SOAP over SSL is enabled. The `<SUBSYS>` must be specified, because multiple daemons running on a single machine may not share a port. There is no default value.

The macro is named by substituting `<SUBSYS>` with the appropriate subsystem string as defined in section 3.3.1.

SOAP_SSL_SERVER_KEYFILE A required complete path and file name to specify the daemon's identity, as used in authentication when SOAP over SSL is enabled. The file is to be an OpenSSL PEM file containing a certificate and private key. There is no default value.

SOAP_SSL_SERVER_KEYFILE_PASSWORD An optional complete path and file name to specify a password for unlocking the daemon's private key. There is no default value.

SOAP_SSL_CA_FILE A required complete path and file name to specify a file containing certificates of trusted Certificate Authorities (CAs). Only clients who present a certificate signed by a trusted CA will be authenticated. There is no default value.

SOAP_SSL_CA_DIR A required complete path to a directory containing certificates of trusted Certificate Authorities (CAs). Only clients who present a certificate signed by a trusted CA will be authenticated. There is no default value.

SOAP_SSL_DH_FILE An optional complete path and file name to a DH file containing keys for a DH key exchange. There is no default value.

3.3.33 Stork Configuration File Macros

STORK_MAX_NUM_JOBS An integer limit on the number of concurrent data placement jobs handled by Stork. The default value when not defined is 10.

STORK_MAX_RETRY An integer limit on the number of attempts for a single data placement job. For data transfers, this includes transfer attempts on the primary protocol, all alternate protocols, and all retries. The default value when not defined is 10.

STORK_MAX_DELAY_IN_MINUTES An integer limit (in minutes) on the run time for a data placement job, after which the job is considered failed. The default value when not defined is 10, and the minimum legal value is 1.

STORK_TMP_CRED_DIR The full path to the temporary credential storage directory used by Stork. The default value is `/tmp` when not defined.

STORK_MODULE_DIR The full path to the directory containing Stork modules. The default value when not defined is as defined by `$(LIBEXEC)`. It is a fatal error for both `STORK_MODULE_DIR` and `LIBEXEC` to be undefined.

CRED_SUPER_USERS Access to a stored credential is restricted to the user who submitted the credential, and any user names specified in this macro. The format is a space or comma separated list of user names which are valid on the *stork_credd* host. The default value of this macro is `root` on Unix systems, and `Administrator` on Windows systems.

CRED_STORE_DIR Directory for storing credentials. This directory must exist prior to starting *stork_credd*. It is highly recommended to restrict access permissions to *only* the directory owner. The default value is `$(SPOOL_DIR)/cred`.

CRED_INDEX_FILE Index file path of saved credentials. This file will be automatically created if it does not exist. The default value is `$(CRED_STORE_DIR)/cred-index`.

DEFAULT_CRED_EXPIRE_THRESHOLD *stork_credd* will attempt to refresh credentials when their remaining lifespan is less than this value. Units = seconds. Default value = 3600 seconds (1 hour).

CRED_CHECK_INTERVAL *stork_credd* periodically checks remaining lifespan of stored credentials, at this interval. Units = seconds. Default value = 60 seconds (1 minute).

3.4 User Priorities and Negotiation

Condor uses priorities to determine machine allocation for jobs. This section details the priorities and the allocation of machines (negotiation).

For accounting purposes, each user is identified by `username@uid_domain`. Each user is assigned a priority value even if submitting jobs from different machines in the same domain, or even if submitting from multiple machines in the different domains.

The numerical priority value assigned to a user is inversely related to the *goodness* of the priority. A user with a numerical priority of 5 gets more resources than a user with a numerical priority of 50. There are two priority values assigned to Condor users:

- Real User Priority (RUP), which measures resource usage of the user.
- Effective User Priority (EUP), which determines the number of resources the user can get.

This section describes these two priorities and how they affect resource allocations in Condor. Documentation on configuring and controlling priorities may be found in section 3.3.17.

3.4.1 Real User Priority (RUP)

A user's RUP measures the resource usage of the user through time. Every user begins with a RUP of one half (0.5), and at steady state, the RUP of a user equilibrates to the number of resources used by that user. Therefore, if a specific user continuously uses exactly ten resources for a long period of time, the RUP of that user stabilizes at ten.

However, if the user decreases the number of resources used, the RUP gets better. The rate at which the priority value decays can be set by the macro `PRIORITY_HALFLIFE`, a time period defined in seconds. Intuitively, if the `PRIORITY_HALFLIFE` in a pool is set to 86400 (one day), and if a user whose RUP was 10 removes all his jobs, the user's RUP would be 5 one day later, 2.5 two days later, and so on.

3.4.2 Effective User Priority (EUP)

The effective user priority (EUP) of a user is used to determine how many resources that user may receive. The EUP is linearly related to the RUP by a *priority factor* which may be defined on a per-user basis. Unless otherwise configured, the priority factor for all users is 1.0, and so the EUP is the same as the the RUP. However, if desired, the priority factors of specific users (such as remote submitters) can be increased so that others are served preferentially.

The number of resources that a user may receive is inversely related to the ratio between the EUPs of submitting users. Therefore user *A* with EUP=5 will receive twice as many resources as user *B* with EUP=10 and four times as many resources as user *C* with EUP=20. However, if *A* does not use the full number of allocated resources, the available resources are repartitioned and distributed among remaining users according to the inverse ratio rule.

Condor supplies mechanisms to directly support two policies in which EUP may be useful:

Nice users A job may be submitted with the parameter `nice_user` set to `TRUE` in the submit command file. A nice user job gets its RUP boosted by the `NICE_USER_PRIO_FACTOR` priority factor specified in the configuration file, leading to a (usually very large) EUP. This corresponds to a low priority for resources. These jobs are therefore equivalent to Unix background jobs, which use resources not used by other Condor users.

Remote Users The flocking feature of Condor (see section 5.2) allows the *condor_schedd* to submit to more than one pool. In addition, the submit-only feature allows a user to run a *condor_schedd* that is submitting jobs into another pool. In such situations, submitters from other domains can submit to the local pool. It is often desirable to have Condor treat local users preferentially over these remote users. If configured, Condor will boost the RUPs of remote users by `REMOTE_PRIO_FACTOR` specified in the configuration file, thereby lowering their priority for resources.

The priority boost factors for individual users can be set with the **setfactor** option of *condor_userprio*. Details may be found in the *condor_userprio* manual page on page 838.

3.4.3 Priorities and Preemption

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time. In addition, Condor may preempt a machine claim and reallocate it when conditions change.

Too many preemptions lead to thrashing, a condition in which negotiation for a machine identifies a new job with a better priority most every cycle. Each job is, in turn, preempted, and no job finishes. To avoid this situation, the `PREEMPTION_REQUIREMENTS` configuration variable is defined for and used only by the *condor_negotiator* daemon to specify the conditions that must be met for a preemption to occur. It is usually defined to deny preemption if a current running job has been running for a relatively short period of time. This effectively limits the number of preemptions per resource per time interval.

Note that `PREEMPTION_REQUIREMENTS` only applies to preemptions due to user priority. It does not have any effect if the machine's `RANK` expression prefers a different job, or if the machine's policy causes the job to vacate due to other activity on the machine. See section 3.5.9 for a general discussion of limiting preemption.

The following attributes may be used within the definition of `PREEMPTION_REQUIREMENTS` and `PREEMPTION_RANK`. In these attributes, those with names that begin with the string `Submitter` refer to characteristics about the candidate job's user; those with names that begin with the string `Remote` refer to characteristics about the user currently using the resource. Further, those with names that end with the string `ResourcesInUse` have values that may change within the time period associated with a single negotiation cycle. Therefore, the configuration variables `PREEMPTION_REQUIREMENTS_STABLE` and `PREEMPTION_RANK_STABLE` exist to inform the *condor_negotiator* daemon that values may change. See section 3.3.17 on page 213 for complete definitions.

SubmitterUserPrio A floating point value representing the user priority of the candidate job.

SubmitterUserResourcesInUse The integer number of slots currently utilized by the user submitting the candidate job.

RemoteUserPrio A floating point value representing the user priority of the job currently running on the machine.

RemoteUserResourcesInUse The integer number of slots currently utilized by the user of the job currently running on the machine.

SubmitterGroupResourcesInUse If the owner of the candidate job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots currently utilized by the group.

SubmitterGroupQuota If the owner of the candidate job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots defined as the group's quota.

RemoteGroupResourcesInUse If the owner of the currently running job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots currently utilized by the group.

RemoteGroupQuota If the owner of the currently running job is a member of a valid accounting group, with a defined group quota, then this attribute is the integer number of slots defined as the group's quota.

3.4.4 Priority Calculation

This section may be skipped if the reader so feels, but for the curious, here is Condor's priority calculation algorithm.

The RUP of a user u at time t , $\pi_r(u, t)$, is calculated every time interval δt using the formula

$$\pi_r(u, t) = \beta \times \pi_r(u, t - \delta t) + (1 - \beta) \times \rho(u, t)$$

where $\rho(u, t)$ is the number of resources used by user u at time t , and $\beta = 0.5^{\delta t/h}$. h is the half life period set by `PRIORITY_HALFLIFE`.

The EUP of user u at time t , $\pi_e(u, t)$ is calculated by

$$\pi_e(u, t) = \pi_r(u, t) \times f(u, t)$$

where $f(u, t)$ is the priority boost factor for user u at time t .

As mentioned previously, the RUP calculation is designed so that at steady state, each user's RUP stabilizes at the number of resources used by that user. The definition of β ensures that the calculation of $\pi_r(u, t)$ can be calculated over non-uniform time intervals δt without affecting the calculation. The time interval δt varies due to events internal to the system, but Condor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

3.4.5 Negotiation

Negotiation is the method Condor undergoes periodically to match queued jobs with resources capable of running jobs. The *condor_negotiator* daemon is responsible for negotiation.

During a negotiation cycle, the *condor_negotiator* daemon accomplishes the following ordered list of items.

1. Build a list of all possible resources, regardless of the state of those resources.
2. Obtain a list of all job submitters (for the entire pool).
3. Sort the list of all job submitters based on EUP (see section 3.4.2 for an explanation of EUP). The submitter with the best priority is first within the sorted list.
4. Iterate until there are either no more resources to match, or no more jobs to match.

For each submitter (in EUP order):

For each submitter, get each job. Since jobs may be submitted from more than one machine (hence to more than one *condor_schedd* daemon), here is a further definition of the ordering of these jobs. With jobs from a single *condor_schedd* daemon, jobs are typically returned in job priority order. When more than one *condor_schedd* daemon is involved, they are contacted in an undefined order. All jobs from a single *condor_schedd* daemon are considered before moving on to the next. For each job:

- For each machine in the pool that can execute jobs:
 - (a) If `machine.requirements` evaluates to `False` or `job.requirements` evaluates to `False`, skip this machine
 - (b) If the machine is in the Claimed state, but not running a job, skip this machine.
 - (c) If this machine is not running a job, add it to the potential match list by reason of No Preemption.
 - (d) If the machine is running a job
 - If the `machine.RANK` on this job is better than the running job, add this machine to the potential match list by reason of Rank.
 - If the EUP of this job is better than the EUP of the currently running job, and `PREEMPTION_REQUIREMENTS` is `True`, and the `machine.RANK` on this job is not worse than the currently running job, add this machine to the potential match list by reason of Priority.
- Of machines in the potential match list, sort by `NEGOTIATOR_PRE_JOB_RANK`, `job.RANK`, `NEGOTIATOR_POST_JOB_RANK`, Reason for claim (No Preemption, then Rank, then Priority), `PREEMPTION_RANK`
- The job is assigned to the top machine on the potential match list. The machine is removed from the list of resources to match (on this negotiation cycle).

The *condor_negotiator* asks the *condor_schedd* for the "next job" from a given submitter/user. Typically, the *condor_schedd* returns jobs in the order of job priority. If priorities are the same, job submission time is used; older jobs go first. If a cluster has multiple procs in it and one of the jobs cannot be matched, the *condor_schedd* will not return any more jobs in that cluster on that negotiation pass. This is an optimization based on the theory that the cluster jobs are similar. The configuration variable `NEGOTIATE_ALL_JOBS_IN_CLUSTER` disables the cluster-skipping optimization. Use of the configuration variable `SIGNIFICANT_ATTRIBUTES` will change the definition of what the *condor_schedd* considers a cluster from the default definition of all jobs that share the same `ClusterId`.

3.4.6 The Layperson's Description of the Pie Spin and Pie Slice

Condor schedules in a variety of ways. First, it takes all users who have submitted jobs and calculates their priority. Then, it totals the number of resources available at the moment, and using the ratios of the user priorities, it calculates the number of machines each user could get. This is their *pie slice*.

The Condor matchmaker goes in user priority order, contacts each user, and asks for job information. The *condor_schedd* daemon (on behalf of a user) tells the matchmaker about a job, and the matchmaker looks at available resources to create a list of resources that match the requirements expression. With the list of resources that match, it sorts them according to the rank expressions within *ClassAds*. If a machine prefers a job, the job is assigned to that machine, potentially preempting a job that might already be running on that machine. Otherwise, give the machine to the job that the job ranks highest. If the machine ranked highest is already running a job, we may preempt running

job for the new job. A default policy for preemption states that the user must have a 20% better priority in order for preemption to succeed. If the job has no preferences as to what sort of machine it gets, matchmaking gives it the first idle resource to meet its requirements.

This matchmaking cycle continues until the user has recieved all of the machines in their pie slice. The matchmaker then contacts the next highest priority user and offers that user their pie slice worth of machines. After contacting all users, the cycle is repeated with any still available resources and recomputed pie slices. The matchmaker continues *spinning the pie* until it runs out of machines or all the *condor_schedd* daemons say they have no more jobs.

3.4.7 Group Accounting

By default, Condor does all accounting on a per-user basis, and this accounting is primarily used to compute priorities for Condor's fair-share scheduling algorithms. However, accounting can also be done on a per-group basis. Multiple users can all submit jobs into the same accounting group, and all of the jobs will be treated with the same priority.

To use an accounting group, each job inserts an attribute into the job ClassAd which defines the accounting group name for the job. A common name is decided upon and used for the group. The following line is an example that defines the attribute within the job's submit description file:

```
+AccountingGroup = "group_physics"
```

The `AccountingGroup` attribute is a string, and it therefore must be enclosed in double quote marks. The string may have a maximum length of 40 characters. The name should *not* be qualified with a domain. Certain parts of the Condor system do append the value `$(UID_DOMAIN)` (as specified in the configuration file on the submit machine) to this string for internal use. For example, if the value of `UID_DOMAIN` is `example.com`, and the accounting group name is as specified, *condor_userprio* will show statistics for this accounting group using the appended domain, for example

User Name	Effective Priority
group_physics@example.com	0.50
user@example.com	23.11
heavyuser@example.com	111.13
...	

Additionally, the *condor_userprio* command allows administrators to remove an entity from the accounting system in Condor. The **-delete** option to *condor_userprio* accomplishes this if all the jobs from a given accounting group are completed, and the administrator wishes to remove that group from the system. The **-delete** option identifies the accounting group with the fully-qualified name of the accounting group. For example

```
condor_userprio -delete group_physics@example.com
```

Condor removes entities itself as they are no longer relevant. Intervention by an administrator to delete entities can be beneficial when the use of thousands of short term accounting groups leads to scalability issues.

Note that the name of an accounting group may include a period (.). Inclusion of a period character in the accounting group name only has relevance if the portion of the name before the period matches a group name, as described in the next section on group quotas.

3.4.8 Group Quotas

The use of group quotas modifies the negotiation for available resources (machines) within a Condor pool. This solves the difficulties inherent when priorities assigned based on each single user are insufficient. This may be the case when different groups (of varying size) own computers, and the groups choose to combine their computers to form a Condor pool. Consider an imaginary Condor pool example with thirty computers; twenty computers are owned by the physics group and ten computers are owned by the chemistry group. One notion of fair allocation could be implemented by configuring the twenty machines owned by the physics group to prefer (using the RANK configuration macro) jobs submitted by the users identified as associated with the physics group. Likewise, the ten machines owned by the chemistry group are configured to prefer jobs from users associated with the chemistry group. This routes jobs to execute on specific machines, perhaps causing more preemption than necessary. The (fair allocation) policy desired is likely somewhat different, if these thirty machines have been pooled. The desired policy does not tie users to specific sets of machines, but to numbers of machines (a quota). Given thirty similar machines, the desired policy allows users within the physics group to have preference on up to twenty of the machines within the pool, and the machines can be any of the machines that are available.

A quota for a set of users requires an identification of the set; members are called group users. Jobs under the group quota specify the group user with the `AccountingGroup` job `ClassAd` attribute. This is the same attribute as is used with group accounting.

The submit file syntax for specifying a group user includes both a group name and a user name. The syntax is

```
+AccountingGroup = "<group>.<user>"
```

The `group` is a name chosen for the group. Group names are case-insensitive for negotiation. Group names are not required to begin with the string `"group_"`, as in the examples `"group_physics.newton"` and `"group_chemistry.curie"`, but it is a useful convention, because group names must not conflict with user names. The period character between the group and the user name is a required part of the syntax. **NOTE:** An accounting group value lacking the period will cause the job to not be considered part of the group when negotiating, even if the group name has a quota. Furthermore, there will be no warnings that the group quota is not in effect for the job, as this syntax defines group accounting.

Configuration controls the order of negotiation for groups and individual users, as well as sets quotas (preferentially allocated numbers of machines) for the groups. A declared number of slots specifies the quota for each group (see `GROUP_QUOTA_<groupname>` in section 3.3.17). The sum of the quotas for all groups must be less than or equal to the number of slots in the entire pool. If the sum is less than the number of slots in the entire pool, the slots are allocated to the `none` group, comprised of the general users not submitting jobs in a group.

Where group users are specified for jobs, accounting is done per group user. It is no longer done by group, or by individual user.

Negotiation is changed when group quotas are used. Condor negotiates first for defined groups, and then for independent job submitters. Given jobs belonging to different groups, Condor negotiates first for the group currently utilizing the smallest percentage of machines in its quota. After this, Condor negotiates for the group currently utilizing the second smallest percentage of machines in its quota. The last group will be the one with the highest percentage of machines in its quota. As an example, again use the imaginary pool and groups given above. If various users within `group_physics` have jobs running on 15 computers, then the physics group has 75% of the machines within its quota. If various users within `group_chemistry` have jobs running on 5 computers, then the chemistry group has 50% of the machines within its quota. Negotiation will take place for the chemistry group first. For independent job submissions (those not part of any group), the classic Condor user fair share algorithm still applies.

Note that there is no verification that a user is a member of the group that he claims. We rely on societal pressure for enforcement.

Configuration variables affect group quotas. See section 3.3.17 for detailed descriptions of the variables mentioned. Group names that may be given quotas to be used in negotiation are listed in the `GROUP_NAMES` macro. The names chosen must not conflict with Condor user names. Quotas (by group) are defined in numbers of machine slots. Each group may be assigned an initial value for its user priority factor with the `GROUP_PRIO_FACTOR_<groupname>` macro. If a group is currently allocated its entire quota of machines, and a group user has a submitted job that is not running, the `GROUP_AUTOREGROUP` macro allows the job to be considered a second time within the negotiation cycle along with all other individual users' jobs.

```
#####
#
# Example 1
# Configuration for group quotas
#
#####

GROUP_NAMES = group_physics, group_chemistry
GROUP_QUOTA_group_physics = 20
GROUP_QUOTA_group_chemistry = 10
GROUP_PRIO_FACTOR_group_physics = 1.0
GROUP_PRIO_FACTOR_group_chemistry = 3.0
GROUP_AUTOREGROUP_group_physics = FALSE
GROUP_AUTOREGROUP_group_chemistry = TRUE
```

This configuration specifies that the `group_physics` users will get 20 machines and the `group_chemistry` users will get ten machines. `group_physics` users will never get more than 20 machines; however, `group_chemistry` users can potentially get more than ten machines because `GROUP_AUTOREGROUP_chemistry` is true. This could happen, for example, if there are

only 15 jobs submitted by `group_physics` users. Also, the default priority factor for the physics groups is 1.0, and the default priority factor for the chemistry group is 3.0.

```
#####
#
# Submit description file for group quota user
#
#####
...
+AccountingGroup = "group_physics.newton"
...
```

This submit file specifies that this job is to be negotiated as part of the `group_physics` group and that the user is `newton`. Remember that both the group name and the user name are required for the group quota to take effect.

3.5 Policy Configuration for the *condor_startd*

This section describes the configuration of machines, such that they, through the *condor_startd* daemon, implement a desired policy for when remote jobs should start, be suspended, (possibly) resumed, vacate (with a checkpoint) or be killed (no checkpoint). This policy is the heart of Condor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to Condor). Please read this section carefully if you plan to change any of the settings described here, as a wrong setting can have a severe impact on either the owners of machines in your pool (they may ask to be removed from the pool entirely) or the users of your pool (they may stop using Condor).

Before the details, there are a few things to note:

- Much of this section refers to ClassAd expressions. Please read through section 4.1 on ClassAd expressions before continuing.
- If defining the policy for an SMP machine (a multi-CPU machine), also read section 3.12.7 for specific information on configuring the *condor_startd* daemon for SMP machines. Each *slot* represented by the *condor_startd* daemon on an SMP machine has its own *state* and *activity* (as described below). In the future, each slot will be able to have its own individual policy expressions defined. Within this manual section, the word “machine” refers to an individual slot within an SMP machine.

To define a policy, set expressions in the configuration file (see section 3.3 on Configuring Condor for an introduction to Condor's configuration files). The expressions are evaluated in the context of the machine's ClassAd and a job ClassAd. The expressions can therefore reference attributes from either ClassAd. See the unnumbered Appendix on page 879 for a list of job ClassAd attributes.

See the unnumbered Appendix on page 886 for a list of machine ClassAd attributes. The `START` expression is explained. It describes the conditions that must be met for a machine to start a job. The `RANK` expression for a machine is described. It allows the specification of the kinds of jobs a machine prefers to run. A final discussion details how the *condor_startd* daemon works. Included are the machine *states* and *activities*, to give an idea of what is possible in policy decisions. Two example policy settings are presented.

3.5.1 Startd ClassAd Attributes

The *condor_startd* daemon represents the machine on which it is running to the Condor pool. The daemon publishes characteristics about the machine in the machine's ClassAd to aid matchmaking with resource requests. The values of these attributes may be listed by using the command: *condor_status -l hostname*. On an SMP machine, the *condor_startd* will break the machine up and advertise it as separate slots, each with its own name and ClassAd.

3.5.2 The `START` expression

The most important expression to the *condor_startd* is the `START` expression. This expression describes the conditions that must be met for a machine to run a job. This expression can reference attributes in the machine's ClassAd (such as `KeyboardIdle` and `LoadAvg`) and attributes in a job ClassAd (such as `Owner`, `Imagesize`, and `Cmd`, the name of the executable the job will run). The value of the `START` expression plays a crucial role in determining the state and activity of a machine.

The `Requirements` expression is used for matching machines with jobs.

The *condor_startd* defines the `Requirements` expression by logically **and**ing the `START` expression and the `IS_VALID_CHECKPOINT_PLATFORM` expression.

In situations where a machine wants to make itself unavailable for further matches, the `Requirements` expression is set to `FALSE`. When the `START` expression locally evaluates to `TRUE`, the machine advertises the `Requirements` expression as `TRUE` and does not publish the `START` expression.

Normally, the expressions in the machine ClassAd are evaluated against certain request ClassAds in the *condor_negotiator* to see if there is a match, or against whatever request ClassAd currently has claimed the machine. However, by locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If an expression cannot be locally evaluated (because it references other expressions that are only found in a request ad, such as `Owner` or `Imagesize`), the expression is (usually) undefined. See section 4.1 for specifics on how undefined terms are handled in ClassAd expression evaluation.

A note of caution is in order when modifying the `START` to reference job ClassAd attributes. The default `IS_OWNER` expression is a function of the `START` expression

```
START =?= FALSE
```

See a detailed discussion of the `IS_OWNER` expression in section 3.5.7. However, the machine locally evaluates the `IS_OWNER` expression to determine if it is capable of running jobs for Condor. Any job ClassAd attributes appearing in the `START` expression, and hence in the `IS_OWNER` expression are undefined in this context, and may lead to unexpected behavior. Whenever the `START` expression is modified to reference job ClassAd attributes, the `IS_OWNER` expression should also be modified to reference only machine ClassAd attributes.

NOTE: If you have machines with lots of real memory and swap space such that the only scarce resource is CPU time, consider defining `JOB_RENICE_INCREMENT` so that Condor starts jobs on the machine with low priority. Then, further configure to set up the machines with:

```
START = True
SUSPEND = False
PREEMPT = False
KILL = False
```

In this way, Condor jobs always run and can never be kicked off from activity on the machine. However, because they would run with “nice priority”, interactive response on the machines will not suffer. You probably would not notice Condor was running the jobs, assuming you had enough free memory for the Condor jobs that there was little swapping.

3.5.3 The `IS_VALID_CHECKPOINT_PLATFORM` expression

A checkpoint is the platform-dependent information necessary to continue the execution of a standard universe job. Therefore, the machine (platform) upon which a job executed and produced a checkpoint limits the machines (platforms) which may use the checkpoint to continue job execution. This platform-dependent information is no longer the obvious combination of architecture and operating system, but may include subtle items such as the difference between the normal, bigmem, and hugemem kernels within the Linux operating system. This results in the incorporation of a separate expression to indicate the ability of a machine to resume and continue the execution of a job that has produced a checkpoint. The `REQUIREMENTS` expression is dependent on this information.

At a high level, `IS_VALID_CHECKPOINT_PLATFORM` is an expression which becomes true when a job’s checkpoint platform matches the current checkpointing platform of the machine. Since this expression is **anded** with the `START` expression to produce the `REQUIREMENTS` expression, it must also behave correctly when evaluating in the context of jobs that are not standard universe.

In words, the current default policy for this expression:

Any non standard universe job may run on this machine. A standard universe job may run on machines with the new checkpointing identification system. A standard universe job may run if it has not yet produced a first checkpoint. If a standard universe job has produced a checkpoint, then make sure the checkpoint platforms between the job and the machine match.

The following is the default boolean expression for this policy. A `JobUniverse` value of 1 denotes the standard universe. This expression may be overridden in the Condor configuration files.

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  ( (TARGET.JobUniverse == 1) == FALSE) ||

  (
    (MY.CheckpointPlatform != UNDEFINED) &&
    (
      (TARGET.LastCheckpointPlatform == MY.CheckpointPlatform) ||
      (TARGET.NumCkpts == 0)
    )
  )
)
```

`IS_VALID_CHECKPOINT_PLATFORM` is a separate policy expression because the complexity of `IS_VALID_CHECKPOINT_PLATFORM` can be very high. While this functionality is conceptually separate from the normal `START` policies usually constructed, it is also a part of the `Requirements` to allow the job to run.

3.5.4 The **RANK** expression

A machine may be configured to prefer certain jobs over others using the `RANK` expression. It is an expression, like any other in a machine `ClassAd`. It can reference any attribute found in either the machine `ClassAd` or a request ad (normally, in fact, it references things in the request ad). The most common use of this expression is likely to configure a machine to prefer to run jobs from the owner of that machine, or by extension, a group of machines to prefer jobs from the owners of those machines.

For example, imagine there is a small research group with 4 machines called `tenorsax`, `piano`, `bass`, and `drums`. These machines are owned by the 4 users `coltrane`, `tyner`, `garrison`, and `jones`, respectively.

Assume that there is a large Condor pool in your department, but you spent a lot of money on really fast machines for your group. You want to implement a policy that gives priority on your machines to anyone in your group. To achieve this, set the `RANK` expression on your machines to reference the `Owner` attribute and prefer requests where that attribute matches one of the people in your group as in

```
RANK = Owner == "coltrane" || Owner == "tyner" \
      || Owner == "garrison" || Owner == "jones"
```

The `RANK` expression is evaluated as a floating point number. However, like in C, boolean expressions evaluate to either 1 or 0 depending on if they are `TRUE` or `FALSE`. So, if this expression evaluated to 1 (because the remote job was owned by one of the preferred users), it would be a larger value than any other user (for whom the expression would evaluate to 0).

A more complex RANK expression has the same basic set up, where anyone from your group has priority on your machines. Its difference is that the machine owner has better priority on their own machine. To set this up for Jimmy Garrison, place the following entry in Jimmy Garrison's local configuration file `bass.local`:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
      + ((Owner == "garrison") * 10) + (Owner == "jones")
```

NOTE: The parentheses in this expression are important, because “+” operator has higher default precedence than “==”.

The use of “+” instead of “| ” allows us to distinguish which terms matched and which ones didn't. If anyone not in the John Coltrane quartet was running a job on the machine called `bass`, the RANK would evaluate numerically to 0, since none of the boolean terms evaluates to 1, and $0+0+0+0$ still equals 0.

Suppose Elvin Jones submits a job. His job would match this machine (assuming the `START` was `True` for him at that time) and the RANK would numerically evaluate to 1. Therefore, Elvin would preempt the Condor job currently running. Assume that later Jimmy submits a job. The RANK evaluates to 10, since the boolean that matches Jimmy gets multiplied by 10. Jimmy would preempt Elvin, and Jimmy's job would run on Jimmy's machine.

The RANK expression is not required to reference the `Owner` of the jobs. Perhaps there is one machine with an enormous amount of memory, and others with not much at all. You can configure your large-memory machine to prefer to run jobs with larger memory requirements:

```
RANK = ImageSize
```

That's all there is to it. The bigger the job, the more this machine wants to run it. It is an altruistic preference, always servicing the largest of jobs, no matter who submitted them. A little less altruistic is John's RANK that prefers his jobs over those with the largest `ImageSize`:

```
RANK = (Owner == "coltrane" * 1000000000000) + ImageSize
```

This RANK breaks if a job is submitted with an image size of more 10^{12} Kbytes. However, with that size, this RANK expression preferring that job would not be Condor's only problem!

3.5.5 Machine States

A machine is assigned a *state* by Condor. The state depends on whether or not the machine is available to run Condor jobs, and if so, what point in the negotiations has been reached. The possible states are

Owner The machine is being used by the machine owner, and/or is not available to run Condor jobs. When the machine first starts up, it begins in this state.

Unclaimed The machine is available to run Condor jobs, but it is not currently doing so.

Matched The machine is available to run jobs, and it has been matched by the negotiator with a specific schedd. That schedd just has not yet claimed this machine. In this state, the machine is unavailable for further matches.

Claimed The machine has been claimed by a schedd.

Preempting The machine was claimed by a schedd, but is now preempting that claim for one of the following reasons.

1. the owner of the machine came back
2. another user with higher priority has jobs waiting to run
3. another request that this resource would rather serve was found

Backfill The machine is running a backfill computation while waiting for either the machine owner to come back or to be matched with a Condor job. This state is only entered if the machine is specifically configured to enable backfill jobs.

Figure 3.3 shows the states and the possible transitions between the states.

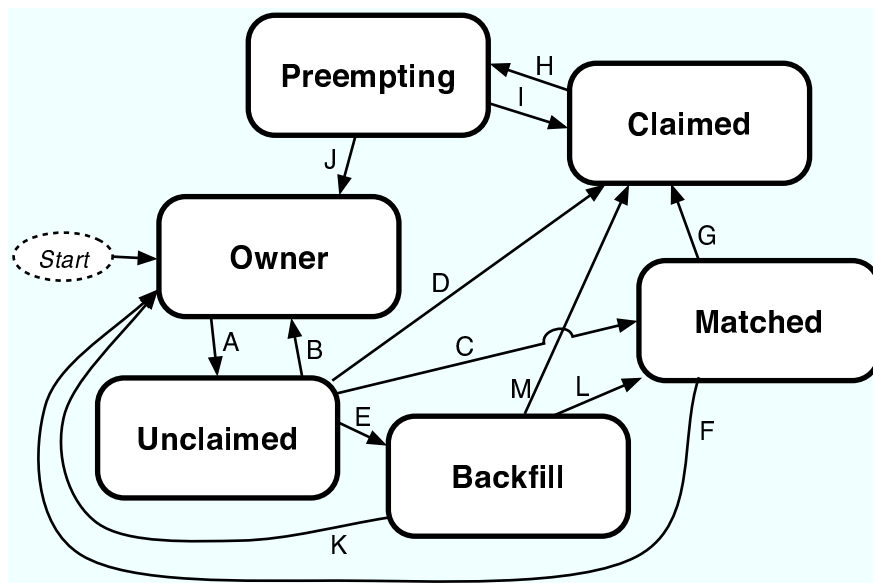


Figure 3.3: Machine States

Each transition is labeled with a letter. The cause of each transition is described below.

- Transitions out of the Owner state

- A** The machine switches from Owner to Unclaimed whenever the `START` expression no longer locally evaluates to `FALSE`. This indicates that the machine is potentially available to run a Condor job.
- Transitions out of the Unclaimed state
 - B** The machine switches from Unclaimed back to Owner whenever the `START` expression locally evaluates to `FALSE`. This indicates that the machine is unavailable to run a Condor job and is in use by the resource owner.
 - C** The transition from Unclaimed to Matched happens whenever the *condor_negotiator* matches this resource with a Condor job.
 - D** The transition from Unclaimed directly to Claimed also happens if the *condor_negotiator* matches this resource with a Condor job. In this case the *condor_schedd* receives the match and initiates the claiming protocol with the machine before the *condor_startd* receives the match notification from the *condor_negotiator*.
 - E** The transition from Unclaimed to Backfill happens if the machine is configured to run backfill computations (see section 3.12.9) and the `START_BACKFILL` expression evaluates to `TRUE`.
- Transitions out of the Matched state
 - F** The machine moves from Matched to Owner if either the `START` expression locally evaluates to `FALSE`, or if the `MATCH_TIMEOUT` timer expires. This timeout is used to ensure that if a machine is matched with a given *condor_schedd*, but that *condor_schedd* does not contact the *condor_startd* to claim it, that the machine will give up on the match and become available to be matched again. In this case, since the `START` expression does not locally evaluate to `FALSE`, as soon as transition **F** is complete, the machine will immediately enter the Unclaimed state again (via transition **A**). The machine might also go from Matched to Owner if the *condor_schedd* attempts to perform the claiming protocol but encounters some sort of error. Finally, the machine will move into the Owner state if the *condor_startd* receives a *condor_vacate* command while it is in the Matched state.
 - G** The transition from Matched to Claimed occurs when the *condor_schedd* successfully completes the claiming protocol with the *condor_startd*.
- Transitions out of the Claimed state
 - H** From the Claimed state, the only possible destination is the Preempting state. This transition can be caused by many reasons:
 - The *condor_schedd* that has claimed the machine has no more work to perform and releases the claim
 - The `PREEMPT` expression evaluates to `TRUE` (which usually means the resource owner has started using the machine again and is now using the keyboard, mouse, CPU, etc)
 - The *condor_startd* receives a *condor_vacate* command
 - The *condor_startd* is told to shutdown (either via a signal or a *condor_off* command)

- The resource is matched to a job with a better priority (either a better user priority, or one where the machine rank is higher)
- Transitions out of the Preempting state
 - I** The resource will move from Preempting back to Claimed if the resource was matched to a job with a better priority.
 - J** The resource will move from Preempting to Owner if the `PREEMPT` expression had evaluated to `TRUE`, if *condor_vacate* was used, or if the `START` expression locally evaluates to `FALSE` when the *condor_startd* has finished evicting whatever job it was running when it entered the Preempting state.
- Transitions out of the Backfill state
 - K** The resource will move from Backfill to Owner for the following reasons:
 - The `EVICT_BACKFILL` expression evaluates to `TRUE`
 - The *condor_startd* receives a *condor_vacate* command
 - The *condor_startd* is being shutdown
 - L** The transition from Backfill to Matched occurs whenever a resource running a backfill computation is matched with a *condor_schedd* that wants to run a Condor job.
 - M** The transition from Backfill directly to Claimed is similar to the transition from Unclaimed directly to Claimed. It only occurs if the *condor_schedd* completes the claiming protocol before the *condor_startd* receives the match notification from the *condor_negotiator*.

The Claimed State and Leases

When a *condor_schedd* claims a *condor_startd*, there is a claim lease. So long as the keep alive updates from the *condor_schedd* to the *condor_startd* continue to arrive, the lease is reset. If the lease duration passes with no updates, the *condor_startd* drops the claim and evicts any jobs the *condor_schedd* sent over.

The alive interval is the amount of time between, or the frequency at which the *condor_schedd* sends keep alive updates to all *condor_schedd* daemons. An alive update resets the claim lease at the *condor_startd*. Updates are UDP packets.

Initially, as when the *condor_schedd* starts up, the alive interval starts at the value set by the configuration variable `ALIVE_INTERVAL`. It may be modified when a job is started. The job's ClassAd attribute `JobLeaseDuration` is checked. If the value of `JobLeaseDuration/3` is less than the current alive interval, then the alive interval is set to either this lower value or the imposed lowest limit on the alive interval of 10 seconds. Thus, the alive interval starts at `ALIVE_INTERVAL` and goes down, never up.

If a claim lease expires, the *condor_startd* will drop the claim. The length of the claim lease is the job's ClassAd attribute `JobLeaseDuration`. `JobLeaseDuration` defaults to 20 minutes time, except when explicitly set within the job's submit description file. If `JobLeaseDuration`

is explicitly set to 0, or it is not set as may be the case for a Web Services job that does not define the attribute, then `JobLeaseDuration` is given the Undefined value. Further, when undefined, the claim lease duration is calculated with `MAX_CLAIM_ALIVES_MISSED * alive interval`. The alive interval is the *current* value, as sent by the *condor_schedd*. If the *condor_schedd* reduces the current alive interval, it does not update the *condor_startd*.

3.5.6 Machine Activities

Within some machine states, *activities* of the machine are defined. The state has meaning regardless of activity. Differences between activities are significant. Therefore, a “state/activity” pair describes a machine. The following list describes all the possible state/activity pairs.

- Owner

Idle This is the only activity for Owner state. As far as Condor is concerned the machine is Idle, since it is not doing anything for Condor.

- Unclaimed

Idle This is the normal activity of Unclaimed machines. The machine is still Idle in that the machine owner is willing to let Condor jobs run, but Condor is not using the machine for anything.

Benchmarking The machine is running benchmarks to determine the speed on this machine. This activity only occurs in the Unclaimed state. How often the activity occurs is determined by the `RUNBENCHMARKS` expression.

- Matched

Idle When Matched, the machine is still Idle to Condor.

- Claimed

Idle In this activity, the machine has been claimed, but the schedd that claimed it has yet to *activate* the claim by requesting a *condor_starter* to be spawned to service a job. The machine returns to this state (usually briefly) when jobs (and therefore *condor_starter*) finish.

Busy Once a *condor_starter* has been started and the claim is active, the machine moves to the Busy activity to signify that it is doing something as far as Condor is concerned.

Suspended If the job is suspended by Condor, the machine goes into the Suspended activity. The match between the schedd and machine has not been broken (the claim is still valid), but the job is not making any progress and Condor is no longer generating a load on the machine.

Retiring When an active claim is about to be preempted for any reason, it enters retirement, while it waits for the current job to finish. The `MaxJobRetirementTime` expression determines how long to wait (counting since the time the job started). Once the job finishes or the retirement time expires, the Preempting state is entered.

- **Preempting** The preempting state is used for evicting a Condor job from a given machine. When the machine enters the Preempting state, it checks the `WANT_VACATE` expression to determine its activity.

Vacating In the Vacating activity, the job that was running is in the process of checkpointing. As soon as the checkpoint process completes, the machine moves into either the Owner state or the Claimed state, depending on the reason for its preemption.

Killing Killing means that the machine has requested the running job to exit the machine immediately, without checkpointing.

- **Backfill**

Idle The machine is configured to run backfill jobs and is ready to do so, but it has not yet had a chance to spawn a backfill manager (for example, the BOINC client).

Busy The machine is performing a backfill computation.

Killing The machine was running a backfill computation, but it is now killing the job to either return resources to the machine owner, or to make room for a regular Condor job.

Figure 3.4 on page 262 gives the overall view of all machine states and activities and shows the possible transitions from one to another within the Condor system. Each transition is labeled with a number on the diagram, and transition numbers referred to in this manual will be **bold**.

Various expressions are used to determine when and if many of these state and activity transitions occur. Other transitions are initiated by parts of the Condor protocol (such as when the *condor_negotiator* matches a machine with a schedd). The following section describes the conditions that lead to the various state and activity transitions.

3.5.7 State and Activity Transitions

This section traces through all possible state and activity transitions within a machine and describes the conditions under which each one occurs. Whenever a transition occurs, Condor records when the machine entered its new activity and/or new state. These times are often used to write expressions that determine when further transitions occurred. For example, enter the Killing activity if a machine has been in the Vacating activity longer than a specified amount of time.

Owner State

When the startd is first spawned, the machine it represents enters the Owner state. The machine remains in the Owner state while the expression `IS_OWNER` is TRUE. If the `IS_OWNER` expression is FALSE, then the machine transitions to the Unclaimed state. The default value for the `IS_OWNER` expression is optimized for a shared resource

```
START =?= FALSE
```

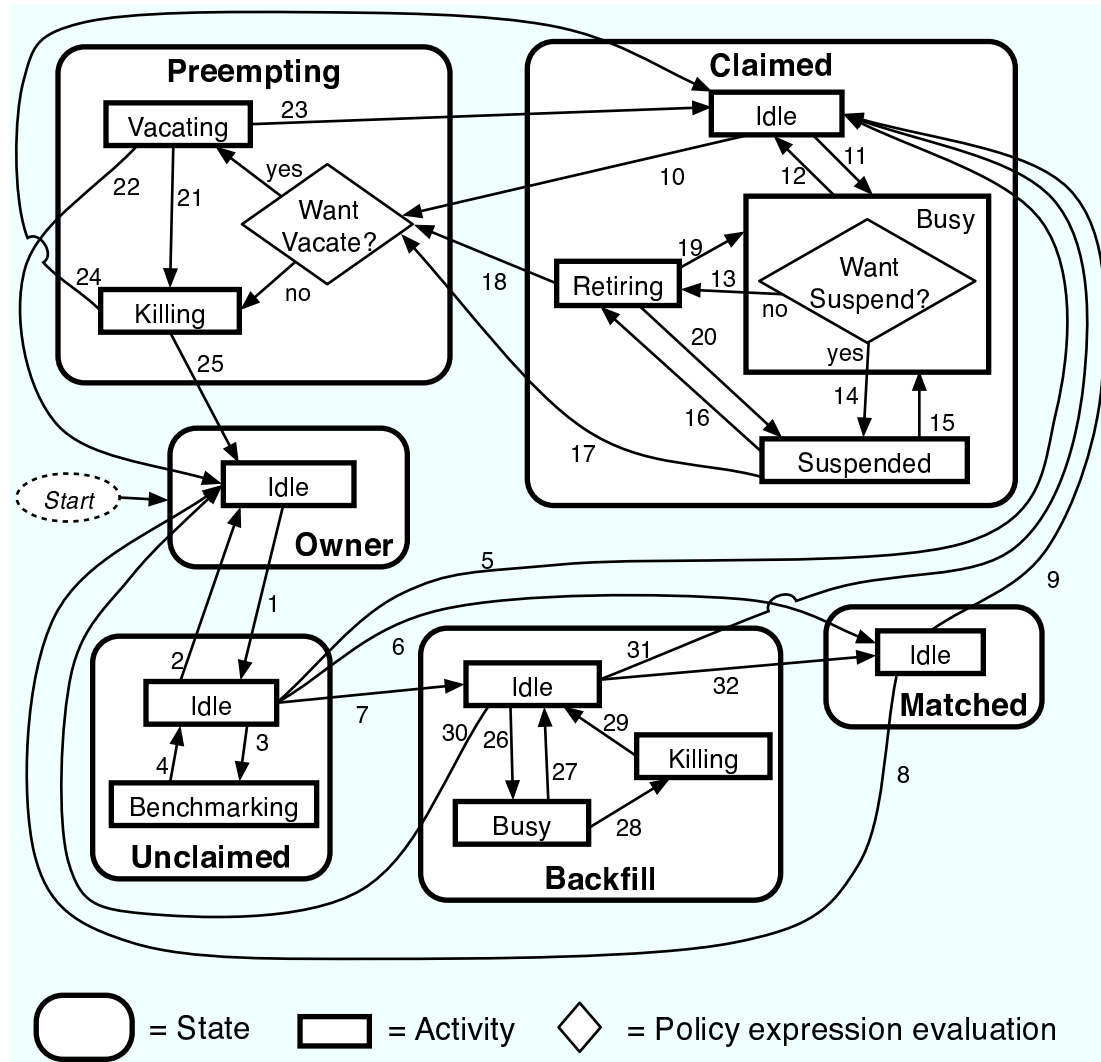


Figure 3.4: Machine States and Activities

So, the machine will remain in the Owner state as long as the *START* expression locally evaluates to *FALSE*. Section 3.5.2 provides more detail on the *START* expression. If the *START* locally evaluates to *TRUE* or cannot be locally evaluated (it evaluates to *UNDEFINED*), transition 1 occurs and the machine enters the Unclaimed state. The *IS_OWNER* expression is locally evaluated by the machine, and should not reference job ClassAd attributes, which would be *UNDEFINED*.

For dedicated resources, the recommended value for the *IS_OWNER* expression is *FALSE*.

The Owner state represents a resource that is in use by its interactive owner (for example, if the keyboard is being used). The Unclaimed state represents a resource that is neither in use by its interactive user, nor the Condor system. From Condor's point of view, there is little difference between the Owner and Unclaimed states. In both cases, the resource is not currently in use by the Condor system. However, if a job matches the resource's *START* expression, the resource is available to run a job, regardless of if it is in the Owner or Unclaimed state. The only differences between the two states are how the resource shows up in *condor_status* and other reporting tools, and the fact that Condor will not run benchmarking on a resource in the Owner state. As long as the *IS_OWNER* expression is *TRUE*, the machine is in the Owner State. When the *IS_OWNER* expression is *FALSE*, the machine goes into the Unclaimed State.

Here is an example that assumes that an *IS_OWNER* expression is not present in the configuration. If the *START* expression is

```
START = KeyboardIdle > 15 * $(MINUTE) && Owner == "coltrane"
```

and if *KeyboardIdle* is 34 seconds, then the machine would remain in the Owner state. *Owner* is undefined, and *anything && FALSE* is *FALSE*.

If, however, the *START* expression is

```
START = KeyboardIdle > 15 * $(MINUTE) || Owner == "coltrane"
```

and *KeyboardIdle* is 34 seconds, then the machine leaves the Owner state and becomes Unclaimed. This is because *FALSE || UNDEFINED* is *UNDEFINED*. So, while this machine is not available to just anybody, if user *coltrane* has jobs submitted, the machine is willing to run them. Any other user's jobs have to wait until *KeyboardIdle* exceeds 15 minutes. However, since *coltrane* might claim this resource, but has not yet, the machine goes to the Unclaimed state.

While in the Owner state, the *startd* polls the status of the machine every *UPDATE_INTERVAL* to see if anything has changed that would lead it to a different state. This minimizes the impact on the Owner while the Owner is using the machine. Frequently waking up, computing load averages, checking the access times on files, computing free swap space take time, and there is nothing time critical that the *startd* needs to be sure to notice as soon as it happens. If the *START* expression evaluates to *TRUE* and five minutes pass before the *startd* notices, that's a drop in the bucket of high-throughput computing.

The machine can only transition to the Unclaimed state from the Owner state. It does so when the *IS_OWNER* expression no longer evaluates to *FALSE*. By default, that happens when *START* no longer locally evaluates to *FALSE*.

Whenever the machine is not actively running a job, it will transition back to the Owner state if *IS_OWNER* evaluates to *TRUE*. Once a job is started, the value of *IS_OWNER* does not matter; the job either runs to completion or is preempted. Therefore, you must configure the preemption policy if you want to transition back to the Owner state from Claimed Busy.

Unclaimed State

If the `IS_OWNER` expression becomes `TRUE`, then the machine returns to the Owner state. If the `IS_OWNER` expression becomes `FALSE`, then the machine remains in the Unclaimed state. If the `IS_OWNER` expression is not present in the configuration files, then the default value for the `IS_OWNER` expression is

```
START =?= FALSE
```

so that while in the Unclaimed state, if the `START` expression locally evaluates to `FALSE`, the machine returns to the Owner state by transition 2.

When in the Unclaimed state, the `RUNBENCHMARKS` expression is relevant. If `RUNBENCHMARKS` evaluates to `TRUE` while the machine is in the Unclaimed state, then the machine will transition from the Idle activity to the Benchmarking activity (transition 3) and perform benchmarks to determine MIPS and KFLOPS. When the benchmarks complete, the machine returns to the Idle activity (transition 4).

The startd automatically inserts an attribute, `LastBenchmark`, whenever it runs benchmarks, so commonly `RunBenchmarks` is defined in terms of this attribute, for example:

```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks = $(BenchmarkTimer) >= (4 * $(HOUR))
```

Here, a macro, `BenchmarkTimer` is defined to help write the expression. This macro holds the time since the last benchmark, so when this time exceeds 4 hours, we run the benchmarks again. The startd keeps a weighted average of these benchmarking results to try to get the most accurate numbers possible. This is why it is desirable for the startd to run them more than once in its lifetime.

NOTE: `LastBenchmark` is initialized to 0 before benchmarks have ever been run. To have the *condor_startd* run benchmarks as soon as the machine is Unclaimed (if it has not done so already), include a term using `LastBenchmark` as in the example above.

NOTE: If `RUNBENCHMARKS` is defined and set to something other than `FALSE`, the startd will automatically run one set of benchmarks when it first starts up. To disable benchmarks, both at startup and at any time thereafter, set `RUNBENCHMARKS` to `FALSE` or comment it out of the configuration file.

From the Unclaimed state, the machine can go to four other possible states: Owner (transition 2), Backfill/Idle, Matched, or Claimed/Idle.

Once the *condor_negotiator* matches an Unclaimed machine with a requester at a given schedd, the negotiator sends a command to both parties, notifying them of the match. If the schedd receives that notification and initiates the claiming procedure with the machine before the negotiator's message gets to the machine, the Match state is skipped, and the machine goes directly to the Claimed/Idle state (transition 5). However, normally the machine will enter the Matched state (transition 6), even if it is only for a brief period of time.

If the machine has been configured to perform backfill jobs (see section 3.12.9), while it is in Unclaimed/Idle it will evaluate the `START_BACKFILL` expression. Once `START_BACKFILL` evaluates to `TRUE`, the machine will enter the Backfill/Idle state (transition 7) to begin the process of running backfill jobs.

Matched State

The Matched state is not very interesting to Condor. Noteworthy in this state is that the machine lies about its `START` expression while in this state and says that `Requirements` are `False` to prevent being matched again before it has been claimed. Also interesting is that the `startd` starts a timer to make sure it does not stay in the Matched state too long. The timer is set with the `MATCH_TIMEOUT` configuration file macro. It is specified in seconds and defaults to 120 (2 minutes). If the schedd that was matched with this machine does not claim it within this period of time, the machine gives up, and goes back into the Owner state via transition 8. It will probably leave the Owner state right away for the Unclaimed state again and wait for another match.

At any time while the machine is in the Matched state, if the `START` expression locally evaluates to `FALSE`, the machine enters the Owner state directly (transition 8).

If the schedd that was matched with the machine claims it before the `MATCH_TIMEOUT` expires, the machine goes into the Claimed/Idle state (transition 9).

Claimed State

The Claimed state is certainly the most complex state. It has the most possible activities and the most expressions that determine its next activities. In addition, the *condor_checkpoint* and *condor_vacate* commands affect the machine when it is in the Claimed state. In general, there are two sets of expressions that might take effect. They depend on the universe of the request: standard or vanilla. The standard universe expressions are the normal expressions. For example:

```
WANT_SUSPEND           = True
WANT_VACATE            = $(ActivationTimer) > 10 * $(MINUTE)
SUSPEND               = $(KeyboardBusy) || $(CPUBusy)
...
```

The vanilla expressions have the string “_VANILLA” appended to their names. For example:

```
WANT_SUSPEND_VANILLA  = True
WANT_VACATE_VANILLA   = True
SUSPEND_VANILLA       = $(KeyboardBusy) || $(CPUBusy)
...
```

Without specific vanilla versions, the normal versions will be used for all jobs, including vanilla jobs. In this manual, the normal expressions are referenced. The difference exists for the the re-

source owner that might want the machine to behave differently for vanilla jobs, since they cannot checkpoint. For example, owners may want vanilla jobs to remain suspended for longer than standard jobs.

While Claimed, the `POLLING_INTERVAL` takes effect, and the startd polls the machine much more frequently to evaluate its state.

If the machine owner starts typing on the console again, it is best to notice this as soon as possible to be able to start doing whatever the machine owner wants at that point. For SMP machines, if any slot is in the Claimed state, the startd polls the machine frequently. If already polling one slot, it does not cost much to evaluate the state of all the slots at the same time.

There are a variety of events that may cause the startd to try to get rid of or temporarily suspend a running job. Activity on the machine's console, load from other jobs, or shutdown of the startd via an administrative command are all possible sources of interference. Another one is the appearance of a higher priority claim to the machine by a different Condor user.

Depending on the configuration, the startd may respond quite differently to activity on the machine, such as keyboard activity or demand for the cpu from processes that are not managed by Condor. The startd can be configured to completely ignore such activity or to suspend the job or even to kill it. A standard configuration for a desktop machine might be to go through successive levels of getting the job out of the way. The first and least costly to the job is suspending it. This works for both standard and vanilla jobs. If suspending the job for a short while does not satisfy the machine owner (the owner is still using the machine after a specific period of time), the startd moves on to vacating the job. Vacating a standard universe job involves performing a checkpoint so that the work already completed is not lost. Vanilla jobs are sent a *soft kill signal* so that they can gracefully shut down if necessary; the default is `SIGTERM`. If vacating does not satisfy the machine owner (usually because it is taking too long and the owner wants their machine back *now*), the final, most drastic stage is reached: killing. Killing is a quick death to the job, using a hard-kill signal that cannot be intercepted by the application. For vanilla jobs that do no special signal handling, vacating and killing are equivalent.

The `WANT_SUSPEND` expression determines if the machine will evaluate the `SUSPEND` expression to consider entering the Suspended activity. The `WANT_VACATE` expression determines what happens when the machine enters the Preempting state. It will go to the Vacating activity or directly to Killing. If one or both of these expressions evaluates to `FALSE`, the machine will skip that stage of getting rid of the job and proceed directly to the more drastic stages.

When the machine first enters the Claimed state, it goes to the Idle activity. From there, it has two options. It can enter the Preempting state via transition **10** (if a *condor_vacate* arrives, or if the `START` expression locally evaluates to `FALSE`), or it can enter the Busy activity (transition **11**) if the schedd that has claimed the machine decides to activate the claim and start a job.

From Claimed/Busy, the machine can transition to three other state/activity pairs. The startd evaluates the `WANT_SUSPEND` expression to decide which other expressions to evaluate. If `WANT_SUSPEND` is `TRUE`, then the startd evaluates the `SUSPEND` expression. If `WANT_SUSPEND` is `FALSE`, then the startd will evaluate the `PREEMPT` expression and skip the Suspended activity entirely. By transition, the possible state/activity destinations from Claimed/Busy:

Claimed/Idle If the starter that is serving a given job exits (for example because the jobs completes), the machine will go to Claimed/Idle (transition **12**).

Claimed/Retiring If WANT_SUSPEND is FALSE and the PREEMPT expression is TRUE, the machine enters the Retiring activity (transition **13**). From there, it waits for a configurable amount of time for the job to finish before moving on to preemption.

Another reason the machine would go from Claimed/Busy to Claimed/Retiring is if the *condor_negotiator* matched the machine with a “better” match. This better match could either be from the machine’s perspective using the startd RANK expression, or it could be from the negotiator’s perspective due to a job with a higher user priority.

Another case resulting in a transition to Claimed/Retiring is when the startd is being shut down. The only exception is a “fast” shutdown, which bypasses retirement completely.

Claimed/Suspended If both the WANT_SUSPEND and SUSPEND expressions evaluate to TRUE, the machine suspends the job (transition **14**).

If a *condor_checkpoint* command arrives, or the PERIODIC_CHECKPOINT expression evaluates to TRUE, there is no state change. The startd has no way of knowing when this process completes, so periodic checkpointing can not be another state. Periodic checkpointing remains in the Claimed/Busy state and appears as a running job.

From the Claimed/Suspended state, the following transitions may occur:

Claimed/Busy If the CONTINUE expression evaluates to TRUE, the machine resumes the job and enters the Claimed/Busy state (transition **15**) or the Claimed/Retiring state (transition **16**), depending on whether the claim has been preempted.

Claimed/Retiring If the PREEMPT expression is TRUE, the machine will enter the Claimed/Retiring activity (transition **16**).

Preempting If the claim is in suspended retirement and the retirement time expires, the job enters the Preempting state (transition **17**). This is only possible if MaxJobRetirementTime *decreases* during the suspension.

For the Claimed/Retiring state, the following transitions may occur:

Preempting If the job finishes or the job’s run time exceeds MaxJobRetirementTime, the Preempting state is entered (transition **18**). The run time is computed from the time when the job was started by the startd minus any suspension time. (When retiring due to startd shutdown or restart, it is possible for the admin to issue a “peaceful” shutdown command, which causes MaxJobRetirementTime to effectively be infinite, avoiding any killing of jobs.)

Claimed/Busy If the startd was retiring because of a preempting claim only and the preempting claim goes away, the normal Claimed/Busy state is resumed (transition **19**). If instead the retirement is due to owner activity (PREEMPT) or the startd is being shut down, no unretirement is possible.

Claimed/Suspended In exactly the same way that suspension may happen from the Claimed/Busy state, it may also happen during the Claimed/Retiring state (transition **20**). In this case, when the job continues from suspension, it moves back into Claimed/Retiring (transition **16**) instead of Claimed/Busy (transition **15**).

Preempting State

The Preempting state is less complex than the Claimed state. There are two activities. Depending on the value of `WANT_VACATE`, a machine will be in the Vacating activity (if `TRUE`) or the Killing activity (if `FALSE`).

While in the Preempting state (regardless of activity) the machine advertises its `Requirements` expression as `FALSE` to signify that it is not available for further matches, either because it is about to transition to the Owner state, or because it has already been matched with one preempting match, and further preempting matches are disallowed until the machine has been claimed by the new match.

The main function of the Preempting state is to get rid of the starter associated with the resource. If the *condor_starter* associated with a given claim exits while the machine is still in the Vacating activity, then the job successfully completed a graceful shutdown. For standard universe jobs, this means that a checkpoint was saved. For other jobs, this means the application was given an opportunity to do a graceful shutdown, by intercepting the soft kill signal.

If the machine is in the Vacating activity, it keeps evaluating the `KILL` expression. As soon as this expression evaluates to `TRUE`, the machine enters the Killing activity (transition **21**).

When the starter exits, or if there was no starter running when the machine enters the Preempting state (transition **10**), the other purpose of the Preempting state is completed: notifying the schedd that had claimed this machine that the claim is broken.

At this point, the machine enters either the Owner state by transition **22** (if the job was preempted because the machine owner came back) or the Claimed/Idle state by transition **23** (if the job was preempted because a better match was found).

If the machine enters the Killing activity, (because either `WANT_VACATE` was `FALSE` or the `KILL` expression evaluated to `TRUE`), it attempts to force the *condor_starter* to immediately kill the underlying Condor job. Once the machine has begun to hard kill the Condor job, the *condor_startd* starts a timer, the length of which is defined by the `KILLING_TIMEOUT` macro. This macro is defined in seconds and defaults to 30. If this timer expires and the machine is still in the Killing activity, something has gone seriously wrong with the *condor_starter* and the startd tries to vacate the job immediately by sending `SIGKILL` to all of the *condor_starter*'s children, and then to the *condor_starter* itself.

Once the *condor_starter* has killed off all the processes associated with the job and exited, and once the schedd that had claimed the machine is notified that the claim is broken, the machine will leave the Preempting/Killing state. If the job was preempted because a better match was found, the machine will enter Claimed/Idle (transition **24**). If the preemption was caused by the machine owner

(the `PREEMPT` expression evaluated to `TRUE`, *condor_vacate* was used, etc), the machine will enter the Owner state (transition **25**).

Backfill State

The Backfill state is used whenever the machine is performing low priority background tasks to keep itself busy. For more information about backfill support in Condor, see section 3.12.9 on page 412. This state is only used if the machine has been configured to enable backfill computation, if a specific backfill manager has been installed and configured, and if the machine is otherwise idle (not being used interactively or for regular Condor computations). If the machine meets all these requirements, and the `START_BACKFILL` expression evaluates to `TRUE`, the machine will move from the Unclaimed/Idle state to Backfill/Idle (transition **7**).

Once a machine is in Backfill/Idle, it will immediately attempt to spawn whatever backfill manager it has been configured to use (currently, only the BOINC client is supported as a backfill manager in Condor). Once the BOINC client is running, the machine will enter Backfill/Busy (transition **26**) to indicate that it is now performing a backfill computation.

NOTE: On SMP machines, the *condor_startd* will only spawn a single instance of the BOINC client, even if multiple slots are available to run backfill jobs. Therefore, only the first machine to enter Backfill/Idle will cause a copy of the BOINC client to start running. If a given slot on an SMP enters the Backfill state and a BOINC client is already running under this *condor_startd*, the slot will immediately enter Backfill/Busy without waiting to spawn another copy of the BOINC client.

If the BOINC client ever exits on its own (which normally wouldn't happen), the machine will go back to Backfill/Idle (transition **27**) where it will immediately attempt to respawn the BOINC client (and return to Backfill/Busy via transition **26**).

As the BOINC client is running a backfill computation, a number of events can occur that will drive the machine out of the Backfill state. The machine can get matched or claimed for a Condor job, interactive users can start using the machine again, the machine might be evicted with *condor_vacate*, or the *condor_startd* might be shutdown. All of these events cause the *condor_startd* to kill the BOINC client and all its descendants, and enter the Backfill/Killing state (transition **28**).

Once the BOINC client and all its children have exited the system, the machine will enter the Backfill/Idle state to indicate that the BOINC client is now gone (transition **29**). As soon as it enters Backfill/Idle after the BOINC client exits, the machine will go into another state, depending on what caused the BOINC client to be killed in the first place.

If the `EVICT_BACKFILL` expression evaluates to `TRUE` while a machine is in Backfill/Busy, after the BOINC client is gone, the machine will go back into the Owner/Idle state (transition **30**). The machine will also return to the Owner/Idle state after the BOINC client exits if *condor_vacate* was used, or if the *condor_startd* is being shutdown.

When a machine running backfill jobs is matched with a requester that wants to run a Condor job, the machine will either enter the Matched state, or go directly into Claimed/Idle. As with the case of a machine in Unclaimed/Idle (described above), the *condor_negotiator* informs both the *con-*

dor_startd and the *condor_schedd* of the match, and the exact state transitions at the machine depend on what order the various entities initiate communication with each other. If the *condor_schedd* is notified of the match and sends a request to claim the *condor_startd* before the *condor_negotiator* has a chance to notify the *condor_startd*, once the BOINC client exits, the machine will immediately enter Claimed/Idle (transition **31**). Normally, the notification from the *condor_negotiator* will reach the *condor_startd* before the *condor_schedd* attempts to claim it. In this case, once the BOINC client exits, the machine will enter Matched/Idle (transition **32**).

3.5.8 State/Activity Transition Expression Summary

This section is a summary of the information from the previous sections. It serves as a quick reference.

START When TRUE, the machine is willing to spawn a remote Condor job.

RUNBENCHMARKS While in the Unclaimed state, the machine will run benchmarks whenever TRUE.

MATCHTIMEOUT If the machine has been in the Matched state longer than this value, it will transition to the Owner state.

WANT_SUSPEND If TRUE, the machine evaluates the SUSPEND expression to see if it should transition to the Suspended activity. If FALSE, the machine look at the PREEMPT expression.

SUSPEND If WANT_SUSPEND is TRUE, and the machine is in the Claimed/Busy state, it enters the Suspended activity if SUSPEND is TRUE.

CONTINUE If the machine is in the Claimed/Suspended state, it enter the Busy activity if CONTINUE is TRUE.

PREEMPT If the machine is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and WANT_SUSPEND is FALSE, the machine enters the Claimed/Retiring state whenever PREEMPT is TRUE.

CLAIMWORKLIFE If provided, this expression specifies the number of seconds during which a claim will continue accepting new jobs. Once this time expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. This may be useful if you want to force periodic renegotiation of resources without preemption having to occur. For example, if you have some low-priority jobs which should never be interrupted with kill signals, you could prevent them from being killed with `MaxJobRetirementTime`, but now high-priority jobs may have to wait in line when they match to a machine that is busy running one of these uninterruptible jobs. You can prevent the high-priority jobs from ever matching to such a machine by using a rank expression in the job or in the negotiator's rank expressions, but then the low-priority claim will never be interrupted; it can keep running more jobs. The solution is to use `CLAIMWORKLIFE` to force the claim to stop running additional jobs after a certain amount of time. The default value for `CLAIMWORKLIFE` is -1, which is treated as an infinite claim worklife, so claims may be held indefinitely (as long as they are not preempted and the schedd does not relinquish them, of course).

MAXJOBRETIREMENTTIME If the machine is in the Claimed/Retiring state, this expression specifies the maximum time (in seconds) that the startd will wait for the job to finish naturally (without any kill signals from the startd). The clock starts when the job is started and is paused during any suspension. The job may provide its own expression for `MaxJobRetirementTime`, but this can only be used to take *less* than the time granted by the startd, never more. (For convenience, standard universe and nice_user jobs are submitted with a default retirement time of 0, so they will never wait in retirement unless the user overrides the default.)

Once the job finishes or if the retirement time expires, the machine enters the Preempting state.

This expression is evaluated in the context of the job ClassAd, so it may refer to attributes of the current job as well as machine attributes. The expression is continually re-evaluated while the job is running, so it is possible, though unusual, to have an expression that changes over time. For example, if you want the retirement time to drop to 0 if an especially high priority job is waiting for the current job to retire, you could use `PreemptingRank` in the expression. Example:

```
MaxJobRetirementTime = 3600 * ( \
    MY.PreemptingRank != UNDEFINED || \
    PreemptingRank < 600 )
```

In this example, the retirement time is 3600 seconds, but if a job gets matched to this machine and it has a `PreemptingRank` of 600 or more, the retirement time drops to 0 and the current job is immediately preempted.

WANT_VACATE This is checked only when the `PREEMPT` expression is `TRUE` and the machine enters the Preempting state. If `WANT_VACATE` is `TRUE`, the machine enters the Vacating activity. If it is `FALSE`, the machine will proceed directly to the Killing activity.

KILL If the machine is in the Preempting/Vacating state, it enters Preempting/Killing whenever `KILL` is `TRUE`.

KILLING_TIMEOUT If the machine is in the Preempting/Killing state for longer than `KILLING_TIMEOUT` seconds, the startd sends a `SIGKILL` to the *condor_starter* and all its children to try to kill the job as quickly as possible.

PERIODIC_CHECKPOINT If the machine is in the Claimed/Busy state and `PERIODIC_CHECKPOINT` is `TRUE`, the user's job begins a periodic checkpoint.

RANK If this expression evaluates to a higher number for a pending resource request than it does for the current request, the machine preempts the current request (enters the Preempting/Vacating state). When the preemption is complete, the machine enters the Claimed/Idle state with the new resource request claiming it.

START_BACKFILL When `TRUE`, if the machine is otherwise idle, it will enter the Backfill state and spawn a backfill computation (using BOINC).

EVICT_BACKFILL When `TRUE`, if the machine is currently running a backfill computation, it will kill the BOINC client and return to the Owner/Idle state.

3.5.9 Policy Settings

This section describes the default configuration policy and then provides examples of extensions to these policies.

Default Policy Settings

These settings are the default as shipped with Condor. They have been used for many years with no problems. The vanilla expressions are identical to the regular ones. (They are not listed here. If not defined, the standard expressions are used for vanilla jobs as well).

The following are macros to help write the expressions clearly.

StateTimer Amount of time in the current state.

ActivityTimer Amount of time in the current activity.

ActivationTimer Amount of time the job has been running on this machine.

LastCkpt Amount of time since the last periodic checkpoint.

NonCondorLoadAvg The difference between the system load and the Condor load (the load generated by everything but Condor).

BackgroundLoad Amount of background load permitted on the machine and still start a Condor job.

HighLoad If the $\$(NonCondorLoadAvg)$ goes over this, the CPU is considered too busy, and eviction of the Condor job should start.

StartIdleTime Amount of time the keyboard must to be idle before Condor will start a job.

ContinueIdleTime Amount of time the keyboard must to be idle before resumption of a suspended job.

MaxSuspendTime Amount of time a job may be suspended before more drastic measures are taken.

MaxVacateTime Amount of time a job may be checkpointing before we give up and kill it outright.

KeyboardBusy A boolean expression that evaluates to TRUE when the keyboard is being used.

CPUIIdle A boolean expression that evaluates to TRUE when the CPU is idle.

CPUBusy A boolean expression that evaluates to TRUE when the CPU is busy.

MachineBusy The CPU or the Keyboard is busy.

CPUIsBusy A boolean value set to the same value as CPUBusy.

CPUBusyTime The value 0 if CPUBusy is False; the time in seconds since CPUBusy became True.

```
## These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg      = (LoadAvg - CondorLoadAvg)
BackgroundLoad        = 0.3
HighLoad              = 0.5
StartIdleTime         = 15 * $(MINUTE)
ContinueIdleTime      = 5 * $(MINUTE)
MaxSuspendTime        = 10 * $(MINUTE)
MaxVacateTime         = 10 * $(MINUTE)

KeyboardBusy          = KeyboardIdle < $(MINUTE)
ConsoleBusy           = (ConsoleIdle < $(MINUTE))
CPUIdle               = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy               = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy       = ($(KeyboardBusy) == False)
MachineBusy           = ($(CPUBusy) || $(KeyboardBusy))
```

Macros are defined to want to suspend jobs (instead of killing them) in the case of jobs that use little memory, when the keyboard is not being used, and for vanilla universe jobs. We want to gracefully vacate jobs which have been running for more than 10 minutes or are vanilla universe jobs.

```
WANT_SUSPEND      = ( $(SmallJob) || $(KeyboardNotBusy) \
                      || $(IsVanilla) )
WANT_VACATE       = ( $(ActivationTimer) > 10 * $(MINUTE) \
                      || $(IsVanilla) )
```

Finally, definitions of the actual expressions. Start a job if the keyboard has been idle long enough and the load average is low enough OR the machine is currently running a Condor job. Note that Condor would only run one job at a time. It just may prefer to run a different job, as defined by the machine rank or user priorities.

```
START             = ( (KeyboardIdle > $(StartIdleTime)) \
                      && ( $(CPUIdle) || \
                          (State != "Unclaimed" && State != "Owner")) )
```

Suspend a job if the keyboard has been touched. Alternatively, suspend if the CPU has been busy for more than two minutes and the job has been running for more than 90 seconds.

```
SUSPEND          = ( $(KeyboardBusy) || \
                    ( (CpuBusyTime > 2 * $(MINUTE)) \
                      && $(ActivationTimer) > 90 ) )
```

Continue a suspended job if the CPU is idle, the Keyboard has been idle for long enough, and the job has been suspended more than 10 seconds.

```
CONTINUE         = ( $(CPUIidle) && ($(ActivityTimer) > 10) \
                    && (KeyboardIdle > $(ContinueIdleTime)) )
```

There are two conditions that signal preemption. The first condition is if the job is suspended, but it has been suspended too long. The second condition is if suspension is not desired and the machine is busy.

```
PREEMPT         = ( ((Activity == "Suspended") && \
                    ($(ActivityTimer) > $(MaxSuspendTime))) \
                    || (SUSPEND && (WANT_SUSPEND == False)) )
```

Do not give jobs any time to retire on their own when they are about to be preempted.

```
MaxJobRetirementTime = 0
```

Kill jobs that take too long leaving gracefully.

```
KILL            = $(ActivityTimer) > $(MaxVacateTime)
```

Finally, specify periodic checkpointing. For jobs smaller than 60 Mbytes, do a periodic checkpoint every 6 hours. For larger jobs, only checkpoint every 12 hours.

```
PERIODIC_CHECKPOINT = ( (ImageSize < 60000) && \
                        ($(LastCkpt) > (6 * $(HOUR))) ) || \
                        ( $(LastCkpt) > (12 * $(HOUR)) )
```

At UW-Madison, we have a fast network. We simplify our expression considerably to

```
PERIODIC_CHECKPOINT = $(LastCkpt) > (3 * $(HOUR))
```

For reference, the entire set of policy settings are included once more without comments:

```

## These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer    = (CurrentTime - EnteredCurrentActivity)
ActivationTimer  = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg      = (LoadAvg - CondorLoadAvg)
BackgroundLoad        = 0.3
HighLoad              = 0.5
StartIdleTime         = 15 * $(MINUTE)
ContinueIdleTime      = 5 * $(MINUTE)
MaxSuspendTime        = 10 * $(MINUTE)
MaxVacateTime         = 10 * $(MINUTE)

KeyboardBusy          = KeyboardIdle < $(MINUTE)
ConsoleBusy           = (ConsoleIdle < $(MINUTE))
CPUIidle              = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy               = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy        = ($(KeyboardBusy) == False)
MachineBusy           = ($(CPUBusy) || $(KeyboardBusy))

WANT_SUSPEND          = ( $(SmallJob) || $(KeyboardNotBusy) \
                          || $(IsVanilla) )
WANT_VACATE           = ( $(ActivationTimer) > 10 * $(MINUTE) \
                          || $(IsVanilla) )
START                 = ( (KeyboardIdle > $(StartIdleTime)) \
                          && ( $(CPUIidle) || \
                              (State != "Unclaimed" && State != "Owner")) )
SUSPEND               = ( $(KeyboardBusy) || \
                          ( (CpuBusyTime > 2 * $(MINUTE)) \
                            && $(ActivationTimer) > 90 ) )
CONTINUE              = ( $(CPUIidle) && ($(ActivityTimer) > 10) \
                          && (KeyboardIdle > $(ContinueIdleTime)) )
PREEMPT               = ( ((Activity == "Suspended") && \
                          ($(ActivityTimer) > $(MaxSuspendTime))) \
                          || (SUSPEND && (WANT_SUSPEND == False)) )
MaxJobRetirementTime = 0
KILL                  = $(ActivityTimer) > $(MaxVacateTime)
PERIODIC_CHECKPOINT    = ( (ImageSize < 60000) && \
                          ( $(LastCkpt) > (6 * $(HOUR)) ) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )

```

Test-job Policy Example

This example shows how the default macros can be used to set up a machine for running test jobs from a specific user. Suppose we want the machine to behave normally, except if user coltrane submits a job. In that case, we want that job to start regardless of what is happening on the machine. We do not want the job suspended, vacated or killed. This is reasonable if we know coltrane is submitting very short running programs for testing purposes. The jobs should be executed right away. This works with any machine (or the whole pool, for that matter) by adding the following 5 expressions to the existing configuration:

```
START      = ( $(START) ) || Owner == "coltrane"
SUSPEND    = ( $(SUSPEND) ) && Owner != "coltrane"
CONTINUE   = $(CONTINUE)
PREEMPT    = ( $(PREEMPT) ) && Owner != "coltrane"
KILL       = $(KILL)
```

Notice that there is nothing special in either the CONTINUE or KILL expressions. If Coltrane's jobs never suspend, they never look at CONTINUE. Similarly, if they never preempt, they never look at KILL.

Time of Day Policy

Condor can be configured to only run jobs at certain times of the day. In general, we discourage configuring a system like this, since you can often get lots of good cycles out of machines, even when their owners say "I'm always using my machine during the day." However, if you submit mostly vanilla jobs or other jobs that cannot checkpoint, it might be a good idea to only allow the jobs to run when you know the machines will be idle and when they will not be interrupted.

To configure this kind of policy, you should use the `ClockMin` and `ClockDay` attributes, defined in section 3.5.1 on "Startd ClassAd Attributes". These are special attributes which are automatically inserted by the *condor_startd* into its ClassAd, so you can always reference them in your policy expressions. `ClockMin` defines the number of minutes that have passed since midnight. For example, 8:00am is 8 hours after midnight, or 8 * 60 minutes, or 480. 5:00pm is 17 hours after midnight, or 17 * 60, or 1020. `ClockDay` defines the day of the week, Sunday = 0, Monday = 1, and so on.

To make the policy expressions easy to read, we recommend using macros to define the time periods when you want jobs to run or not run. For example, assume regular "work hours" at your site are from 8:00am until 5:00pm, Monday through Friday:

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
              (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
              (ClockDay == 0 || ClockDay == 6) )
```

Of course, you can fine-tune these settings by changing the definition of `AfterHours` and `WorkHours` for your site.

Assuming you are using the default policy expressions discussed above, there are only a few minor changes required to force Condor jobs to stay off of your machines during work hours:

```
# Only start jobs after hours.
START = $(AfterHours) && $(CPUIIdle) && KeyboardIdle > $(StartIdleTime)

# Consider the machine busy during work hours, or if the keyboard or
# CPU are busy.
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

By default, the `MachineBusy` macro is used to define the `SUSPEND` and `PREEMPT` expressions. If you have changed these expressions at your site, you will need to add `$(WorkHours)` to your `SUSPEND` and `PREEMPT` expressions as appropriate.

Depending on your site, you might also want to avoid suspending jobs during work hours, so that in the morning, if a job is running, it will be immediately preempted, instead of being suspended for some length of time:

```
WANT_SUSPEND = $(AfterHours)
```

Desktop/Non-Desktop Policy

Suppose you have two classes of machines in your pool: desktop machines and dedicated cluster machines. In this case, you might not want keyboard activity to have any effect on the dedicated machines. For example, when you log into these machines to debug some problem, you probably do not want a running job to suddenly be killed. Desktop machines, on the other hand, should do whatever is necessary to remain responsive to the user.

There are many ways to achieve the desired behavior. One way is to make a standard desktop policy and a standard non-desktop policy and to copy the desired one into the local configuration file for each machine. Another way is to define one standard policy (in `condor_config`) with a simple toggle that can be set in the local configuration file. The following example illustrates the latter approach.

For ease of use, an entire policy is included in this example. Some of the expressions are just the usual default settings.

```
# If "IsDesktop" is configured, make it an attribute of the machine ClassAd.
STARTD_ATTRS = IsDesktop

# Only consider starting jobs if:
```

```

# 1) the load average is low enough OR the machine is currently
#    running a Condor job
# 2) AND the user is not active (if a desktop)
START = ( ($(CPUIidle) || (State != "Unclaimed" && State != "Owner")) \
          && (IsDesktop != True || (KeyboardIdle > $(StartIdleTime))) )

# Suspend (instead of vacating/killing) for the following cases:
WANT_SUSPEND = ( $(SmallJob) || $(JustCpu) \
                 || $(IsVanilla) )

# When preempting, vacate (instead of killing) in the following cases:
WANT_VACATE = ( $(ActivationTimer) > 10 * $(MINUTE) \
                || $(IsVanilla) )

# Suspend jobs if:
# 1) The CPU has been busy for more than 2 minutes, AND
# 2) the job has been running for more than 90 seconds
# 3) OR suspend if this is a desktop and the user is active
SUSPEND = ( ((CpuBusyTime > 2 * $(MINUTE)) && ($(ActivationTimer) > 90)) \
            || ( IsDesktop == True && $(KeyboardBusy) ) )

# Continue jobs if:
# 1) the CPU is idle, AND
# 2) we've been suspended more than 5 minutes AND
# 3) the keyboard has been idle for long enough (if this is a desktop)
CONTINUE = ( $(CPUIidle) && ($(ActivityTimer) > 300) \
             && (IsDesktop != True || (KeyboardIdle > $(ContinueIdleTime))) )

# Preempt jobs if:
# 1) The job is suspended and has been suspended longer than we want
# 2) OR, we don't want to suspend this job, but the conditions to
#    suspend jobs have been met (someone is using the machine)
PREEMPT = ( ((Activity == "Suspended") && \
              ($(ActivityTimer) > $(MaxSuspendTime))) \
            || (SUSPEND && (WANT_SUSPEND == False)) )

# Replace 0 in the following expression with whatever amount of
# retirement time you want dedicated machines to provide. The other part
# of the expression forces the whole expression to 0 on desktop
# machines.
MaxJobRetirementTime = (IsDesktop != True) * 0

# Kill jobs if they have taken too long to vacate gracefully
KILL = $(ActivityTimer) > $(MaxVacateTime)

```

With this policy in *condor_config*, the local configuration files for desktops can be easily configured with the following line:

```
IsDesktop = True
```

In all other cases, the default policy described above will ignore keyboard activity.

Disabling Preemption

Preemption can result in jobs being killed by Condor. When this happens, the jobs remain in the queue and will be automatically rescheduled. We highly recommend designing jobs that work well in this environment, rather than simply disabling preemption.

Planning for preemption makes jobs more robust in the face of other sources of failure. One way to live happily with preemption is to use Condor's standard universe, which provides the ability to produce checkpoints. If a job is incompatible with the requirements of standard universe, the job can still gracefully shutdown and restart by intercepting the soft kill signal.

All that being said, there may be cases where it is appropriate to force Condor to never kill jobs within some upper time limit. This can be achieved with the following policy in the configuration of the execute nodes:

```
# When we want to kick a job off, let it run uninterrupted for  
# up to 2 days before forcing it to vacate.  
MAXJOBRETIREMENTTIME = $(HOUR) * 24 * 2
```

Construction of this expression may be more complicated. For example, it could provide a different retirement time to different users or different types of jobs. Also be aware that the job may come with its own definition of *MaxJobRetirementTime*, but this may only cause *less* retirement time to be used, never more than what the machine offers.

The longer the retirement time that is given, the slower reallocation of resources in the pool can become if there are long-running jobs. However, by preventing jobs from being killed, you may decrease the number of cycles that are wasted on non-checkpointable jobs that are killed. That is the basic trade off.

Note that the use of *MAXJOBRETIREMENTTIME* limits the killing of jobs, but it does not prevent the preemption of resource claims. Therefore, it is technically not a way of disabling preemption, but simply a way of forcing preempting claims to wait until an existing job finishes or runs out of time. In other words, it limits the preemption of jobs but not the preemption of claims.

Limiting the preemption of jobs is often more desirable than limiting the preemption of resource claims. However, if you really do want to limit the preemption of resource claims, the following policy may be used. Some of these settings apply to the execute node and some apply to the central manager, so this policy should be configured so that it is read by both.

```
#Disable preemption by machine activity.
PREEMPT = False
#Disable preemption by user priority.
PREEMPTION_REQUIREMENTS = False
#Disable preemption by machine RANK by ranking all jobs equally.
RANK = 0
#Since we are disabling claim preemption, we
# may as well optimize negotiation for this case:
NEGOTIATOR_CONSIDER_PREEMPTION = False
```

Be aware of the consequences of this policy. Without any preemption of resource claims, once the *condor_negotiator* gives the *condor_schedd* a match to a machine, the *condor_schedd* may hold onto this claim indefinitely, as long as the user keeps supplying more jobs to run. If this is not desired, force claims to be retired after some amount of time using `CLAIM_WORKLIFE`. This enforces a time limit, beyond which no new jobs may be started on an existing claim; therefore the *condor_schedd* daemon is forced to go back to the *condor_negotiator* to request a new match, if there is still more work to do. Example execute machine configuration to include in addition to the example above:

```
# after 20 minutes, schedd must renegotiate to run
# additional jobs on the machine
CLAIM_WORKLIFE = 1200
```

Also be aware that in all versions of Condor prior to 6.8.1, it is not advisable to set `NEGOTIATOR_CONSIDER_PREEMPTION` to False, because of a bug that can lead to some machines never being matched to jobs.

Job Suspension

As new jobs are submitted that receive a higher priority than currently executing jobs, the executing jobs may be preempted. These jobs lose whatever forward progress they have made, and are sent back to the job queue to await starting over again as another machine becomes available.

Condor may be configured with a policy that allows these potentially evicted jobs to be suspended instead. The policy utilizes two slots, one (called slot1 in the example) that only runs jobs identified as high priority jobs. The second slot (called slot2 in the example) is set to run jobs according to the usual policy and to suspend them when slot1 is claimed. A policy for a machine with more than one physical CPU may be adapted from this example. Instead of having 2 slots, you would have 2 times the number of physical CPUs. Half of the slots would be for high priority jobs and the other half would be for suspendable jobs.

```
# Lie to Condor, to achieve 2 slots with only a single CPU
NUM_CPUS = 2

# slot1 is the high-prio slot, while slot2 is the background slot...
START = (SlotID == 1) && $(SLOT1_START) || \
        (SlotID == 2) && $(SLOT2_START)

# Only start jobs on slot1 if the job is marked as a high-priority job
```

```

SLOT1_START = (TARGET.IsHighPrioJob == TRUE)

# Only start jobs on slot2 if there is no job on slot1, and if the
# machine is otherwise idle... NOTE: the "Busy" activity is only in
# the Claimed state, and only when there is an active job, so that is
# good enough for our needs...
SLOT2_START = ( (slot1_Activity != "Busy") && \
                (KeyboardIdle > $(StartIdleTime)) && \
                ($(CPUIIdle) || (State != "Unclaimed" && State != "Owner")) )

# Only suspend jobs on slot2. Suspend if there is keyboard activity or
# if a job starts on slot1...
SUSPEND = (SlotID == 2) && \
          ( (slot1_Activity == "Busy") || ($(KeyboardBusy)) )

CONTINUE = (SlotID == 2) && \
           (KeyboardIdle > $(ContinueIdleTime)) && \
           (slot1_Activity != "Busy")

```

Note that in this example, the job ClassAd attribute `IsHighPrioJob` has no special meaning to Condor. It is an invented name chosen for this example. To take advantage of the policy, a user must submit high priority jobs with this attribute defined. The following line appears in the job's submit description file as

```
+IsHighPrioJob = True
```

3.6 Security

Security in Condor is a broad issue, with many aspects to consider. Because Condor's main purpose is to allow users to run arbitrary code on large numbers of computers, it is important to try to limit who can access a Condor pool and what privileges they have when using the pool. This section covers these topics.

There is a distinction between the kinds of resource attacks Condor can defeat, and the kinds of attacks Condor cannot defeat. Condor cannot prevent security breaches of users that can elevate their privilege to the root or administrator account. Condor does not run user jobs in sandboxes (standard universe jobs are a partial exception to this), so Condor cannot defeat all malicious actions by user jobs. An example of a malicious job is one that launches a distributed denial of service attack. Condor assumes that users are trustworthy. Condor can prevent unauthorized access to the Condor pool, to help ensure that only trusted users have access to the pool. In addition, Condor provides encryption and integrity checking, to ensure that data (both Condor's data and user jobs' data) has not been examined or tampered with while in transit.

Broadly speaking, the aspects of security in Condor may be categorized and described:

Users Authorization or capability in an operating system is based on a process owner. Both those that submit jobs and Condor daemons become process owners. The Condor system prefers

that Condor daemons are run as the user `root`, while other common operations are owned by a user of Condor. Operations that do not belong to either `root` or a Condor user are often owned by the `condor` user. See Section 3.6.11 for more detail.

Authentication Proper identification of a user is accomplished by the process of authentication. It attempts to distinguish between real users and impostors. By default, Condor's authentication uses the user id (UID) to determine identity, but Condor can choose among a variety of authentication mechanisms, including the stronger authentication methods Kerberos and GSI.

Authorization Authorization specifies who is allowed to do what. Some users are allowed to submit jobs, while other users are allowed administrative privileges over Condor itself. Condor provides authorization on either a per-user or on a per-machine basis.

Privacy Condor may encrypt data sent across the network, which prevents others from viewing the data. With persistence and sufficient computing power, decryption is possible. Condor can encrypt the data sent for internal communication, as well as user data, such as files and executables. Encryption operates on network transmissions: unencrypted data is stored on disk.

Integrity The *man-in-the-middle* attack tampers with data without the awareness of either side of the communication. Condor's integrity check sends additional cryptographic data to verify that network data transmissions have not been tampered with. Note that the integrity information is only for network transmissions: data stored on disk does not have this integrity information.

3.6.1 Condor's Security Model

At the heart of Condor's security model is the notion that communications are subject to various security checks. A request from one Condor daemon to another may require authentication to prevent subversion of the system. A request from a user of Condor may need to be denied due to the confidential nature of the request. The security model handles these example situations and many more.

Requests to Condor are categorized into groups of *access levels*, based on the type of operation requested. The user of a specific request must be authorized at the required access level. For example, executing the `condor_status` command requires the `READ` access level. Actions that accomplish management tasks, such as shutting down or restarting of a daemon require an `ADMINISTRATOR` access level. See Section 3.6.7 for a full list of Condor's access levels and their meanings.

There are two sides to any communication or command invocation in Condor. One side is identified as the *client*, and the other side is identified as the *daemon*. The client is the party that initiates the command, and the daemon is the party that processes the command and responds. In some cases it is easy to distinguish the client from the daemon, while in other cases it is not as easy. Condor tools such as `condor_submit` and `condor_config_val` are clients. They send commands to daemons and act as clients in all their communications. For example, the `condor_submit` command communicates with the `condor_schedd`. Behind the scenes, Condor daemons also communicate with each other; in this case the daemon initiating the command plays the role of the client. For instance,

the *condor_negotiator* daemon acts as a client when contacting the *condor_schedd* daemon to initiate matchmaking. Once a match has been found, the *condor_schedd* daemon acts as a client and contacts the *condor_startd* daemon.

Condor's security model is implemented using configuration. Commands in Condor are executed over TCP/IP network connections. While network communication enables Condor to manage resources that are distributed across an organization (or beyond), it also brings in security challenges. Condor must have ways of ensuring that commands are being sent by trustworthy users. Jobs that are operating on sensitive data must be allowed to use encryption such that the data is not seen by outsiders. Jobs may need assurance that data has not been tampered with. These issues can be addressed with Condor's authentication, encryption, and integrity features.

Access Level Descriptions

Authorization is granted based on specified access levels. This list describes each access level, and provides examples of their usage. The levels implement a partial hierarchy; a higher level often implies a READ or both a WRITE and a READ level of access as described.

READ This access level can obtain or read information about Condor. Examples that require only READ access are viewing the status of the pool with *condor_status*, checking a job queue with *condor_q*, or viewing user priorities with *condor_userprio*. READ access does not allow any changes, and it does not allow job submission.

WRITE This access level is required to send (write) information to Condor. Examples that require WRITE access are job submission with *condor_submit* and advertising a machine so it appears in the pool (this is usually done automatically by the *condor_startd* daemon). The WRITE level of access implies READ access.

ADMINISTRATOR This access level has additional Condor administrator rights to the pool. It includes the ability to change user priorities (with the command *condor_userprio -set*), as well as the ability to turn Condor on and off (as with the commands *condor_on* and *condor_off*). The ADMINISTRATOR level of access implies both READ and WRITE access.

SOAP This access level is required for the authorization of any party that will use the Web Services (SOAP) interface to Condor. It is not a general access level to be used with the variety of configuration variables for authentication, encryption, and integrity checks.

CONFIG This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, this level of access can change any configuration parameters of a Condor pool, except those specified in the *condor_config.root* configuration file. The CONFIG level of access implies READ access.

OWNER This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. An example that requires the OWNER access level is the *condor_vacate* command. The command causes the *condor_startd* daemon to vacate any Condor job currently running on a machine. The owner of that machine should be able to cause the removal of a job running on the machine.

DAEMON This access level is used for commands that are internal to the operation of Condor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon (which may be more specifically controlled by the `ADVERTISE_STARTD` access level). Authorization at this access level should only be given to the user account under which the Condor daemons run. The **DAEMON** level of access implies both `READ` and `WRITE` access. Any setting for this access level that is not defined will default to the corresponding setting in the `WRITE` access level.

NEGOTIATOR This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run. The **NEGOTIATOR** level of access implies `READ` access.

ADVERTISEMASTER This access level is used specifically for commands used to advertise a *condor_master* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the **DAEMON** access level.

ADVERTISESTARTD This access level is used specifically for commands used to advertise a *condor_startd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the **DAEMON** access level.

ADVERTISESCHEDD This access level is used specifically for commands used to advertise a *condor_schedd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the **DAEMON** access level.

CLIENT This access level is different from all the others. Whereas all of the other access levels refer to the security policy for accepting connections *from* others, the **CLIENT** access level applies when a Condor daemon or tool is connecting *to* some other Condor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

3.6.2 Security Negotiation

Because of the wide range of environments and security demands necessary, Condor must be flexible. Configuration provides this flexibility. The process by which Condor determines the security settings that will be used when a connection is established is called *security negotiation*. Security negotiation's primary purpose is to determine which of the features of authentication, encryption, and integrity checking will be enabled for a connection. In addition, since Condor supports multiple technologies for authentication and encryption, security negotiation also determines which technology is chosen for the connection.

Security negotiation is a completely separate process from matchmaking, and should not be confused with any specific function of the *condor_negotiator* daemon. Security negotiation occurs when one Condor daemon or tool initiates communication with another Condor daemon, to determine the security settings by which the communication will be ruled. The *condor_negotiator* daemon does

negotiation, whereby queued jobs and available machines within a pool go through the process of matchmaking (deciding out which machines will run which jobs).

Configuration

The configuration macro names that determine what features will be used during client-daemon communication follow the pattern:

```
SEC_<context>_<feature>
```

The <feature> portion of the macro name determines which security feature's policy is being set. <feature> may be any one of

```
AUTHENTICATION  
ENCRYPTION  
INTEGRITY  
NEGOTIATION
```

The <context> component of the security policy macros can be used to craft a fine-grained security policy based on the type of communication taking place. <context> may be any one of

```
CLIENT  
READ  
WRITE  
ADMINISTRATOR  
CONFIG  
OWNER  
DAEMON  
NEGOTIATOR  
ADVERTISE_MASTER  
ADVERTISE_STARTD  
ADVERTISE_SCHEDD  
DEFAULT
```

Any of these constructed configuration macros may be set to any of the following values:

```
REQUIRED  
PREFERRED  
OPTIONAL  
NEVER
```

Security negotiation resolves various client-daemon combinations of desired security features in order to set a policy.

		Daemon Setting		
		NEVER	OPTIONAL	REQUIRED
Client Setting	NEVER	No	No	Fail
	REQUIRED	Fail	Yes	Yes

Table 3.1: Resolution of security negotiation.

As an example, consider Frida the scientist. Frida wants to avoid authentication when possible. She sets

```
SEC_DEFAULT_AUTHENTICATION = OPTIONAL
```

The machine running the *condor_schedd* to which Frida will remotely submit jobs, however, is operated by a security-conscious system administrator who dutifully sets:

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

When Frida submits her jobs, Condor's security negotiation determines that authentication will be used, and allows the command to continue. This example illustrates the point that the most restrictive security policy sets the levels of security enforced. There is actually more to the understanding of this scenario. Some Condor commands, such as the use of *condor_submit* to submit jobs *always* require authentication of the submitter, no matter what the policy says. This is because the identity of the submitter needs to be known in order to carry out the operation. Others commands, such as *condor_q*, do not always require authentication, so in the above example, the server's policy would force Frida's *condor_q* queries to be authenticated, whereas a different policy could allow *condor_q* to happen without any authentication.

Whether or not security negotiation occurs depends on the setting at both the client and daemon side of the configuration variable(s) defined by `SEC_*_NEGOTIATION`. `SEC_DEFAULT_NEGOTIATION` is a variable representing the entire set of configuration variables for `NEGOTIATION`. For the client side setting, the only definitions that make sense are `REQUIRED` and `NEVER`. For the daemon side setting, the `PREFERRED` value makes no sense. Table 3.1 shows how security negotiation resolves various client-daemon combinations of security negotiation policy settings. Within the table, Yes means the security negotiation will take place. No means it will not. Fail means that the policy settings are incompatible and the communication cannot continue.

Enabling authentication, encryption, and integrity checks is dependent on security negotiation taking place. The enabled security negotiation further sets the policy for these other features. Table 3.2 shows how security features are resolved for client-daemon combinations of security feature policy settings. Like Table 3.1, Yes means the feature will be utilized. No means it will not. Fail implies incompatibility and the feature cannot be resolved.

The enabling of encryption and/or integrity checks is dependent on authentication taking place. The authentication provides a key exchange. The key is needed for both encryption and integrity checks.

		Daemon Setting			
		NEVER	OPTIONAL	PREFERRED	REQUIRED
Client Setting	NEVER	No	No	No	Fail
	OPTIONAL	No	No	Yes	Yes
	PREFERRED	No	Yes	Yes	Yes
	REQUIRED	Fail	Yes	Yes	Yes

Table 3.2: Resolution of security features.

Setting `SEC_CLIENT_<feature>` determines the policy for all outgoing commands. The policy for incoming commands (the daemon side of the communication) takes a more fine-grained approach that implements a set of access levels for the received command. For example, it is desirable to have all incoming administrative requests require authentication. Inquiries on pool status may not be so restrictive. To implement this, the administrator configures the policy:

```
SEC_ADMINISTRATOR_AUTHENTICATION = REQUIRED
SEC_READ_AUTHENTICATION           = OPTIONAL
```

The `DEFAULT` value for `<context>` provides a way to set a policy for all access levels (`READ`, `WRITE`, etc.) that do not have a specific configuration variable defined. In addition, some access levels will default to the settings specified for other access levels. For example, `ADVERTISE_STARTD` defaults to `DAEMON`, and `DAEMON` defaults to `WRITE`, which then defaults to the general `DEFAULT` setting.

Configuration for Security Methods

Authentication and encryption can each be accomplished by a variety of methods or technologies. Which method is utilized is determined during security negotiation.

The configuration macros that determine the methods to use for authentication and/or encryption are

```
SEC_<context>_AUTHENTICATION_METHODS
SEC_<context>_CRYPTO_METHODS
```

These macros are defined by a comma or space delimited list of possible methods to use. Section 3.6.3 lists all implemented authentication methods. Section 3.6.5 lists all implemented encryption methods.

3.6.3 Authentication

The client side of any communication uses one of two macros to specify whether authentication is to occur:

```
SEC_DEFAULT_AUTHENTICATION
SEC_CLIENT_AUTHENTICATION
```

For the daemon side, there are a larger number of macros to specify whether authentication is to take place, based upon the necessary access level:

```
SEC_DEFAULT_AUTHENTICATION
SEC_READ_AUTHENTICATION
SEC_WRITE_AUTHENTICATION
SEC_ADMINISTRATOR_AUTHENTICATION
SEC_CONFIG_AUTHENTICATION
SEC_OWNER_AUTHENTICATION
SEC_DAEMON_AUTHENTICATION
SEC_NEGOTIATOR_AUTHENTICATION
SEC_ADVERTISE_MASTER_AUTHENTICATION
SEC_ADVERTISE_STARTD_AUTHENTICATION
SEC_ADVERTISE_SCHEDD_AUTHENTICATION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

signifies that the daemon must authenticate the client for any communication that requires the WRITE access level. If the daemon's configuration contains

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

and does not contain any other security configuration for AUTHENTICATION, then this default defines the daemon's needs for authentication over all access levels. Where a specific macro is defined, the more specific value takes precedence over the default definition.

If authentication is to be done, then the communicating parties must negotiate a mutually acceptable method of authentication to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_CLIENT_AUTHENTICATION_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_READ_AUTHENTICATION_METHODS
SEC_WRITE_AUTHENTICATION_METHODS
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS
```

```
SEC_CONFIG_AUTHENTICATION_METHODS
SEC_OWNER_AUTHENTICATION_METHODS
SEC_DAEMON_AUTHENTICATION_METHODS
SEC_NEGOTIATOR_AUTHENTICATION_METHODS
SEC_ADVERTISE_MASTER_AUTHENTICATION_METHODS
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS
SEC_ADVERTISE_SCHEDD_AUTHENTICATION_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the authentication methods that are available to be used. The ordering of the list defines preference; the first item in the list indicates the highest preference. Defined values are

```
GSI
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
NTSSPI
CLAIMTOBE
ANONYMOUS
```

For example, a client may be configured with:

```
SEC_CLIENT_AUTHENTICATION_METHODS = FS, GSI
```

and a daemon the client is trying to contact with:

```
SEC_DEFAULT_AUTHENTICATION_METHODS = GSI
```

Security negotiation will determine that GSI authentication is the only compatible choice. If there are multiple compatible authentication methods, security negotiation will make a list of acceptable methods and they will be tried in order until one succeeds.

As another example, the macro

```
SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS, NTSSPI
```

indicates that either Kerberos or Windows authentication may be used, but Kerberos is preferred over Windows. Note that if the client and daemon agree that multiple authentication methods may be used, then they are tried in turn. For instance, if they both agree that Kerberos or NTSSPI may be used, then Kerberos will be tried first, and if there is a failure for any reason, then NTSSPI will be tried.

An additional specialized method of authentication exists for communication between the *condor_schedd* and *condor_startd*. It is especially useful when operating at large scale over high latency networks or in situations where it is inconvenient to set up one of the other methods of strong authentication between the submit and execute daemons. See the description of `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION` on 231 for details.

If the configuration for a machine does not define any variable for `SEC_<access-level>_AUTHENTICATION`, then Condor uses a default value of `OPTIONAL`. Authentication will be required for any operation which modifies the job queue, such as *condor_qedit* and *condor_rm*. If the configuration for a machine does not define any variable for `SEC_<access-level>_AUTHENTICATION_METHODS`, the default value for a Unix machine is `FS, KERBEROS, GSI`. This default value for a Windows machine is `NTSSPI, KERBEROS, GSI`.

GSI Authentication

The GSI (Grid Security Infrastructure) protocol provides an avenue for Condor to do PKI-based (Public Key Infrastructure) authentication using X.509 certificates. The basics of GSI are well-documented elsewhere, such as <http://www.globus.org/>.

A simple introduction to this type of authentication defines Condor's use of terminology, and it illuminates the needed items that Condor must access to do this authentication. Assume that A authenticates to B. In this example, A is the client, and B is the daemon within their communication. This example's one-way authentication implies that B is verifying the identity of A, using the certificate A provides, and utilizing B's own set of trusted CAs (Certification Authorities). Client A provides its certificate (or proxy) to daemon B. B does two things: B checks that the certificate is valid, and B checks to see that the CA that signed A's certificate is one that B trusts.

For the GSI authentication protocol, an X.509 certificate is required. Files with predetermined names hold a certificate, a key, and optionally, a proxy. A separate directory has one or more files that become the list of trusted CAs.

Allowing Condor to do this GSI authentication requires knowledge of the locations of the client A's certificate and the daemon B's list of trusted CAs. When one side of the communication (as either client A or daemon B) is a Condor daemon, these locations are determined by configuration or by default locations. When one side of the communication (as a client A) is a user of Condor (the process owner of a Condor tool, for example *condor_submit*), these locations are determined by the pre-set values of environment variables or by default locations.

GSI certificate locations for Condor daemons For a Condor daemon, the certificate may be a single host certificate, and all Condor daemons on the same machine may share the same certificate. In some cases, the certificate can also be copied to other machines, where local copies are necessary. This may occur only in cases where a single host certificate can match multiple host names, something that is beyond the scope of this manual. The certificates must be protected by access rights to files, since the password file is not encrypted.

The specification of the location of the necessary files through configuration uses the following precedence.

1. Configuration variable `GSI_DAEMON_DIRECTORY` gives the complete path name to the directory that contains the certificate, key, and directory with trusted CAs. Condor uses this directory as follows in its construction of the following configuration variables:

```
GSI_DAEMON_CERT      = $(GSI_DAEMON_DIRECTORY)/hostcert.pem
GSI_DAEMON_KEY       = $(GSI_DAEMON_DIRECTORY)/hostkey.pem
GSI_DAEMON_TRUSTED_CA_DIR = $(GSI_DAEMON_DIRECTORY)/certificates
```

Note that no proxy is assumed in this case.

2. If the `GSI_DAEMON_DIRECTORY` is not defined, or when defined, the location may be overridden with specific configuration variables that specify the complete path and file name of the certificate with

```
GSI_DAEMON_CERT
```

the key with

```
GSI_DAEMON_KEY
```

a proxy with

```
GSI_DAEMON_PROXY
```

the complete path to the directory containing the list of trusted CAs with

```
GSI_DAEMON_TRUSTED_CA_DIR
```

3. The default location assumed is `/etc/grid-security`. Note that this implemented by setting the value of `GSI_DAEMON_DIRECTORY`.

When a daemon acts as the client within authentication, the daemon needs a listing of those from which it will accept certificates. This is done with `GSI_DAEMON_NAME`. This name is specified with the following format

```
GSI_DAEMON_NAME = /X.509/name/of/server/1,/X.509/name/of/server/2,...
```

A complete example that has the question marks filled in and the daemon's user name filled in is given in the example configuration below.

Condor will also need a way to map an X.509 distinguished name to a Condor user id. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.6.4 to use Condor's unified map file. The second way to do the mapping is within an administrator-maintained GSI-specific file called an X.509 map file, mapping from X509 Distinguished Name (DN) to Condor user id. It is similar to a Globus grid map file, except that it is only used for mapping to a user id, not for authorization. If the user names in the map file do not specify a domain for the user (specification would appear as `user@domain`), then the value of `UID_DOMAIN` is used. Information about authorization can be found in Section 3.6.7. Entries (lines) in the file each contain two items. The first item in an entry is the X.509 certificate subject name, and it is enclosed in quotes (using the character `"`). The second item is the Condor user id. The two items in an entry are separated by tab or space character(s). Here is an example of an entry in an X.509 map file. Entries must be on a single line; this example is broken onto two lines for formatting reasons.

```
"/C=US/O=Globus/O=University of Wisconsin/
OU=Computer Sciences Department/CN=Alice Smith" asmith
```

Condor finds the map file in one of three ways. If the configuration variable `GRIDMAP` is defined, it gives the full path name to the map file. When not defined, Condor looks for the map file in

```
$(GSI_DAEMON_DIRECTORY)/grid-mapfile
```

If `GSI_DAEMON_DIRECTORY` is not defined, then the third place Condor looks for the map file is given by

```
/etc/grid-security/grid-mapfile
```

GSI certificate locations for Users The user specifies the location of a certificate, proxy, etc. in one of two ways:

1. Environment variables give the location of necessary items.
`X509_USER_PROXY` gives the path and file name of the proxy. This proxy will have been created using the *grid-proxy-init* program, which will place the proxy in the `/tmp` directory with the file name being determined by the format:

```
/tmp/x509up_uXXXX
```

The specific file name is given by substituting the `XXXX` characters with the UID of the user. Note that when a valid proxy is used, the certificate and key locations are not needed.

`X509_USER_CERT` gives the path and file name of the certificate. It is also used if a proxy location has been checked, but the proxy is no longer valid.

`X509_USER_KEY` gives the path and file name of the key. Note that most keys are password encrypted, such that knowing the location could not lead to using the key.

`X509_CERT_DIR` gives the path to the directory containing the list of trusted CAs.

2. Without environment variables to give locations of necessary certificate information, Condor uses a default directory for the user. This directory is given by

```
$(HOME)/.globus
```

Example GSI Security Configuration Here is an example portion of the configuration file that would enable and require GSI authentication, along with a minimal set of other variables to make it work.

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
SEC_DEFAULT_AUTHENTICATION_METHODS = GSI
SEC_DEFAULT_INTEGRITY = REQUIRED
GSI_DAEMON_DIRECTORY = /etc/grid-security
GRIDMAP = /etc/grid-security/grid-mapfile

# authorize based on user names produced by the map file
ALLOW_READ = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_DAEMON = condor@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR = condor@cs.wisc.edu/condor.cs.wisc.edu, \
                    condor@cs.wisc.edu/condor2.cs.wisc.edu
```

```

ALLOW_ADMINISTRATOR = condor-admin@cs.wisc.edu/*.cs.wisc.edu

# condor daemon certificate(s) trusted by condor tools and daemons
# when connecting to other condor daemons
GSI_DAEMON_NAME = /C=US/O=Condor/O=UW/OU=CS/CN=condor@cs.wisc.edu

# clear out any host-based authorizations
# (unnecessary if you leave authentication REQUIRED,
# but useful if you make it optional and want to
# allow some unauthenticated operations, such as
# ALLOW_READ = /*.cs.wisc.edu)
HOSTALLOW_READ =
HOSTALLOW_WRITE =
HOSTALLOW_NEGOTIATOR =
HOSTALLOW_ADMINISTRATOR =

```

The `SEC_DEFAULT_AUTHENTICATION` macro specifies that authentication is required for all communications. This single macro covers all communications, but could be replaced with a set of macros that require authentication for only specific communications.

The macro `GSI_DAEMON_DIRECTORY` is specified to give Condor a single place to find the daemon's certificate. This path may be a directory on a shared file system such as AFS. Alternatively, this path name can point to local copies of the certificate stored in a local file system.

The macro `GRIDMAP` specifies the file to use for mapping GSI names to user names within Condor. For example, it might look like this:

```
" /C=US/O=Condor/O=UW/OU=CS/CN=condor@cs.wisc.edu" condor@cs.wisc.edu
```

Additional mappings would be needed for the users who submit jobs to the pool or who issue administrative commands.

SSL Authentication

SSL authentication is similar to GSI authentication, but without GSI's delegation (proxy) capabilities. SSL utilizes X.509 certificates.

All SSL authentication is mutual authentication in Condor. This means that when SSL authentication is used and when one process communicates with another, each process must be able to verify the signature on the certificate presented by the other process. The process that initiates the connection is the client, and the process that receives the connection is the server. For example, when a *condor_startd* daemon authenticates with a *condor_collector* daemon to provide a machine ClassAd, the *condor_startd* daemon initiates the connection and acts as the client, and the *condor_collector* daemon acts as the server.

The names and locations of keys and certificates for clients, servers, and the files used to specify trusted certificate authorities (CAs) are defined by settings in the configuration files. The contents of the files are identical in format and interpretation to those used by other systems which use SSL, such as Apache httpd.

The configuration variables `AUTH_SSL_CLIENT_CERTFILE` and `AUTH_SSL_SERVER_CERTFILE` specify the file location for the certificate file for the initiator and recipient of connections, respectively. Similarly, the configuration variables `AUTH_SSL_CLIENT_KEYFILE` and `AUTH_SSL_SERVER_KEYFILE` specify the locations for keys.

The configuration variables `AUTH_SSL_SERVER_CAFILE` and `AUTH_SSL_CLIENT_CAFILE` each specify a path and file name, providing the location of a file containing one or more certificates issued by trusted certificate authorities. Similarly, `AUTH_SSL_SERVER_CADIR` and `AUTH_SSL_CLIENT_CADIR` each specify a directory with one or more files, each which may contain a single CA certificate. The directories must be prepared using the OpenSSL `c_rehash` utility.

Kerberos Authentication

If Kerberos is used for authentication, then a mapping from a Kerberos domain (called a realm) to a Condor UID domain is necessary. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.6.4 to use Condor's unified map file. A second way to specify the mapping defines the configuration variable `KERBEROS_MAP_FILE` to define a path to an administrator-maintained Kerberos-specific map file. The configuration syntax is

```
KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

Lines within this map file have the syntax

```
KERB.REALM = UID.domain.name
```

Here are two lines from a map file to use as an example:

```
CS.WISC.EDU    = cs.wisc.edu
ENGR.WISC.EDU  = ee.wisc.edu
```

If a `KERBEROS_MAP_FILE` configuration variable is defined and set, then all permitted realms must be explicitly mapped. If no map file is specified, then Condor assumes that the Kerberos realm is the same as the Condor UID domain.

The configuration variable `KERBEROS_SERVER_PRINCIPAL` defines the name of a Kerberos principal. If `KERBEROS_SERVER_PRINCIPAL` is not defined, then the default value used is `host`. A principal specifies a unique name to which a set of credentials may be assigned.

Condor takes the specified (or default) principal and appends a slash character, the host name, an '@' (at sign character), and the Kerberos realm. As an example, the configuration

```
KERBEROS_SERVER_PRINCIPAL = condor-daemon
```

results in Condor's use of

```
condor-daemon/the.host.name@YOUR.KERB.REALM
```

as the server principal.

Here is an example of configuration settings that use Kerberos for authentication and require authentication of all communications of the write or administrator access level.

```
SEC_WRITE_AUTHENTICATION          = REQUIRED
SEC_WRITE_AUTHENTICATION_METHODS  = KERBEROS
SEC_ADMINISTRATOR_AUTHENTICATION  = REQUIRED
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS = KERBEROS
```

Kerberos authentication on Unix platforms requires access to various files that usually are only accessible by the root user. At this time, the only supported way to use KERBEROS authentication on Unix platforms is to start daemons Condor as user `root`.

Password Authentication

The password method provides mutual authentication through the use of a shared secret. This is often a good choice when strong security is desired, but an existing Kerberos or X.509 infrastructure is not in place. Password authentication is available on both Unix and Windows. It currently can only be used for daemon-to-daemon authentication. The shared secret in this context is referred to as the *pool password*.

Before a daemon can use password authentication, the pool password must be stored on the daemon's local machine. On Unix, the password will be placed in a file defined by the configuration variable `SEC_PASSWORD_FILE`. This file will be accessible only by the UID that Condor is started as. On Windows, the same secure password store that is used for user passwords will be used for the pool password (see section 6.2.3).

Under Unix, the password file can be generated by using the following command to write directly to the password file:

```
condor_store_cred -f /path/to/password/file
```

Under Windows (or under Unix), storing the pool password is done with the `-c` option when using to `condor_store_cred add`. Running

```
condor_store_cred -c add
```

prompts for the pool password and store it on the local machine, making it available for daemons to use in authentication. The *condor_master* must be running for this command to work.

In addition, storing the pool password to a given machine requires CONFIG-level access. For example, if the pool password should only be set locally, and only by root, the following would be placed in the global configuration file.

```
ALLOW_CONFIG = root@mydomain/${IP_ADDRESS}
```

It is also possible to set the pool password remotely, but this is recommended only if it can be done over an encrypted channel. This is possible on Windows, for example, in an environment where common accounts exist across all the machines in the pool. In this case, `ALLOW_CONFIG` can be set to allow the Condor administrator (who in this example has an account `condor` common to all machines in the pool) to set the password from the central manager as follows.

```
ALLOW_CONFIG = condor@mydomain/${CONDOR_HOST}
```

The Condor administrator then executes

```
condor_store_cred -c -n host.mydomain add
```

from the central manager to store the password to a given machine. Since the `condor` account exists on both the central manager and `host.mydomain`, the NTSSPI authentication method can be used to authenticate and encrypt the connection. `condor_store_cred` will warn and prompt for cancellation, if the channel is not encrypted for whatever reason (typically because common accounts do not exist or Condor's security is misconfigured).

When a daemon is authenticated using a pool password, its security principle is `condor_pool@$(UID_DOMAIN)`, where `$(UID_DOMAIN)` is taken from the daemon's configuration. The `ALLOW_DAEMON` and `ALLOW_NEGOTIATOR` configuration variables for authorization should restrict access using this name. For example,

```
ALLOW_DAEMON = condor_pool@mydomain/*, condor@mydomain/${IP_ADDRESS}
ALLOW_NEGOTIATOR = condor_pool@mydomain/${CONDOR_HOST}
```

This configuration allows remote DAEMON-level and NEGOTIATOR-level access, if the pool password is known. Local daemons authenticated as `condor@mydomain` are also allowed access. This is done so local authentication can be done using another method such as FS.

Example Security Configuration Using Pool Password The following example configuration uses pool password authentication and network message integrity checking for all communication between Condor daemons.

```
SEC_PASSWORD_FILE = $(LOCK)/pool_password
SEC_DAEMON_AUTHENTICATION = REQUIRED
SEC_DAEMON_INTEGRITY = REQUIRED
```

```

SEC_DAEMON_AUTHENTICATION_METHODS = PASSWORD
SEC_NEGOTIATOR_AUTHENTICATION = REQUIRED
SEC_NEGOTIATOR_INTEGRITY = REQUIRED
SEC_NEGOTIATOR_AUTHENTICATION_METHODS = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_DAEMON = condor_pool@$(UID_DOMAIN)/*.cs.wisc.edu, \
               condor@$(UID_DOMAIN)/$(IP_ADDRESS)
ALLOW_NEGOTIATOR = condor_pool@$(UID_DOMAIN)/negotiator.machine.name

```

Example Using Pool Password for *condor_startd* Advertisement One problem with the pool password method of authentication is that it involves a single, shared secret. This does not scale well with the addition of remote users who flock to the local pool. However, the pool password may still be used for authenticating portions of the local pool, while others (such as the remote *condor_schedd* daemons involved in flocking) are authenticated by other means.

In this example, only the *condor_startd* daemons in the local pool are required to have the pool password when they advertise themselves to the *condor_collector* daemon.

```

SEC_PASSWORD_FILE = $(LOCK)/pool_password
SEC_ADVERTISE_STARTD_AUTHENTICATION = REQUIRED
SEC_ADVERTISE_STARTD_INTEGRITY = REQUIRED
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS = PASSWORD
SEC_CLIENT_AUTHENTICATION_METHODS = FS, PASSWORD, KERBEROS, GSI
ALLOW_ADVERTISE_STARTD = condor_pool@$(UID_DOMAIN)/*.cs.wisc.edu

```

File System Authentication

This form of authentication utilizes the ownership of a file in the identity verification of a client. A daemon authenticating a client requires the client to write a file in a specific location (*/tmp*). The daemon then checks the ownership of the file. The file's ownership verifies the identity of the client. In this way, the file system becomes the trusted authority. This authentication method is only appropriate for clients and daemons that are on the same computer.

File System Remote Authentication

Like file system authentication, this form of authentication utilizes the ownership of a file in the identity verification of a client. In this case, a daemon authenticating a client requires the client to write a file in a specific location, but the location is not restricted to */tmp*. The location of the file is specified by the configuration variable *FS_REMOTE_DIR*.

Windows Authentication

This authentication is done only among Windows machines using a proprietary method. The Windows security interface SSPI is used to enforce NTLM (NT LAN Manager). The authentication is

based on challenge and response, using the user's password as a key. This is similar to Kerberos. The main difference is that Kerberos provides an access token that typically grants access to an entire network, whereas NTLM authentication only verifies an identity to one machine at a time. NTSSPI is best-used in a way similar to file system authentication in Unix, and probably should not be used for authentication between two computers.

Claim To Be Authentication

Claim To Be authentication accepts any identity claimed by the client. As such, it does not authenticate. It is included in Condor and in the list of authentication methods for testing purposes only.

Anonymous Authentication

Anonymous authentication causes authentication to be skipped entirely. As such, it does not authenticate. It is included in Condor and in the list of authentication methods for testing purposes only.

3.6.4 The Unified Map File for Authentication

Condor's unified map file allows the mappings from authenticated names to a Condor canonical user name to be specified as a single list within a single file. The location of the unified map file is defined by the configuration variable `CERTIFICATE_MAPFILE` ; it specifies the path and file name of the unified map file. Each mapping is on its own line of the unified map file. Each line contains 3 fields, separated by white space (space or tab characters):

1. The name of the authentication method to which the mapping applies.
2. A regular expression representing the authenticated name to be mapped.
3. The canonical Condor user name.

Allowable authentication method names are the same as used to define any of the configuration variables `SEC_*_AUTHENTICATION_METHODS`, as repeated here:

```
GSI
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
NTSSPI
```

```
CLAIMTOBE
ANONYMOUS
```

The fields that represent an authenticated name and the canonical Condor user name may utilize regular expressions as defined by PCRE (Perl-Compatible Regular Expressions). Due to this, more than one line (mapping) within the unified map file may match. Look ups are therefore defined to use the first mapping that matches.

A regular expression may need to contain spaces, and in this case the entire expression can be surrounded by double-quotes. If a double-quote character also needs to appear in such an expression, it should be preceded by a backslash.

The default behavior of Condor when no mapfile is specified is to do the following mappings, with some additional logic noted below:

```
FS (.* ) \1
FS_REMOTE (.* ) \1
GSI (.* ) GSS_ASSIST_GRIDMAP
SSL (.* ) ssl@unmappeduser
KERBEROS ([^/]* )/?[^@]*@(.* ) \1@\2
NTSSPI (.* ) \1
CLAIMTOBE (.* ) \1
PASSWORD (.* ) \1
```

For GSI (or SSL), the special name `GSS_ASSIST_GRIDMAP` instructs Condor to use the GSI grid mapfile (configured with `GRIDMAP` as shown in section 3.6.3) to do the mapping. If no mapping can be found for GSI (with or without the use of `GSS_ASSIST_GRIDMAP`), the user is mapped to `gsi@unmappeduser`.

For Kerberos, if `KERBEROS_MAP_FILE` is specified, the domain portion of the name is obtained by mapping the Kerberos realm to the value specified in the map file, rather than just using the realm verbatim as the domain portion of the condor user name. See section 3.6.3 for details.

3.6.5 Encryption

Encryption provides privacy support between two communicating parties. Through configuration macros, both the client and the daemon can specify whether encryption is required for further communication.

The client uses one of two macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_CLIENT_ENCRYPTION
```

For the daemon, there are seven macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_READ_ENCRYPTION
SEC_WRITE_ENCRYPTION
SEC_ADMINISTRATOR_ENCRYPTION
SEC_CONFIG_ENCRYPTION
SEC_OWNER_ENCRYPTION
SEC_DAEMON_ENCRYPTION
SEC_NEGOTIATOR_ENCRYPTION
SEC_ADVERTISE_MASTER_ENCRYPTION
SEC_ADVERTISE_STARTD_ENCRYPTION
SEC_ADVERTISE_SCHDED_ENCRYPTION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_ENCRYPTION = REQUIRED
```

signifies that any communication that changes a daemon's configuration must be encrypted. If a daemon's configuration contains

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

and does not contain any other security configuration for ENCRYPTION, then this default defines the daemon's needs for encryption over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If encryption is to be done, then the communicating parties must find (negotiate) a mutually acceptable method of encryption to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_CRYPTOMETHODS
SEC_CLIENT_CRYPTOMETHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_CRYPTOMETHODS
SEC_READ_CRYPTOMETHODS
SEC_WRITE_CRYPTOMETHODS
SEC_ADMINISTRATOR_CRYPTOMETHODS
SEC_CONFIG_CRYPTOMETHODS
SEC_OWNER_CRYPTOMETHODS
SEC_DAEMON_CRYPTOMETHODS
SEC_NEGOTIATOR_CRYPTOMETHODS
SEC_ADVERTISE_MASTER_CRYPTOMETHODS
SEC_ADVERTISE_STARTD_CRYPTOMETHODS
SEC_ADVERTISE_SCHDED_CRYPTOMETHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the encryption methods that are available to be used. The ordering of the list gives preference; the first item in the list indicates the highest preference. Possible values are

```
3DES
BLOWFISH
```

3.6.6 Integrity

An integrity check assures that the messages between communicating parties have not been tampered with. Any change, such as addition, modification, or deletion can be detected. Through configuration macros, both the client and the daemon can specify whether an integrity check is required of further communication.

The client uses one of two macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_CLIENT_INTEGRITY
```

For the daemon, there are seven macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_READ_INTEGRITY
SEC_WRITE_INTEGRITY
SEC_ADMINISTRATOR_INTEGRITY
SEC_CONFIG_INTEGRITY
SEC_OWNER_INTEGRITY
SEC_DAEMON_INTEGRITY
SEC_NEGOTIATOR_INTEGRITY
SEC_ADVERTISE_MASTER_INTEGRITY
SEC_ADVERTISE_STARTD_INTEGRITY
SEC_ADVERTISE_SCHEDD_INTEGRITY
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_INTEGRITY = REQUIRED
```

signifies that any communication that changes a daemon's configuration must have its integrity assured. If a daemon's configuration contains

```
SEC_DEFAULT_INTEGRITY = REQUIRED
```

and does not contain any other security configuration for `INTEGRITY`, then this default defines the daemon's needs for integrity checks over all access levels. Where a specific macro is present, its value takes precedence over any default given.

A signed MD5 check sum is currently the only available method for integrity checking. Its use is implied whenever integrity checks occur. If more methods are implemented, then there will be further macros to allow both the client and the daemon to specify which methods are acceptable.

3.6.7 Authorization

Authorization protects resource usage by granting or denying access requests made to the resources. It defines who is allowed to do what.

Authorization is defined in terms of users. An initial implementation provided authorization based on hosts (machines), while the current implementation relies on user-based authorization. Section 3.6.9 on Setting Up IP/Host-Based Security in Condor describes the previous implementation. This IP/Host-Based security still exists, and it can be used, but significantly stronger and more flexible security can be achieved with the newer authorization based on fully qualified user names. This section discusses user-based authorization.

The authorization portion of the security of a Condor pool is based on a set of configuration macros. The macros list which user will be authorized to issue what request given a specific access level. When a daemon is to be authorized, its user name is the login under which the daemon is executed.

These configuration macros define a set of users that will be allowed to (or denied from) carrying out various Condor commands. Each access level may have its own list of authorized users. A complete list of the authorization macros:

```
ALLOW_READ
ALLOW_WRITE
ALLOW_ADMINISTRATOR
ALLOW_CONFIG
ALLOW_SOAP
ALLOW_OWNER
ALLOW_NEGOTIATOR
ALLOW_DAEMON
DENY_READ
DENY_WRITE
DENY_ADMINISTRATOR
DENY_SOAP
DENY_CONFIG
DENY_OWNER
DENY_NEGOTIATOR
DENY_DAEMON
```

In addition, the following are used to control authorization of specific types of Condor daemons when advertising themselves to the pool. If unspecified, these default to the broader ALLOW_DAEMON and DENY_DAEMON settings.

```
ALLOW_ADVERTISE_MASTER
ALLOW_ADVERTISE_STARTD
ALLOW_ADVERTISE_SCHEDD
DENY_ADVERTISE_MASTER
DENY_ADVERTISE_STARTD
DENY_ADVERTISE_SCHEDD
```

Each client side of a connection may also specify its own list of trusted servers. This is done using the following settings. Note that the FS and CLAIMTOBE authentication methods are not symmetric. The client is authenticated by the server, but the server is not authenticated by the client. When the server is not authenticated to the client, only the network address of the host may be authorized and not the specific identity of the server.

```
ALLOW_CLIENT
DENY_CLIENT
```

All authorization settings are defined by a comma-separated list of fully qualified users. Each fully qualified user is described using the following format:

```
username@domain/hostname
```

The information to the left of the slash character describes a user within a domain. The information to the right of the slash character describes one or more machines from which the user would be issuing a command. This host name may take the form of either a fully qualified host name of the form

```
bird.cs.wisc.edu
```

or an IP address of the form

```
128.105.128.0
```

An example is

```
zmiller@cs.wisc.edu/bird.cs.wisc.edu
```

Within the format, wild card characters (the asterisk, *) are allowed. The use of wild cards is limited to one wild card on either side of the slash character. A wild card character used in the host name is further limited to come at the beginning of a fully qualified host name or at the end of an IP address. For example,

```
*@cs.wisc.edu/bird.cs.wisc.edu
```

refers to any user that comes from `cs.wisc.edu`, where the command is originating from the machine `bird.cs.wisc.edu`. Another valid example,

```
zmiller@cs.wisc.edu/*.cs.wisc.edu
```

refers to commands coming from any machine within the `cs.wisc.edu` domain, and issued by `zmiller`. A third valid example,

```
*@cs.wisc.edu/*
```

refers to commands coming from any user within the `cs.wisc.edu` domain where the command is issued from any machine. A fourth valid example,

```
*@cs.wisc.edu/128.105.*
```

refers to commands coming from any user within the `cs.wisc.edu` domain where the command is issued from machines within the network that match the first two octets of the IP address.

If the set of machines is specified by an IP address, then further specification using a net mask identifies a physical set (subnet) of machines. This physical set of machines is specified using the form

```
network/netmask
```

The `network` is an IP address. The net mask takes one of two forms. It may be a decimal number which refers to the number of leading bits of the IP address that are used in describing a subnet. Or, the net mask may take the form of

```
a.b.c.d
```

where `a`, `b`, `c`, and `d` are decimal numbers that each specify an 8-bit mask. An example net mask is

```
255.255.192.0
```

which specifies the bit mask

```
11111111.11111111.11000000.00000000
```

A single complete example of a configuration variable that uses a net mask is

```
ALLOW_WRITE = joesmith@cs.wisc.edu/128.105.128.0/17
```

User `joesmith` within the `cs.wisc.edu` domain is given write authorization when originating from machines that match their leftmost 17 bits of the IP address.

This flexible set of configuration macros could be used to define conflicting authorization. Therefore, the following protocol defines the precedence of the configuration macros.

1. `DENY_*` macros take precedence over `ALLOW_*` macros where there is a conflict. This implies that if a specific user is both denied and granted authorization, the conflict is resolved by denying access.
2. If macros are omitted, the default behavior is to grant authorization for every user.

Example of Authorization Security Configuration

An example of the configuration variables for the user-side authorization is derived from the necessary access levels as described in Section 3.6.1.

```
ALLOW_READ          = *@cs.wisc.edu/*
ALLOW_WRITE         = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_ADMINISTRATOR = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_CONFIG        = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR    = condor@cs.wisc.edu/condor.cs.wisc.edu, \
                      condor@cs.wisc.edu/condor2.cs.wisc.edu
ALLOW_DAEMON        = condor@cs.wisc.edu/*.cs.wisc.edu

# Clear out any old-style HOSTALLOW settings:
HOSTALLOW_READ =
HOSTALLOW_WRITE =
HOSTALLOW_DAEMON =
HOSTALLOW_NEGOTIATOR =
HOSTALLOW_ADMINISTRATOR =
HOSTALLOW_OWNER =
```

This example configuration authorizes any authenticated user in the `cs.wisc.edu` domain to carry out a request that requires the `READ` access level from any machine. Any user in the `cs.wisc.edu` domain may carry out a request that requires the `WRITE` access level from any machine in the `cs.wisc.edu` domain. Only the user called `condor-admin` may carry out a request that requires the `ADMINISTRATOR` access level from any machine in the `cs.wisc.edu` domain. The administrator, logged into any machine within the `cs.wisc.edu` domain is authorized at the `CONFIG` access level. Only the negotiator daemon, running as `condor` on the two central managers are authorized with the `NEGOTIATOR` access level. And, the last line of the example presumes that there is a user called `condor`, and that the daemons have all been started up as this user. It authorizes only programs (which will be the daemons) running as `condor` to carry out requests that require the `DAEMON` access level, where the commands originate from any machine in the `cs.wisc.edu` domain.

In the local configuration file for each host, the host's owner should be authorized as the owner of the machine. An example of the entry in the local configuration file:

```
ALLOW_OWNER          = username@cs.wisc.edu/hostname.cs.wisc.edu
```

In this example the owner has a login of `username`, and the machine's name is represented by `hostname`.

Debugging Security Configuration

If the authorization policy denies a network request, an explanation of why the request was denied is printed in the log file of the daemon that denied the request. The line in the log file contains the words `PERMISSION DENIED`.

To get Condor to generate a similar explanation of why requests are accepted, add `D_SECURITY` to the daemon's debug options (and restart or reconfig the daemon). The line in the log file for these cases will contain the words `PERMISSION GRANTED`. If you do not want to see a full explanation but just want to see when requests are made, add `D_COMMAND` to the daemon's debug options.

If the authorization policy makes use of host or domain names, then be aware that Condor depends on DNS to map IP addresses to names. The security and accuracy of your DNS service is therefore a requirement. Typos in DNS mappings are an occasional source of unexpected behavior. If the authorization policy is not behaving as expected, carefully compare the names in the policy with the host names Condor mentions in the explanations of why requests are granted or denied.

3.6.8 Security Sessions

To set up and configure secure communications in Condor, authentication, encryption, and integrity checks can be used. However, these come at a cost: performing strong authentication can take a significant amount of time, and generating the cryptographic keys for encryption and integrity checks can take a significant amount of processing power.

The Condor system makes many network connections between different daemons. If each one of these was to be authenticated, and new keys were generated for each connection, Condor would not be able to scale well. Therefore, Condor uses the concept of *sessions* to cache relevant security information for future use and greatly speed up the establishment of secure communications between the various Condor daemons.

A new session is established the first time a connection is made from one daemon to another. Each session has a fixed lifetime after which it will expire and a new session will need to be created again. But while a valid session exists, it can be re-used as many times as needed, thereby preventing the need to continuously re-establish secure connections. Each entity of a connection will have access to a *session key* that proves the identity of the other entity on the opposing side of the connection. This session key is exchanged securely using a strong authentication method, such as Kerberos or GSI. Other authentication methods, such as NTSSPI, `FS_REMOTE`, `CLAIMTOBE`, and `ANONYMOUS`, do not support secure key exchange. An entity listening on the wire may be able to impersonate the client or server in a session that does not use a strong authentication method.

Establishing a secure session requires that either the encryption or the integrity options be enabled. If the encryption capability is enabled, then the session will be restarted using the session key as the encryption key. If integrity capability is enabled, then the check sum includes the session key even though it is not transmitted. Without either of these two methods enabled, it is possible for an attacker to use an open session to make a connection to a daemon and use that connection for nefarious purposes. It is strongly recommended that if *you have authentication turned on, you should also turn on integrity and/or encryption.*

The configuration parameter `SEC_DEFAULT_NEGOTIATION` will allow a user to set the default level of secure sessions in Condor. Like other security settings, the possible values for this parameter can be `REQUIRED`, `PREFERRED`, `OPTIONAL`, or `NEVER`. If you disable sessions and you have authentication turned on, then most authentication (other than commands like `condor_submit`) will fail because Condor requires sessions when you have security turned on. On the other hand, if you are not using strong security in Condor, but you are relying on the default host-based security, turning off sessions may be useful in certain situations. These might include debugging problems with the security session management or slightly decreasing the memory consumption of the daemons, which keep track of the sessions in use.

Session lifetimes for specific daemons are already properly configured in the default installation of Condor. Condor tools such as `condor_q` and `condor_status` create a session that expires after one minute. Theoretically they should not create a session at all, because the session cannot be reused between program invocations, but this is difficult to do in the general case. This allows a very small window of time for any possible attack, and it helps keep the memory footprint of running daemons down, because they are not keeping track of all of the sessions. The session durations may be manually tuned by using macros in the configuration file, but this is not recommended.

3.6.9 Host-Based Security in Condor

This section describes the mechanisms for setting up Condor's host-based security. This is now an outdated form of implementing security levels for machine access. It remains available and documented for purposes of backward compatibility. If used at the same time as the user-based authorization, the two specifications are merged together.

The host-based security paradigm allows control over which machines can join a Condor pool, which machines can find out information about your pool, and which machines within a pool can perform administrative commands. By default, Condor is configured to allow anyone to view or join a pool. It is recommended that this parameter is changed to only allow access from machines that you trust.

This section discusses how the host-based security works inside Condor. It lists the different levels of access and what parts of Condor use which levels. There is a description of how to configure a pool to grant or deny certain levels of access to various machines. Configuration examples and the settings of configuration variables using the `condor_config_val` command complete this section.

Inside the Condor daemons or tools that use DaemonCore (see section 3.9 for details), most tasks are accomplished by sending commands to another Condor daemon. These commands are

represented by an integer value to specify which command is being requested, followed by any optional information that the protocol requires at that point (such as a ClassAd, capability string, etc). When the daemons start up, they will register which commands they are willing to accept, what to do with arriving commands, and the access level required for each command. When a command request is received by a daemon, Condor identifies the access level required and checks the IP address of the sender to verify that it satisfies the allow/deny settings from the configuration file. If permission is granted, the command request is honored; otherwise, the request will be aborted.

Settings for the access levels in the global configuration file will affect all the machines in the pool. Settings in a local configuration file will only affect the specific machine. The settings for a given machine determine what other hosts can send commands to that machine. If a machine foo is to be given administrator access on machine bar, place foo in bar's configuration file access list (not the other way around).

The following are the various access levels that commands within Condor can be registered with:

READ Machines with READ access can read information from the Condor daemons. For example, they can view the status of the pool, see the job queue(s), and view user permissions. READ access does not allow a machine to alter any information, and does not allow job submission. A machine listed with READ permission will be unable join a Condor pool; the machine can only view information about the pool.

WRITE Machines with WRITE access can write information to the Condor daemons. Most important for granting a machine with this access is that the machine will be able to join a pool since they are allowed to send ClassAd updates to the central manager. The machine can talk to the other machines in a pool in order to submit or run jobs. In addition, any machine with WRITE access can request the *condor_startd* daemon to perform periodic checkpoints on an executing job. After the checkpoint is completed, the job will continue to execute and the machine will still be claimed by the original *condor_schedd* daemon. This allows users on the machines where they submitted their jobs to use the *condor_checkpoint* command to get their jobs to periodically checkpoint, even if the users do not have an account on the machine where the jobs execute.

IMPORTANT: For a machine to join a Condor pool, the machine must have both WRITE permission **AND** READ permission. WRITE permission is not enough.

ADMINISTRATOR Machines with ADMINISTRATOR access are granted additional Condor administrator rights to the pool. This includes the ability to change user priorities (with the command *userprio -set*), and the ability to turn Condor on and off (with the command *condor_off <machine>*). It is recommended that few machines be granted administrator access in a pool; typically these are the machines that are used by Condor and system administrators as their primary workstations, or the machines running as the pool's central manager.

IMPORTANT: Giving ADMINISTRATOR privileges to a machine grants administrator access for the pool to **ANY USER** on that machine. This includes any users who can run Condor jobs on that machine. It is recommended that ADMINISTRATOR access is granted with due diligence.

OWNER This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. For example, the *condor_vacate* command causes the *condor_startd* daemon to vacate any running Condor job. It requires OWNER permission, so that any user logged into a local machine can issue a *condor_vacate* command.

NEGOTIATOR This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run.

CONFIG This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, machines with this level of access are able to change any configuration parameter, except those specified in the *condor_config.root* configuration file. Therefore, one should exercise extreme caution before granting this level of host-wide access. Because of the implications caused by CONFIG privileges, it is disabled by default for all hosts.

DAEMON This access level is used for commands that are internal to the operation of Condor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon (which may be more specifically controlled by the ADVERTISE_STARTD access level). Authorization at this access level should only be given to hosts that actually run Condor in your pool. The DAEMON level of access implies both READ and WRITE access. Any setting for this access level that is not defined will default to the corresponding setting in the WRITE access level.

ADVERTISEMASTER This access level is used specifically for commands used to advertise a *condor_master* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

ADVERTISESTARTD This access level is used specifically for commands used to advertise a *condor_startd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

ADVERTISESCHEDD This access level is used specifically for commands used to advertise a *condor_schedd* daemon to the collector. Any setting for this access level that is not defined will default to the corresponding setting in the DAEMON access level.

CLIENT This access level is different from all the others. Whereas all of the other access levels refer to the security policy for accepting connections *from* others, the CLIENT access level applies when a Condor daemon or tool is connecting *to* some other Condor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

Condor provides a mechanism for more fine-grained control over the configuration settings that can be modified remotely with *condor_config_val*. Host-based security access permissions are specified in configuration files.

ADMINISTRATOR and NEGOTIATOR access default to the central manager machine. OWNER access defaults to the local machine, as well as any machines given with ADMINISTRATOR access. CONFIG access is not granted to any machine as its default. These defaults are sufficient for most pools, and should not be changed without a compelling reason. If machines other than the default are to have to have OWNER access, they probably should also have ADMINISTRATOR access. By granting machines ADMINISTRATOR access, they will automatically have OWNER access, given how OWNER access is set within the configuration.

The default access configuration is

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
HOSTALLOW_WRITE_COLLECTOR = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_WRITE_STARTD = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_READ_COLLECTOR = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_READ_STARTD = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_CLIENT = *
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine.

For each access level, an ALLOW or a DENY may be added.

- If you have an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.
- If you have a DENY, it means "deny these machines". No DENY means to deny nobody.
- If you have both an ALLOW and a DENY, it means allow the machines listed in ALLOW except for the machines listed in DENY.
- Exclusively for the CONFIG access, no ALLOW means allow no one. Note that this is different than the other ALLOW configurations. It is different to enable more stringent security where older configurations are used, since older configuration files would not have a CONFIG configuration entry.

Multiple machine entries in the configuration files may be separated by either a space or a comma. The machines may be listed by

- Individual host names - for example: condor.cs.wisc.edu
- Individual IP address - for example: 128.105.67.29
- IP subnets (use a trailing "*") - for example: 144.105.*, 128.105.67.*

- Host names with a wild card “*” character (only one “*” is allowed per name) - for example:
.cs.wisc.edu, sol.cs.wisc.edu

To resolve an entry that falls into both allow and deny: individual machines have a higher order of precedence than wild card entries, and host names with a wild card have a higher order of precedence than IP subnets. Otherwise, DENY has a higher order of precedence than ALLOW. (this is how most people would intuitively expect it to work).

In addition, the above access levels may be specified on a per-daemon basis, instead of machine-wide for all daemons. Do this with the subsystem string (described in section 3.3.1 on Subsystem Names), which is one of: STARTD, SCHEDD, MASTER, NEGOTIATOR, or COLLECTOR. For example, to grant different read access for the *condor_schedd*:

```
HOSTALLOW_READ_SCHEDD = <list of machines>
```

The following is a list of registered commands that daemons will accept. The list is ordered by daemon. For each daemon, the commands are grouped by the access level required for a daemon to accept the command from a given machine.

ALL DAEMONS:

WRITE The command sent as a result of *condor_reconfig* to reconfigure a daemon.

ADMINISTRATOR The command sent as a result of *reconfig -full* to perform a full reconfiguration on a daemon.

STARTD:

WRITE All commands that relate to a *condor_schedd* daemon claiming a machine, starting jobs there, or stopping those jobs.

The command that *condor_checkpoint* sends to periodically checkpoint all running jobs.

READ The command that *condor_preen* sends to request the current state of the *condor_startd* daemon.

OWNER The command that *condor_vacate* sends to cause any running jobs to stop running.

NEGOTIATOR The command that the *condor_negotiator* daemon sends to match a machine's *condor_startd* daemon with a given *condor_schedd* daemon.

NEGOTIATOR:

WRITE The command that initiates a new negotiation cycle. It is sent by the *condor_schedd* when new jobs are submitted or a *condor_reschedule* command is issued.

READ The command that can retrieve the current state of user priorities in the pool (sent by the *condor_userprio* command).

ADMINISTRATOR The command that can set the current values of user priorities (sent as a result of the `userprio -set` command).

COLLECTOR:

ADVERTISEMASTER Commands that update the *condor_collector* daemon with new *condor_master* ClassAds.

ADVERTISESCHEDD Commands that update the *condor_collector* daemon with new *condor_schedd* ClassAds.

ADVERTISESTARTD Commands that update the *condor_collector* daemon with new *condor_startd* ClassAds.

DAEMON All other commands that update the *condor_collector* daemon with new ClassAds. Note that the specific access levels such as `ADVERTISE_STARTD` default to the `DAEMON` settings, which in turn defaults to `WRITE`.

READ All commands that query the *condor_collector* daemon for ClassAds.

SCHEDD:

NEGOTIATOR The command that the *condor_negotiator* sends to begin negotiating with this *condor_schedd* to match its jobs with available *condor_startds*.

WRITE The command which *condor_reschedule* sends to the *condor_schedd* to get it to update the *condor_collector* with a current ClassAd and begin a negotiation cycle.

The commands that a *condor_startd* sends to the *condor_schedd* when it must vacate its jobs and release the *condor_schedd*'s claim.

The commands which write information into the job queue (such as *condor_submit* and *condor_hold*). Note that for most commands which attempt to write to the job queue, Condor will perform an additional user-level authentication step. This additional user-level authentication prevents, for example, an ordinary user from removing a different user's jobs.

READ The command from any tool to view the status of the job queue.

MASTER: All commands are registered with `ADMINISTRATOR` access:

restart : Master restarts itself (and all its children)

off : Master shuts down all its children

off -master : Master shuts down all its children and exits

on : Master spawns all the daemons it is configured to spawn

This section provides examples of configuration settings. Notice that ADMINISTRATOR access is only granted through a HOSTALLOW setting to explicitly grant access to a small number of machines. We recommend this.

- Let any machine join your pool. Only the central manager has administrative access (this is the default that ships with Condor)

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA to join or view the pool. The central manager is the only machine with ADMINISTRATOR access.

```
HOSTALLOW_READ = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and the U of I Math department join the pool, EXCEPT do **not** allow lab machines to do so. Also, do not allow the 177.55 subnet (perhaps this is the dial-in subnet). Allow anyone to view pool statistics. The machine named bigcheese administers the pool (not the central manager).

```
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
HOSTDENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
HOSTALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and UW-Madison's CS department to view the pool. Only NCSA machines and the machine raven.cs.wisc.edu can join the pool. (Note: the machine raven has the read access it needs through the wild card setting in HOSTALLOW_READ). This example also shows how to use “\” to continue a long list of machines onto multiple lines, making it more readable (this works for all configuration file entries, not just host access entries)

```
HOSTALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
    biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Allow anyone except the military to view the status of the pool, but only let machines at NCSA view the job queues. Only NCSA machines can join the pool. The central manager, bigcheese, and biggercheese can perform most administrative functions. However, only biggercheese can update user priorities.

```
HOSTDENY_READ = *.mil
HOSTALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
    biggercheese.uiuc.edu
HOSTALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

A new security feature introduced in Condor version 6.3.2 enables more fine-grained control over the configuration settings that can be modified remotely with the *condor_config_val* command. The manual page for *condor_config_val* on page 700 details how to use *condor_config_val* to modify configuration settings remotely. Since certain configuration attributes can have a large impact on the functioning of the Condor system and the security of the machines in a Condor pool, it is important to restrict the ability to change attributes remotely.

For each security access level described, the Condor administrator can define which configuration settings a host at that access level is allowed to change. Optionally, the administrator can define separate lists of settable attributes for each Condor daemon, or the administrator can define one list that is used by all daemons.

For each command that requests a change in configuration setting, Condor searches all the different possible security access levels to see which, if any, the request satisfies. (Some hosts can qualify for multiple access levels. For example, any host with ADMINISTRATOR permission probably has WRITE permission also). Within the qualified access level, Condor searches for the list of attributes that may be modified. If the request is covered by the list, the request will be granted. If not covered, the request will be refused.

The default configuration shipped with Condor is exceedingly restrictive. Condor users or administrators cannot set configuration values from remote hosts with *condor_config_val*. Enabling this feature requires a change to the settings in the configuration file. Use this security feature carefully. Grant access only for attributes which you need to be able to modify in this manner, and grant access only at the most restrictive security level possible.

The most secure use of this feature allows Condor users to set attributes in the configuration file which are not used by Condor directly. These are custom attributes published by various Condor daemons with the <SUBSYS>_ATTRS setting described in section 3.3.5 on page 163. It is secure to grant access only to modify attributes that are used by Condor to publish information. Granting access to modify settings used to control the behavior of Condor is not secure. The goal is to ensure no one can use the power to change configuration attributes to compromise the security of your Condor pool.

The control lists are defined by configuration settings that contain SETTABLE_ATTRS in their name. The name of the control lists have the following form:

```
<SUBSYS>_SETTABLE_ATTRS_PERMISSION-LEVEL
```

The two parts of this name that can vary are PERMISSION-LEVEL and the <SUBSYS>. The PERMISSION-LEVEL can be any of the security access levels described earlier in this section. Examples include WRITE, OWNER, and CONFIG.

The <SUBSYS> is an optional portion of the name. It can be used to define separate rules for which configuration attributes can be set for each kind of Condor daemon (for example, STARTD, SCHEDD, MASTER). There are many configuration settings that can be defined differently for each daemon that use this <SUBSYS> naming convention. See section 3.3.1 on page 147 for a list. If there is no daemon-specific value for a given daemon, Condor will look for SETTABLE_ATTRS_PERMISSION-LEVEL.

Each control list is defined by a comma-separated list of attribute names which should be allowed to be modified. The lists can contain wild cards characters ('*').

Some examples of valid definitions of control lists with explanations:

- `SETTABLE_ATTRS_CONFIG = *`

Grant unlimited access to modify configuration attributes to any request that came from a machine in the `CONFIG` access level. This was the default behavior before Condor version 6.3.2.

- `SETTABLE_ATTRS_ADMINISTRATOR = *_DEBUG, MAX_*_LOG`

Grant access to change any configuration setting that ended with “_DEBUG” (for example, `STARTD_DEBUG`) and any attribute that matched “MAX_*_LOG” (for example, `MAX_SCHEDD_LOG`) to any host with `ADMINISTRATOR` access.

- `STARTD_SETTABLE_ATTRS_OWNER = HasDataSet`

Allows any request to modify the `HasDataSet` attribute that came from a host with `OWNER` access. By default, `OWNER` covers any request originating from the local host, plus any machines listed in the `ADMINISTRATOR` level. Therefore, any Condor job would qualify for `OWNER` access to the machine where it is running. So, this setting would allow any process running on a given host, including a Condor job, to modify the `HasDataSet` variable for that host. `HasDataSet` is not used by Condor, it is an invented attribute included in the `STARTD_ATTRS` setting in order for this example to make sense.

3.6.10 Using Condor w/ Firewalls, Private Networks, and NATs

This topic is now addressed in more detail in section 3.7, which explains network communication in Condor.

3.6.11 User Accounts in Condor

On a Unix system, `UIDs` (User IDentification numbers) form part of an operating system’s tools for maintaining access control. Each executing program has a `UID`, a unique identifier of a user executing the program. This is also called the real `UID`. A common situation has one user executing the program owned by another user. Many system commands work this way, with a user (corresponding to a person) executing a program belonging to (owned by) `root`. Since the program may require privileges that `root` has which the user does not have, a special bit in the program’s protection specification (a `setuid` bit) allows the program to run with the `UID` of the program’s owner, instead of the user that executes the program. This `UID` of the program’s owner is called an effective `UID`.

Condor works most smoothly when its daemons run as `root`. The daemons then have the ability to switch their effective `UIDs` at will. When the daemons run as `root`, they normally leave their effective `UID` and `GID` (Group IDentification) to be those of user and group `condor`. This allows

access to the log files without changing the ownership of the log files. It also allows access to these files when the user `condor`'s home directory resides on an NFS server. `root` can not normally access NFS files.

If there is no `condor` user and group on the system, an administrator can specify which UID and GID the Condor daemons should use when they do not need root privileges in two ways: either with the `CONDOR_IDS` environment variable or the `CONDOR_IDS` configuration file setting. In either case, the value should be the UID integer, followed by a period, followed by the GID integer. For example, if a Condor administrator does not want to create a `condor` user, and instead wants their Condor daemons to run as the `daemon` user (a common non-root user for system daemons to execute as), the `daemon` user's UID was 2, and group `daemon` had a GID of 2, the corresponding setting in the Condor configuration file would be `CONDOR_IDS = 2.2`.

On a machine where a job is submitted, the `condor_schedd` daemon changes its effective UID to `root` such that it has the capability to start up a `condor_shadow` daemon for the job. Before a `condor_shadow` daemon is created, the `condor_schedd` daemon switches back to `root`, so that it can start up the `condor_shadow` daemon with the (real) UID of the user who submitted the job. Since the `condor_shadow` runs as the owner of the job, all remote system calls are performed under the owner's UID and GID. This ensures that as the job executes, it can access only files that its owner could access if the job were running locally, without Condor.

On the machine where the job executes, the job runs either as the submitting user or as user `nobody`, to help ensure that the job cannot access local resources or do harm. If the `UID_DOMAIN` matches, and the user exists as the same UID in password files on both the submitting machine and on the execute machine, the job will run as the submitting user. If the user does not exist in the execute machine's password file and `SOFT_UID_DOMAIN` is `True`, then the job will run under the submitting user's UID anyway (as defined in the submitting machine's password file). If `SOFT_UID_DOMAIN` is `False`, and `UID_DOMAIN` matches, and the user is not in the execute machine's password file, then the job execution attempt will be aborted.

Running Condor as Non-Root

While we strongly recommend starting up the Condor daemons as `root`, we understand that it is not always possible to do so. The main problems appear when one Condor installation is shared by many users on a single machine, or if machines are set up to only execute Condor jobs. With a submit-only installation for a single user, there is no need for (or benefit from) running as `root`.

What follows are the effects on the various parts of Condor of running both with and without root access.

condor_startd If you're setting up a machine to run Condor jobs and don't start the `condor_startd` as `root`, you're basically relying on the goodwill of your Condor users to agree to the policy you configure the `condor_startd` to enforce as far as starting, suspending, vacating and killing Condor jobs under certain conditions. If you run as `root`, however, you can enforce these policies regardless of malicious users. By running as `root`, the Condor daemons run with a different UID than the Condor job that gets started (since the user's job is started as either

the UID of the user who submitted it, or as user `nobody`, depending on the `UID_DOMAIN` settings). Therefore, the Condor job cannot do anything to the Condor daemons. If you don't start the daemons as `root`, all processes started by Condor, including the end user's job, run with the same UID (since you can't switch UIDs unless you're `root`). Therefore, a user's job could just kill the `condor_startd` and `condor_starter` as soon as it starts up and by doing so, avoid getting suspended or vacated when a user comes back to the machine. This is nice for the user, since they get unlimited access to the machine, but awful for the machine owner or administrator. If you trust the users submitting jobs to Condor, this might not be a concern. To ensure, however, that the policy you choose is effectively enforced by Condor, the `condor_startd` should be started as `root`.

In addition, some system information cannot be obtained without `root` access on some platforms (such as load average on IRIX). As a result, when running without `root` access, the `condor_startd` must call other programs (for example, `uptime`) to get this information. This is much less efficient than getting the information directly from the kernel (which is what we do if we're running as `root`). On Linux and Solaris, we can get this information directly without `root` access, so this is not a concern on those platforms.

If you cannot have all of Condor running as `root`, at least consider whether you can install the `condor_startd` as `setuid root`. That would solve both of these problems. If you cannot do that, you could also install it as a `setgid sys` or `kmem` program (depending on whatever group has read access to `/dev/kmem` on your system), and that would at least solve the system information problem.

condor_schedd The biggest problem running the `condor_schedd` without `root` access is that the `condor_shadow` processes which it spawns are stuck with the same UID the `condor_schedd` has. This means that users submitting their jobs must go out of their way to grant write access to user or group `condor` (or whoever the `condor_schedd` is running as) for any files or directories their jobs write or create. Similarly, read access must be granted to their input files.

Consider installing `condor_submit` as a `setgid condor` program so that at least the `stdout`, `stderr` and `UserLog` files get created with the right permissions. If `condor_submit` is a `setgid` program, it will automatically set its `umask` to `002`, and create group-writable files. This way, the simple case of a job that only writes to `stdout` and `stderr` will work. If users have programs that open their own files, they will need to know and set the proper permissions on the directories they submit from.

condor_master The `condor_master` is what spawns the `condor_startd` and `condor_schedd`. To have both running as `root`, have the `condor_master` run as `root`. This happens automatically if you start the master from your boot scripts.

condor_negotiator* and *condor_collector There is no need to have either of these daemons running as `root`.

condor_kbdd On platforms that need the `condor_kbdd` (Digital Unix and IRIX) the `condor_kbdd` must run as `root`. If it is started as any other user, it will not work. You might consider installing this program as a `setuid root` binary if you cannot run the `condor_master` as `root`. Without the `condor_kbdd`, the `startd` has no way to monitor mouse activity at all, and the only keyboard activity it will notice is activity on ttys (such as `xterms`, remote logins, etc).

If you do choose to run Condor as non-root, then you may choose almost any user you like. A common choice is to use the `condor` user; this simplifies the setup because Condor will look for its configuration files in the `condor` user's directory. If you do not select the `condor` user, then you will need to ensure that the configuration is set properly so that Condor can find its configuration files.

If users will be submitting jobs as a user different than the user Condor is running as (perhaps you are running as the `condor` user and users are submitting as themselves), then users have to be careful to only have file permissions properly set up to be accessible by the user Condor is using. In practice, this means creating world-writable directories for output from Condor jobs. This creates a potential security risk, in that any user on the machine where the job is submitted can alter the data, remove it, or do other undesirable things. It is only acceptable in an environment where users can trust other users.

Normally, users without root access who wish to use Condor on their machines create a `condor` home directory somewhere within their own accounts and start up the daemons (to run with the UID of the user). As in the case where the daemons run as user `condor`, there is no ability to switch UIDs or GIDs. The daemons run as the UID and GID of the user who started them. On a machine where jobs are submitted, the `condor_shadow` daemons all run as this same user. But if other users are using Condor on the machine in this environment, the `condor_shadow` daemons for these other users' jobs execute with the UID of the user who started the daemons. This is a security risk, since the Condor job of the other user has access to all the files and directories of the user who started the daemons. Some installations have this level of trust, but others do not. Where this level of trust does not exist, it is best to set up a `condor` account and group, or to have each user start up their own Personal Condor submit installation.

When a machine is an execution site for a Condor job, the Condor job executes with the UID of the user who started the `condor_startd` daemon. This is also potentially a security risk, which is why we do not recommend starting up the execution site daemons as a regular user. Use either `root` or a user (such as the user `condor`) that exists only to run Condor jobs.

Running Jobs as the Nobody User

Under Unix, Condor runs jobs either as the user that submitted the jobs, or as the user called `nobody`. Condor uses user `nobody` if the value of the `UID_DOMAIN` configuration variable of the submitting and executing machines are different or if `STARTER_ALLOW_RUNAS_OWNER` is false or if the job ClassAd contains `RunAsOwner=False`. Under Windows, Condor by default runs jobs under a dynamically created local account that exists for the duration of the job, but it can optionally run the job as the user account that owns the job if `STARTER_ALLOW_RUNAS_OWNER` is True and the job contains `RunAsOwner=True`.

When Condor cleans up after executing a vanilla universe job, it does the best that it can by deleting all of the processes started by the job. During the life of the job, it also does its best to track the CPU usage of all processes created by the job. There are a variety of mechanisms used by Condor to detect all such processes, but, in general, the only foolproof mechanism is for the job to run under a dedicated execution account (as it does under Windows by default). With all other mechanisms, it

is possible to fool Condor, and leave processes behind after Condor has cleaned up. In the case of a shared account, such as the Unix user `nobody`, it is possible for the job to leave a lurker process lying in wait for the next job run as `nobody`. The lurker process may prey maliciously on the next `nobody` user job, wreaking havoc.

Condor could prevent this problem by simply killing all processes run by the `nobody` user, but this would annoy many system administrators. The `nobody` user is often used for non-Condor system processes. It may also be used by other Condor jobs running on the same machine, if it is a multi-processor machine.

Condor provides a two-part solution to this difficulty. First, create user accounts specifically for Condor to use instead of user `nobody`. These can be low-privilege accounts, as the `nobody` user is. Create one of these accounts for each job execution slot per computer, so that distinct users can be used for concurrent processes. This prevents malicious behavior between processes running on distinct slots. Section 3.12.7 details slots. For a sample machine with two compute slots, create two users that are intended only to be used by Condor. As an example, call them `cndrusr1` and `cndrusr2`. Tell Condor about these users with the `SLOTx_USER` configuration variables, where `x` is replaced with the slot number. In this example:

```
SLOT1_USER = cndrusr1
SLOT2_USER = cndrusr2
```

Then tell Condor that these accounts are intended only to be used by Condor, so Condor can kill all the processes belonging to these users upon job completion. The configuration variable `DEDICATED_EXECUTE_ACCOUNT_REGEX` is introduced and set to a regular expression that matches the account names we have just created.

```
DEDICATED_EXECUTE_ACCOUNT_REGEX = cndrusr[0-9] +
```

Finally, tell Condor not to run jobs as the job owner:

```
STARTER_ALLOW_RUNAS_OWNER = False
```

Notes:

1. Currently, none of these configuration settings apply to standard universe jobs. Normally, standard universe jobs do not create additional processes.
2. On Windows, `SLOTx_USER` will only work if the credential of the specified user is stored on the execute machine using `condor_store_cred`. See the `condor_store_cred` manual page (in section 9) for details of this command. However, the default behavior in Windows is to run jobs under a dynamically created dedicated execution account, so just using the default behavior is sufficient to avoid problems with lurker processes.

3. You can tell if the starter is in fact treating the account as a dedicated account, because it will print a line such as the following in its log file:

```
Tracking process family by login "cndrusr1"
```

Working Directories for Jobs

Every executing process has a notion of its current working directory. This is the directory that acts as the base for all file system access. There are two current working directories for any Condor job: one where the job is submitted and a second where the job executes. When a user submits a job, the submit-side current working directory is the same as for the user when the *condor_submit* command is issued. The **initialdir** submit command may change this, thereby allowing different jobs to have different working directories. This is useful when submitting large numbers of jobs. This submit-side current working directory remains unchanged for the entire life of a job. The submit-side current working directory is also the working directory of the *condor_shadow* daemon. This is particularly relevant for standard universe jobs, since file system access for the job goes through the *condor_shadow* daemon, and therefore all accesses behave as if they were executing without Condor.

There is also an execute-side current working directory. For standard universe jobs, it is set to the *execute* subdirectory of Condor's home directory. This directory is world-writable, since a Condor job usually runs as user *nobody*. Normally, standard universe jobs would never access this directory, since all I/O system calls are passed back to the *condor_shadow* daemon on the submit machine. In the event, however, that a job crashes and creates a core dump file, the execute-side current working directory needs to be accessible by the job so that it can write the core file. The core file is moved back to the submit machine, and the *condor_shadow* daemon is informed. The *condor_shadow* daemon sends e-mail to the job owner announcing the core file, and provides a pointer to where the core file resides in the submit-side current working directory.

3.6.12 Privilege Separation

Section 3.6.11 discusses why, under most circumstances, it is beneficial to run the Condor daemons as *root*. In situations where multiple users are involved or where Condor is responsible for enforcing a machine owner's policy, running as *root* is the *only* way for Condor to do its job correctly and securely.

Unfortunately, this requirement of running Condor as *root* is at odds with a well-established goal of security-conscious administrators: keeping the amount of software that runs with superuser privileges to a minimum. Condor's nature as a large distributed system that routinely communicates with potentially untrusted components over the network further aggravates this goal.

The privilege separation (PrivSep) effort in Condor aims to minimize the amount of code that needs *root*-level access, while still giving Condor the tools it needs to work properly. Note that PrivSep is currently only available for execute side functionality, and is not implemented on Windows.

In the PrivSep model, all logic in Condor that requires superuser privilege is contained in a small component called the PrivSep Kernel. The Condor daemons execute as an unprivileged account. They explicitly request action from the PrivSep Kernel whenever `root`-level operations are needed.

The PrivSep model then prevents the following attack scenario. In the attack scenario, an attacker has found an exploit in the `condor_startd` that allows for execution of arbitrary code on that daemon's behalf. This gives the attacker `root` access and therefore control over any machine on which the `condor_startd` is running as `root` and the exploit can be exercised. Under the PrivSep model, the `condor_startd` no longer runs as `root`. This prevents the attacker from taking arbitrary action as `root`. Further, limits on requested actions from the PrivSep Kernel contain and restrict the attacker's sphere of influence.

The following section describes the configuration necessary to enable PrivSep for an execute-side Condor installation. After this is a detailed description of the services that the PrivSep Kernel provides to Condor, and how it limits the allowed `root`-level actions.

PrivSep Configuration

The PrivSep Kernel is implemented as two programs: the `condor_root_switchboard` and the `condor_procd`. Both are contained in the `sbin` directory of the Condor distribution. When Condor is running in PrivSep mode, these are to be the only two Condor daemons that run with `root` privilege.

Each of these binaries must be accessible on the file system via a *trusted path*. A trusted path ensures that no user (other than `root`) can alter the binary or path to the binary referred to. To ensure that the paths to these binaries are trusted, use only `root`-owned directories, and set the permissions on these directories to deny write access to all but `root`. The binaries themselves must also be owned by `root` and not writable by any other. The `condor_root_switchboard` program additionally is installed with the `setuid` bit set. The following command properly sets the permissions on the `condor_root_switchboard` binary:

```
chmod 4755 /opt/condor/release/sbin/condor_root_switchboard
```

The PrivSep Kernel has its own configuration file. This file must be `/etc/condor/privsep_config`. The format of this file is different than a Condor configuration file. It consists of lines with “key = value” pairs. Lines with only white space or lines with “#” as the first non-white space character are ignored.

In the PrivSep Kernel configuration file, some configuration settings are interpreted as single values, while others are interpreted as lists. To populate a list with multiple values, use multiple lines with the same key. For example, the following configures the `valid-dirs` setting as a list with two entries:

```
valid-dirs = /opt/condor/execute_1
valid-dirs = /opt/condor/execute_2
```

It is an error to have multiple lines with the same key for a setting that is not interpreted as a list.

Some PrivSep Kernel configuration file settings require a list of UIDs or GIDs, and these allow for a more specialized syntax. User and group IDs can be specified either numerically or textually. Multiple list entries may be given on a single line using the `:` (colon) character as a delimiter. In addition, list entries may specify a range of IDs using a `-` (dash) character to separate the minimum and maximum IDs included. The `*` (asterisk) character on the right-hand side of such a range indicates that the range extends to the maximum possible ID. The following example builds a complex list of IDs:

```
valid-target-uids = nobody : nfsuser1 : nfsuser2
valid-target-uids = condor_run_1 - condor_run_8
valid-target-uids = 800 - *
```

If `condor_run_1` maps to UID 701, and `condor_run_8` maps to UID 708, then this range specifies the 8 UIDs of 701 through 708 (inclusive).

The following settings are required to configure the PrivSep Kernel:

- `valid-caller-uids` and `valid-caller-gids`. These lists specify users and groups that will be allowed to request action from the PrivSep Kernel. The list typically will contain the UID and primary GID that the Condor daemons will run as.
- `valid-target-uids` and `valid-target-gids`. These lists specify the users and groups that Condor will be allowed to act on behalf of. The list will need to include IDs of all users and groups that Condor jobs may use on the given execute machine.
- `valid-dirs`. This list specifies directories that Condor will be allowed to manage for the use of temporary job files. Normally, this will only need to include the value of Condor's `$(EXECUTE)` directory. Any entry in this list must be a trusted path. This means that all components of the path must be directories that are `root`-owned and only writable by `root`. For many sites, this may require a change in ownership and permissions to the `$(LOCAL_DIR)` and `$(EXECUTE)` directories. Note also that the PrivSep Kernel does not have access to Condor's configuration variables, and therefore may not refer to them in this file.
- `procd-executable`. A (trusted) full path to the `condor_procd` executable. Note that the PrivSep Kernel does not have access to Condor's configuration variables, and therefore may not refer to them in this file.

Here is an example of a full `privsep_config` file. This file gives the `condor` account access to the PrivSep Kernel. Condor's use of this execute machine will be restricted to a set of eight dedicated accounts, along with the `users` group. Condor's `$(EXECUTE)` directory and the `condor_procd` executable are also specified, as required.

```
valid-caller-uids = condor
valid-caller-gids = condor
valid-target-uids = condor_run_1 - condor_run_8
```

```
valid-target-gids = users : condor_run_1 - condor_run_8
valid-dirs = /opt/condor/local/execute
procd-executable = /opt/condor/release/sbin/condor_procd
```

Once the PrivSep Kernel is properly installed and configured, Condor's configuration must be updated to specify that PrivSep should be used. The Condor configuration variable `PRIVSEP_ENABLED` is a boolean flag serving this purpose. In addition, Condor must be told where the *condor_root_switchboard* binary is located using the `PRIVSEP_SWITCHBOARD` setting. The following example illustrates:

```
PRIVSEP_ENABLED = True
PRIVSEP_SWITCHBOARD = $(SBIN)/condor_root_switchboard
```

Finally, note that while the *condor_procd* is in general an optional component of Condor, it is required when PrivSep is in use. If `PRIVSEP_ENABLED` is `True`, the *condor_procd* will be used regardless of the `USE_PROCD` setting. Details on these Condor configuration variables are in section 3.3.27 for PrivSep variables and section 3.3.18 for *condor_procd* variables.

PrivSep Kernel Interface

This section describes the `root`-enabled operations that the PrivSep Kernel makes available to Condor. The PrivSep Kernel's interface is designed to provide only operations needed by Condor in order to function properly. Each operation is further restricted based on the PrivSep Kernel's configuration settings.

The following list describes each action that can be performed via the PrivSep Kernel, along with the limitations enforced on how it may be used. The terms *valid target users*, *valid target groups*, and *valid directories* refer respectively to the settings for `valid-target-uids`, `valid-target-gids`, and `valid-dirs` from the PrivSep Kernel's configuration.

- *Make a directory as a user.* This operation creates an empty directory, owned by a user. The user must be a valid target user, and the new directory's parent must be a valid directory.
- *Change ownership of a directory tree.* This operation involves recursively changing ownership of all files and subdirectories contained in a given directory. The directory's parent must be a valid directory, and the new owner must either be a valid target user or the user invoking the PrivSep Kernel.
- *Remove a directory tree.* This operation deletes a given directory, including everything contained within. The directory's parent must be a valid directory.
- *Execute a program as a user.* Condor can invoke the PrivSep kernel to execute a program as a valid target user. The user's primary group and any supplemental groups that it is a member of must all be valid target groups. This operation may also include opening files for standard input, output, and error before executing the program.

After launching a program as a valid target user, the PrivSep Kernel allows Condor limited control over its execution. The following operations are supported on a program executed via the PrivSep Kernel:

- *Get resource usage information.* This allows Condor to gather usage statistics such as CPU time and memory image size. This applies to the program's initial process and any of its descendants.
- *Signal the program.* Condor may ask that signals be sent to the program's initial process as a notification mechanism.
- *Suspend and resume the program.* These operations send SIGSTOP or SIGCONT signals to all processes that make up the program.
- *Kill the process and all descendants.* Condor is allowed to terminate the execution of the program or any processes left behind when the program completes.

By sufficiently constraining the valid target accounts and valid directories to which the PrivSep Kernel allows access, the ability of a compromised Condor daemon to do damage can be considerably reduced.

3.6.13 Support for *glexec*

glexec is a tool that provides a sudo-like capability in a grid environment. *glexec* takes an X.509 proxy and a command to run as inputs, and maps the proxy to a local identity (that is, a Unix UID), which it then uses to execute the command. Like the *condor_root_switchboard* command, which provides similar functionality for Condor's PrivSep mode (see section 3.6.12), *glexec* must be installed as a root-owned setuid program. See <http://www.nikhef.nl/grid/lcaslcmaps/glexec/> for more information about *glexec*.

Condor can interoperate with *glexec*, using it in a similar way to how the *condor_root_switchboard* is used when running Condor in PrivSep mode. The *condor_starter* uses *glexec* when launching a job, in order to give the job a separate UID from that of the Condor daemons. *glexec* is also used when performing maintenance actions such as cleaning up a job's files and processes, which cannot be done well directly under the Condor daemons' UID due to permissions. A consequence of this type of integration with *glexec* is that the execution of a single Condor job results in several *glexec* invocations, and each must map the proxy to the same UID. It is thus important to ensure that *glexec* is configured to provide this guarantee.

Configuration for *glexec* support is straightforward. The boolean configuration variable `GLEEXEC_JOB` must be set `True` on execute machines where *glexec* is to be used. Condor also must be given the full path to the *glexec* binary using the `GLEEXEC` configuration variable. Note that Condor must be started as a non-root user when *glexec* is used. This is because when Condor runs as root, it can perform actions as other UIDs arbitrarily, and *glexec*'s services are not needed. Finally, for a job to execute properly in the mode utilizing *glexec*, the job must be submitted with a proxy

specified via the **x509userproxy** command in its submit description file, since a proxy is needed as input to *glexec*.

Earlier versions of Condor employed a different form of *glexec* support, where the *condor_starter* daemon ran under the same UID as the job. This feature was enabled using the `GLEEXEC_STARTER` configuration variable. This configuration variable is no longer used, and it is replaced by the `GLEEXEC_JOB` configuration variable, to enable usage of *glexec*.

3.7 Networking (includes sections on Port Usage and GCB)

This section on network communication in Condor discusses which network ports are used, how Condor behaves on machines with multiple network interfaces and IP addresses, and how to facilitate functionality in a pool that spans firewalls and private networks.

The security section of the manual contains some information that is relevant to the discussion of network communication which will not be duplicated here, so please see section 3.6 as well.

Firewalls, private networks, and network address translation (NAT) pose special problems for Condor. There are currently two main mechanisms for dealing with firewalls within Condor:

1. Restrict Condor to use a specific range of port numbers, and allow connections through the firewall that use any port within the range.
2. Use *Generic Connection Brokering* (GCB).

Each method has its own advantages and disadvantages, as described below.

3.7.1 Port Usage in Condor

Default Port Usage

Every Condor daemon listens on a network port for incoming commands. Most daemons listen on a dynamically assigned port. In order to send a message, Condor daemons and tools locate the correct port to use by querying the *condor_collector*, extracting the port number from the ClassAd. One of the attributes included in every daemon's ClassAd is the full IP address and port number upon which the daemon is listening.

To access the *condor_collector* itself, all Condor daemons and tools must know the port number where the *condor_collector* is listening. The *condor_collector* is the only daemon with a well-known, fixed port. By default, Condor uses port 9618 for the *condor_collector* daemon. However, this port number can be changed (see below).

As an optimization for daemons and tools communicating with another daemon that is running on the same host, each Condor daemon can be configured to write its IP address and port num-

ber into a well-known file. The file names are controlled using the `<SUBSYS>_ADDRESS_FILE` configuration variables, as described in section 3.3.5 on page 162.

NOTE: In the 6.6 stable series, and Condor versions earlier than 6.7.5, the *condor_negotiator* also listened on a fixed, well-known port (the default was 9614). However, beginning with version 6.7.5, the *condor_negotiator* behaves like all other Condor daemons, and publishes its own ClassAd to the *condor_collector* which includes the dynamically assigned port the *condor_negotiator* is listening on. All Condor tools and daemons that need to communicate with the *condor_negotiator* will either use the `NEGOTIATOR_ADDRESS_FILE` or will query the *condor_collector* for the *condor_negotiator*'s ClassAd.

Sites that configure any checkpoint servers will introduce other fixed ports into their network. Each *condor_ckpt_server* will listen to 4 fixed ports: 5651, 5652, 5653, and 5654. There is currently no way to configure alternative values for any of these ports.

Using a Non Standard, Fixed Port for the *condor_collector*

By default, Condor uses port 9618 for the *condor_collector* daemon. To use a different port number for this daemon, the configuration variables that tell Condor these communication details are modified. Instead of

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST)
```

the configuration might be

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST):9650
```

If a non standard port is defined, the same value of `COLLECTOR_HOST` (including the port) must be used for all machines in the Condor pool. Therefore, this setting should be modified in the global configuration file (`condor_config` file), or the value must be duplicated across all configuration files in the pool if a single configuration file is not being shared.

When querying the *condor_collector* for a remote pool that is running on a non standard port, any Condor tool that accepts the **-pool** argument can optionally be given a port number. For example:

```
% condor_status -pool foo.bar.org:1234
```

Using a Dynamically Assigned Port for the *condor_collector*

On single machine pools, it is permitted to configure the *condor_collector* daemon to use a dynamically assigned port, as given out by the operating system. This prevents port conflicts with other services on the same machine. However, a dynamically assigned port is only to be used on single

machine Condor pools, and only if the `COLLECTOR_ADDRESS_FILE` configuration variable has also been defined. This mechanism allows all of the Condor daemons and tools running on the same machine to find the port upon which the *condor_collector* daemon is listening, even when this port is not defined in the configuration file and is not known in advance.

To enable the *condor_collector* daemon to use a dynamically assigned port, the port number is set to 0 in the `COLLECTOR_HOST` variable. The `COLLECTOR_ADDRESS_FILE` configuration variable must also be defined, as it provides a known file where the IP address and port information will be stored. All Condor clients know to look at the information stored in this file. For example:

```
COLLECTOR_HOST = $(CONDOR_HOST):0
COLLECTOR_ADDRESS_FILE = $(LOG)/.collector_address
```

NOTE: Using a port of 0 for the *condor_collector* and specifying a `COLLECTOR_ADDRESS_FILE` only works in Condor version 6.6.8 or later in the 6.6 stable series, and in version 6.7.4 or later in the 6.7 development series. Do not attempt to do this with older versions of Condor.

Configuration definition of `COLLECTOR_ADDRESS_FILE` is in section 3.3.5 on page 162, and `COLLECTOR_HOST` is in section 3.3.3 on page 150.

Restricting Port Usage to Operate with Firewalls

If a Condor pool is completely behind a firewall, then no special consideration or port usage is needed. However, if there is a firewall between the machines within a Condor pool, then configuration variables may be set to force the usage of specific ports, and to utilize a specific range of ports.

By default, Condor uses port 9618 for the *condor_collector* daemon, and dynamic (apparently random) ports for everything else. See section 3.7.1, if a dynamically assigned port is desired for the *condor_collector* daemon.

The configuration variables `HIGHPORT` and `LOWPORT` facilitate setting a restricted range of ports that Condor will use. This may be useful when some machines are behind a firewall. The configuration macros `HIGHPORT` and `LOWPORT` will restrict dynamic ports to the range specified. The configuration variables are fully defined in section 3.3.6. All of these ports must be greater than 0 and less than 65,536. Note that both `HIGHPORT` and `LOWPORT` must be at least 1024 for Condor version 6.6.8. In general, use ports greater than 1024, in order to avoid port conflicts with standard services on the machine. Another reason for using ports greater than 1024 is that daemons and tools are often not run as `root`, and only `root` may listen to a port lower than 1024. Also, the range must include enough ports that are not in use, or Condor cannot work.

The range of ports assigned may be restricted based on incoming (listening) and outgoing (connect) ports with the configuration variables `IN_HIGHPORT`, `IN_LOWPORT`, `OUT_HIGHPORT`, and `OUT_LOWPORT`. See section 3.3.6 for complete definitions of these configuration variables. A range of ports lower than 1024 for daemons running as `root` is appropriate for incoming ports, but not for outgoing ports. The use of ports below 1024 (versus above 1024) has security implications; therefore, it is inappropriate to assign a range that crosses the 1024 boundary.

NOTE: Setting `HIGHPORT` and `LOWPORT` will not automatically force the *condor_collector* to bind to a port within the range. The only way to control what port the *condor_collector* uses is by setting the `COLLECTOR_HOST` (as described above).

The total number of ports needed depends on the size of the pool, the usage of the machines within the pool (which machines run which daemons), and the number of jobs that may execute at one time. Here we discuss how many ports are used by each participant in the system.

The central manager of the pool needs $5 + \text{NEGOTIATOR_SOCKET_CACHE_SIZE}$ ports for daemon communication, where `NEGOTIATOR_SOCKET_CACHE_SIZE` is specified in the configuration or defaults to the value 16.

Each execute machine (those machines running a *condor_startd* daemon) requires $5 + (5 * \text{number of slots advertised by that machine})$ ports. By default, the number of slots advertised will equal the number of physical CPUs in that machine.

Submit machines (those machines running a *condor_schedd* daemon) require $5 + (5 * \text{MAX_JOBS_RUNNING})$ ports. The configuration variable `MAX_JOBS_RUNNING` limits (on a per-machine basis, if desired) the maximum number of jobs. Without this configuration macro, the maximum number of jobs that could be simultaneously executing at one time is a function of the number of reachable execute machines.

Also be aware that `HIGHPORT` and `LOWPORT` only impact dynamic port selection used by the Condor system, and they do not impact port selection used by jobs submitted to Condor. Thus, jobs submitted to Condor that may create network connections may not work in a port restricted environment. For this reason, specifying `HIGHPORT` and `LOWPORT` is not going to produce the expected results if a user submits jobs to be executed under the MPI job universe.

Where desired, a local configuration for machines *not* behind a firewall can override the usage of `HIGHPORT` and `LOWPORT`, such that the ports used for these machines are not restricted. This can be accomplished by adding the following to the local configuration file of those machines *not* behind a firewall:

```
HIGHPORT = UNDEFINED
LOWPORT  = UNDEFINED
```

If the maximum number of ports allocated using `HIGHPORT` and `LOWPORT` is too few, socket binding errors of the form

```
failed to bind any port within <$LOWPORT> - <$HIGHPORT>
```

are likely to appear repeatedly in log files.

Multiple Collectors

This section has not yet been written

Port Conflicts

This section has not yet been written

3.7.2 Configuring Condor for Machines With Multiple Network Interfaces

Condor can run on machines with multiple network interfaces. Starting with Condor version 6.7.13 (and therefore all Condor 6.8 and more recent versions), new functionality is available that allows even better support for multi-homed machines, using the configuration variable `BIND_ALL_INTERFACES`. A multi-homed machine is one that has more than one NIC (Network Interface Card). Further improvements to this new functionality will remove the need for any special configuration in the common case. For now, care must still be given to machines with multiple NICs, even when using this new configuration variable.

Using `BIND_ALL_INTERFACES`

Machines can be configured such that whenever Condor daemons or tools call `bind()`, the daemons or tools use all network interfaces on the machine. This means that outbound connections will always use the appropriate network interface to connect to a remote host, instead of being forced to use an interface that might not have a route to the given destination. Furthermore, sockets upon which a daemon listens for incoming connections will be bound to all network interfaces on the machine. This means that so long as remote clients know the right port, they can use any IP address on the machine and still contact a given Condor daemon.

This functionality is on by default. To disable this functionality, the boolean configuration variable `BIND_ALL_INTERFACES` is defined and set to `False`:

```
BIND_ALL_INTERFACES = FALSE
```

This functionality has limitations. Here are descriptions of the limitations.

Using all network interfaces does not work with Kerberos. Every Kerberos ticket contains a specific IP address within it. Authentication over a socket (using Kerberos) requires the socket to also specify that same specific IP address. Use of `BIND_ALL_INTERFACES` causes outbound connections from a multi-homed machine to originate over any of the interfaces. Therefore, the IP address of the outbound connection and the IP address in the Kerberos ticket will not necessarily match, causing the authentication to fail. Sites using Kerberos authentication on multi-homed machines are strongly encouraged not to enable `BIND_ALL_INTERFACES`, at least until Condor's Kerberos functionality supports using multiple Kerberos tickets together with finding the right one to match the IP address a given socket is bound to.

There is a potential security risk. Consider the following example of a security risk. A multi-homed machine is at a network boundary. One interface is on the public Internet, while

the other connects to a private network. Both the multi-homed machine and the private network machines comprise a Condor pool. If the multi-homed machine enables `BIND_ALL_INTERFACES`, then it is at risk from hackers trying to compromise the security of the pool. Should this multi-homed machine be compromised, the entire pool is vulnerable. Most sites in this situation would run an `sshd` on the multi-homed machine so that remote users who wanted to access the pool could log in securely and use the Condor tools directly. In this case, remote clients do not need to use Condor tools running on machines in the public network to access the Condor daemons on the multi-homed machine. Therefore, there is no reason to have Condor daemons listening on ports on the public Internet, causing a potential security threat.

Only one IP address will be advertised. At present, even though a given Condor daemon will be listening to ports on multiple interfaces, each with their own IP address, there is currently no mechanism for that daemon to advertise all of the possible IP addresses where it can be contacted. Therefore, Condor clients (other Condor daemons or tools) will not necessarily be able to locate and communicate with a given daemon running on a multi-homed machine where `BIND_ALL_INTERFACES` has been enabled.

Currently, Condor daemons can only advertise a single IP address in the ClassAd they send to their *condor_collector*. Condor tools and other daemons only know how to look up a single IP address, and they attempt to use that single IP address when connecting to the daemon. So, even if the daemon is listening on 2 or more different interfaces, each with a separate IP, the daemon must choose what IP address to publicly advertise so that other daemons and tools can locate it.

By default, Condor advertises the IP address of the network interface used to contact the collector, since this is the most likely to be accessible to other processes that query the same collector. The `NETWORK_INTERFACE` setting can still be used to specify the IP address Condor should advertise, even if `BIND_ALL_INTERFACES` is set to `True`. Therefore, some of the considerations described below regarding what interface should be used in various situations still apply when deciding what interface is to be advertised.

Sites that make heavy use of private networks and multi-homed machines should consider if using Generic Connection Brokering, GCB, is right for them. More information about GCB and Condor can be found in section 3.7.3 on page 332.

Central Manager with Two or More NICs

Often users of Condor wish to set up “compute farms” where there is one machine with two network interface cards (one for the public Internet, and one for the private net). It is convenient to set up the “head” node as a central manager in most cases and so here are the instructions required to do so.

Setting up the central manager on a machine with more than one NIC can be a little confusing because there are a few external variables that could make the process difficult. One of the biggest mistakes in getting this to work is that either one of the separate interfaces is not active, or the host/domain names associated with the interfaces are incorrectly configured.

Given that the interfaces are up and functioning, and they have good host/domain names associated with them here is how to configure Condor:

In this example, **farm-server.farm.org** maps to the private interface.

On the central manager's global (to the cluster) configuration file:
CONDOR_HOST = **farm-server.farm.org**

On your central manager's local configuration file:
NETWORK_INTERFACE = ip address of **farm-server.farm.org**
NEGOTIATOR = \$(SBIN)/condor_negotiator
COLLECTOR = \$(SBIN)/condor_collector
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, SCHEDD, STARTD

If your central manager and farm machines are all NT, then you only have vanilla universe and it will work now. However, if you have this setup for UNIX, then at this point, standard universe jobs should be able to function in the pool, but if you did not configure the `UID_DOMAIN` macro to be homogeneous across the farm machines, the standard universe jobs will run as **nobody** on the farm machines.

In order to get vanilla jobs and file server load balancing for standard universe jobs working (under Unix), do some more work both in the cluster you have put together and in Condor to make everything work. First, you need a file server (which could also be the central manager) to serve files to all of the farm machines. This could be NFS or AFS, it does not really matter to Condor. The mount point of the directories you wish your users to use must be the same across all of the farm machines. Now, configure `UID_DOMAIN` and `FILESYSTEM_DOMAIN` to be homogeneous across the farm machines and the central manager. Now, you will have to inform Condor that an NFS or AFS filesystem exists and that is done in this manner. In the global (to the farm) configuration file:

```
# If you have NFS
USE_NFS = True
# If you have AFS
HAS_AFS = True
USE_AFS = True
# if you want both NFS and AFS, then enable both sets above
```

Now, if you've set up your cluster so that it is possible for a machine name to never have a domain name (for example: there is machine name but no fully qualified domain name in `/etc/hosts`), you must configure `DEFAULT_DOMAIN_NAME` to be the domain that you wish to be added on to the end of your host name.

A Client Machine with Multiple Interfaces

If you have a client machine with two or more NICs, then there might be a specific network interface with which you desire a client machine to communicate with the rest of the Condor pool. In this case, in the local configuration file for that machine, place:

NETWORK_INTERFACE = ip address of interface desired

A Checkpoint Server on a Machine with Multiple NICs

If your Checkpoint Server is on a machine with multiple interfaces, the only way to get things to work is if your different interfaces have different host names associated with them, and you set `CKPT_SERVER_HOST` to the host name that corresponds with the IP address you want to use in the global configuration file for your pool. You will still need to specify `NETWORK_INTERFACE` in the local config file for your Checkpoint Server.

3.7.3 Generic Connection Brokering (GCB)

Generic Connection Brokering, or GCB, is a system for managing network connections across private network and firewall boundaries. Condor's Linux releases are linked with GCB, and can use GCB functionality to run jobs (either directly or via flocking) on pools that span public and private networks.

While GCB provides numerous advantages over restricting Condor to use a range of ports which are then opened on the firewall (see section 3.7.1 on page 327), GCB is also a very complicated system, with major implications for Condor's networking and security functionality. Therefore, sites must carefully weigh the advantages and disadvantages of attempting to configure and use GCB before making a decision.

Advantages:

- Better connectivity. GCB works with pools that have multiple private networks (even multiple private networks that use the same IP addresses (for example, 192.168.2.*). GCB also works with sites that use network address translation (NAT).
- More secure. Administrators never need to allow inbound connections through the firewall. With GCB, only outbound connections from behind the firewall must be allowed (which is a standard firewall configuration). It is possible to trade decreased performance for better security, and configure the firewall to only allow outbound connections to a single public IP address.
- Does not require `root` access to any machines. All parts of a GCB system can be run as an unprivileged user, and in the common case, no changes to the firewall configuration are required.

Disadvantages:

- The GCB broker (section 3.7.3 describes the broker) node(s) is a potential failure point to the pool. Any private nodes that want to communicate outside their own network must be

represented by a GCB broker. This machine must be highly reliable, since if the broker is ever down, all inbound communication with the private nodes is impossible. Furthermore, no other Condor services should be run on a GCB broker (for example, the Condor pool's central manager). While it is possible to do so, it is not recommended. In general, no other services should be run on the machine at all, and the host should be dedicated to the task of serving as a GCB broker.

- All Condor nodes behind a given firewall share a single IP address (the public IP address of their GCB broker). All Condor daemons using a GCB broker will advertise themselves with this single IP address, and in some cases, connections to/from those daemons will actually originate at the broker. This has implications for Condor's host/IP based security, and the general level of confusion for users and administrators of the pool. Debugging problems will be more difficult, as any log messages which only print the IP address (not the name and/or port) will become ambiguous. Even log or error messages that include the port will not necessarily be helpful, as it is difficult to correlate ports on the broker with the corresponding private nodes.
- Can not function with Kerberos authentication. Kerberos tickets include the IP address of the machine where they were created. However, when Condor daemons are using GCB, they use a different IP address, and therefore, any attempt to authenticate using Kerberos will fail, as Kerberos will consider this a (poor) attempt to fool it into using an invalid host principle.
- Scalability and performance degradation:
 - Connections are more expensive to establish.
 - In some cases, connections must be forwarded through a proxy server on the GCB broker.
 - Each network port on each private node must correspond to a unique port on the broker host, so there is a fixed limit to how many private nodes a given broker can service (which is a function of the number of ports each private node requires and the total number of available ports on the broker).
 - Each private node must maintain an open TCP connection to its GCB broker. GCB will attempt to recover in the case of the socket being closed, but this means the broker must have at least as many sockets open as there are private nodes.
- It is more complex to configure and debug.

Given the increased complexity, use of GCB requires a careful read of this entire manual section, followed by a thorough installation.

Details of GCB and how it works can be found at the GCB homepage:

<http://www.cs.wisc.edu/condor/gcb>

This information is useful for understanding the technical details of how GCB works, and the various parts of the system. While some of the information is partly out of date (especially the discussion of how to configure GCB) most of the sections are perfectly accurate and worth reading.

Ignore the section on “GCBnize”, which describes how to get a given application to use GCB, as the Linux port of all Condor daemons and tools have already been converted to use GCB.

The rest of this section gives the details for configuring a Condor pool to use GCB. It is divided into the following topics:

- Introduction to the GCB broker
- Configuring the GCB broker
- Spawning a GCB broker (with a *condor_master* or using *initd*)
- How to configure Condor machines to use GCB
- Configuring the GCB routing table
- Implications for Condor’s host/IP security settings
- Implications for other Condor configuration settings

Introduction to the GCB Broker

At the heart of GCB is a logical entity known as a *broker* or *inagent*. In reality, the entity is made up of daemon processes running on the same machine comprised of the *gcb_broker* and a set of *gcb_relay_server* processes, each one spawned by the *gcb_broker*.

Every private network using GCB must have at least one broker to arrange connections. The broker must be installed on a machine that nodes in both the public and the private (firewalled) network can directly talk to. The broker need not be able to initiate connections to the private nodes. It can take advantage of the case where it can initiate connections to the private nodes, and that will improve performance. The broker is generally installed on a machine with multiple network interfaces (on the network boundary) or just outside of a network that allows outbound connections. If the private network contains many hosts, sites can configure multiple GCB brokers, and partition the private nodes so that different subsets of the nodes use different brokers.

For a more thorough explanation of what a GCB broker is, check out: <http://www.cs.wisc.edu/~sschang/firewall/gcb/mechanism.htm>

A GCB broker should generally be installed on a dedicated machine. These are machines that are not running other Condor daemons or services. If running any other Condor service (for example, the central manager of the pool) on the same machine as the GCB broker, all other machines attempting to use this Condor service (for example, to connect to the *condor_collector* or *condor_negotiator*) will incur additional connection costs and latency. It is possible that future versions of GCB and Condor will be able to overcome these limitations, but for now, we recommend that a broker is run on a dedicated machine with no other Condor daemons (except perhaps a single *condor_master* used to spawn the *gcb_broker* daemon, as described below).

In principle, a GCB broker is a network element that functions almost like a router. It allows certain connections through the firewall by redirecting connections or forwarding connections. In

general, it is not a good idea to run a lot of other services on the network elements, especially not services like Condor which can spawn arbitrary jobs. Furthermore, the GCB broker relies on listening to many network ports. If other applications are running on the same host as the broker, problems exist where the broker does not have enough network ports available to forward all the connections that might be required of it. Also, all nodes inside a private network rely on the GCB broker for all incoming communication. For performance reasons, avoid forcing the GCB broker to contend with other processes for system resources, such that it is always available to handle communication requests. There is nothing in GCB or Condor requiring the broker to run on a separate machine, but it is the recommended configuration.

The *gcb_broker* daemon listens on two hard-coded, fixed ports (65432 and 65430). A future version of Condor and GCB will remove this limitation. However, for now, to run a *gcb_broker* on a given host, ensure that ports 65432 and 65430 are not already in use.

If *root* access on a machine where a GCB broker is planned, one good option is to have *initd* configured to spawn (and re-spawn) the *gcb_broker* binary (which is located in the `<release_dir>/libexec` directory). This way, the *gcb_broker* will be automatically restarted on reboots, or in the event that the broker itself crashes or is killed. Without *root* access, use a *condor_master* to manage the *gcb_broker* binary.

Configuring the GCB broker

Since the *gcb_broker* and *gcb_relay_server* are not Condor daemons, they do not read the Condor configuration files. Therefore, they must be configured by other means, namely the environment and through the use of command-line arguments.

There is one required command-line argument for the *gcb_broker*. This argument defines the public IP address this broker will use to represent itself and any private network nodes that are configured to use this broker. This information is defined with **-i xxx.xxx.xxx.xxx** on the command-line when the *gcb_broker* is executed. If the broker is being setup outside the private network, it is likely that the machine will only have one IP address, which is clearly the one to use. However, if the broker is being run on a machine on the network boundary (a multi-homed machine with interfaces into both the private and public networks), be sure to use the IP address of the interface on the public network.

Additionally, specify environment variables to control how the *gcb_broker* (and the *gcb_relay_server* processes it spawns) will behave. Some of these settings can also be specified as command-line arguments to the *gcb_broker*. All of them have reasonable defaults if not defined.

- General daemon behavior

The environment variable `GCB_RELAY_SERVER` defines the full path to the *gcb_relay_server* binary the broker should use. The command-line override for this is **-r /full/path/to/relayserver**. If not set either on the command-line or in the environment, the *gcb_broker* process will search for a program named

`gcb_relay_server` in the same directory where the `gcb_broker` binary is located, and attempt to use that one.

The environment variable `GCB_ACTIVE_TO_CLIENT` is a boolean that defines whether the GCB broker can directly talk to servers running inside the network that it manages. The value must be `yes` or `no`, case sensitive. `GCB_ACTIVE_TO_CLIENT` should be set to `yes` only if this GCB broker is running on a network boundary and can connect to both the private and public nodes. If the broker is running in the public network, it should be left undefined or set to `no`.

- Log file locations

The environment variable `GCB_LOG_DIR` defines a directory to use for all GCB-related log files. If defined, and the per-daemon log file settings (described below) are not defined, the broker will write to `$GCB_LOG_DIR/BrokerLog` and the relay server will write to `$GCB_LOG_DIR/RelayServerLog.<pid>`

The environment variable `GCB_BROKER_LOG` defines the full path for the GCB broker's log file. The command-line override is **-l /full/path/to/log/file**. This definition overrides `GCB_LOG_DIR`.

The environment variable `GCB_RELAY_SERVER_LOG` defines the full path to the GCB relay server's log file. Each relay server writes its own log file, so the actual filename will be: `$GCB_RELAY_SERVER_LOG.<pid>` where `<pid>` is replaced with the process id of the corresponding `gcb_relay_server`. When defined, this setting overrides `GCB_LOG_DIR`.

- Verbose logging

The environment variable `GCB_DEBUG_LEVEL` controls how verbose all the GCB daemon's log files should be. Can be either `fulldebug` (more verbose) or `basic`. This defines logging behavior for all GCB daemons, unless the following daemon-specific settings are defined.

The environment variable `GCB_BROKER_DEBUG` controls verbose logging specifically for the GCB broker. The command-line override for this is **-d level**. Overrides `GCB_DEBUG_LEVEL`.

The environment variable `GCB_RELAY_SERVER_DEBUG` controls verbose logging specifically for the GCB relay server. Overrides `GCB_DEBUG_LEVEL`.

- Maximum log file size

The environment variable `GCB_MAX_LOG` defines the maximum size in bytes of all GCB log files. When the log file reaches this size, the content of the file will be moved to `filename.old`, and a new log is started. This defines logging behavior for all GCB daemons, unless the following daemon-specific settings are used.

The environment variable `GCB_BROKER_MAX_LOG` defines the maximum size in bytes of the GCB broker log file.

The environment variable `GCB_RELAY_SERVER_MAX_LOG` defines the maximum size in bytes of the GCB relay server log file.

Spawning the GCB Broker

There are two ways to spawn the GCB broker:

- Use a *condor_master*.

To spawn the GCB broker with a *condor_master*, here are the recommended *condor_config* settings that will work:

```
# Specify that you only want the master and the broker running
DAEMON_LIST = MASTER, GCB_BROKER

# Define the path to the broker binary for the master to spawn
GCB_BROKER = $(RELEASE_DIR)/libexec/gcb_broker

# Define the path to the release_server binary for the broker to use
GCB_RELAY = $(RELEASE_DIR)/libexec/gcb_relay_server

# Setup the gcb_broker's environment. We use a macro to build up the
# environment we want in pieces, and then finally define
# GCB_BROKER_ENVIRONMENT, the setting that condor_master uses.

# Initialize an empty macro
GCB_BROKER_ENV =

# (recommended) Provide the full path to the gcb_relay_server
GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER=$(GCB_RELAY)

# (recommended) Tell GCB to write all log files into the Condor log
# directory (the directory used by the condor_master itself)
GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_LOG_DIR=$(LOG)
# Or, you can specify a log file separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_LOG=$(LOG)/GCB_Broker_Log
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_LOG=$(LOG)/GCB_RS_Log

# (optional -- only set if true) Tell the GCB broker that it can
# directly connect to machines in the private network which it is
# handling communication for. This should only be enabled if the GCB
# broker is running directly on a network boundary and can open direct
# connections to the private nodes.
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_ACTIVE_TO_CLIENT=yes

# (optional) turn on verbose logging for all of GCB
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_DEBUG_LEVEL=fulldebug
# Or, you can turn this on separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_DEBUG=fulldebug
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_DEBUG=fulldebug

# (optional) specify the maximum log file size (in bytes)
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_MAX_LOG=640000
# Or, you can define this separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_MAX_LOG=640000
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_MAX_LOG=640000

# Finally, set the value the condor_master really uses
GCB_BROKER_ENVIRONMENT = $(GCB_BROKER_ENV)
```

```
# If your Condor installation on this host already has a public
# interface as the default (either because it is the first interface
# listed in this machine's host entry, or because you've already
# defined NETWORK_INTERFACE), you can just use Condor's special macro
# that holds the IP address for this.
GCB_BROKER_IP = $(ip_address)
# Otherwise, you could define it yourself with your real public IP:
# GCB_BROKER_IP = 123.123.123.123

# (required) define the command-line arguments for the broker
GCB_BROKER_ARGS = -i $(GCB_BROKER_IP)
```

Once those settings are in place, either spawn or restart the *condor_master* and the *gcb_broker* should be started. Ensure the broker is running by reading the log file specified with `GCB_BROKER_LOG`, or in `$(LOG)/BrokerLog` if using the default.

- Use *initd*.

The system's *initd* may be used to manage the *gcb_broker* without running the *condor_master* on the broker node, but this requires root access. Generally, this involves adding a line to the `/etc/inittab` file. Some sites use other means to manage and generate the `/etc/inittab`, such as *cfengine* or other system configuration management tools, so check with the local system administrator to be sure. An example line might be something like:

```
GB:23:respawn:/path/to/gcb_broker -i 123.123.123.123 -r /path/to/relay_server
```

It may be easier to wrap the *gcb_broker* binary in a shell script, in order to change the command-line arguments (and set environment variables) without having to edit `/etc/inittab` all the time. This will be similar to:

```
GB:23:respawn:/opt/condor-6.7.13/libexec/gcb_broker.sh
```

Then, create the wrapper, as similar to:

```
#!/bin/sh

libexec=/opt/condor-6.7.13/libexec
ip=123.123.123.123
relay=$libexec/gcb_relay_server

exec $libexec/gcb_broker -i $ip -r $relay
```

You will probably also want to set some environment variables to tell the GCB daemons where to write their log files (`GCB_LOG_DIR`), and possibly some of the other variables described above.

Either way, after updating the `/etc/inittab`, send the *initd* process (always PID 1) a `SIGHUP` signal, and it will re-read the `inittab` and spawn the *gcb_broker*.

Configuring Condor nodes to be GCB clients

In general, before configuring a node in a Condor pool to use GCB, the GCB broker node(s) for the pool must be set up and running. Set up, configure, and spawn the broker first.

To enable the use of GCB on a given Condor host, set the following Condor configuration variables:

```
# Tell Condor to use a network remapping service (currently only GCB
# is supported, but in the future, there might be other options)
NET_REMAP_ENABLE = true
NET_REMAP_SERVICE = GCB
```

Only GCB clients within a private network need to define the following variable, which specifies the IP addresses of the brokers serving this network. Note that these IP addresses must match the IP address that was specified on each broker's command-line with the **-i** option.

```
# Public IP address (in standard dot notation) of the GCB broker(s)
# serving this private node.
NET_REMAP_INAGENT = xxx.xxx.xxx.xxx, yyy.yyy.yyy.yyy
```

When more than one IP address is given, the *condor_master* picks one at random for it and all of its descendants to use. Because the `NET_REMAP_INAGENT` setting is only valid on private nodes, it should not be defined in a global Condor configuration file (`condor_config`) if the pool also contains nodes on a public network.

Finally, if setting up the recommended (but optional) GCB routing table, tell Condor daemons where to find their table. Define the following variable:

```
# The full path to the routing table used by GCB
NET_REMAP_ROUTE = /full/path/to/GCB-routing-table
```

Setting `NET_REMAP_ENABLE` causes the `BIND_ALL_INTERFACES` variable to be automatically set. More information about this setting can be found in section 3.7.2 on page 329. It would not hurt to place the following in the configuration file near the other GCB-related settings, just to remember it:

```
# Tell Condor to bind to all network interfaces, instead of a single
# interface.
BIND_ALL_INTERFACES = true
```

Once a GCB broker is set up and running to manage connections for each private network, and the Condor installation for all the nodes in either private and public networks are configured to enable GCB, restart the Condor daemons, and all of the different machines should be able to communicate with each other.

Configuring the GCB routing table

By default, a GCB-enabled application will always attempt to directly connect to a given IP/port pair. In the case of a private nodes being represented by a GCB broker, the IP/port will be a proxy socket on the broker node, not the real address at each private node. When the GCB broker receives a direct connection to one of its proxy sockets, it notifies the corresponding private node, which establishes a new connection to the broker. The broker then forwards packets between these two sockets, establishing a communication pathway into the private node. This allows clients which are not linked with the GCB libraries to communicate with private nodes using a GCB broker.

This mechanism is expensive in terms of latency (time between messages) and total bandwidth (how much data can be moved in a given time period), as well as expensive in terms of the broker's system resources such as network I/O, processor time, and memory. This expensive mechanism is unnecessary in the case of GCB-aware clients trying to connect to private nodes that can directly communicate with the public host. The alternative is to contact the GCB broker's command interface (the fixed port where the broker is listening for GCB management commands), and use a GCB-specific protocol to request a connection to the given IP/port. In this case, the GCB broker will notify the private node to directly connect to the public client (technically, to a new socket created by the GCB client library linked in with the client's application), and a direct socket between the two is established, removing the need for packet forwarding between the proxy sockets at the GCB broker.

On the other hand, in cases where a direct connection from the client to a given server is possible (for example, two GCB-aware clients in the same public network attempting to communicate with each other), it is expensive and unnecessary to attempt to contact a GCB broker, and the client should connect directly.

To allow a GCB-enabled client to know if it should make a direct connection (which might involve packet forwarding through proxy sockets), or if it should use the GCB protocol to communicate with the broker's command port and arrange a direct socket, GCB provides a *routing table*. Using this table, an administrator can define what IP addresses should be considered private nodes where the GCB connection protocol will be used, and what nodes are public, where a direct connection (without incurring the latency of contacting the GCB broker, only to find out there is no information about the given IP/port) should be made immediately.

If the attempt to contact the GCB broker for a given IP/port fails, or if the desired port is not being managed by the broker, the GCB client library making the connection will fall back and attempt a direct connection. Therefore, configuring a GCB routing table is not required for communication to work within a GCB-enabled environment. However, the GCB routing table can significantly improve performance for communication with private nodes being represented by a GCB broker.

One confusing aspect of GCB is that all of the nodes on a private network believe that their own IP address is the address of their GCB broker. Due to this, all the Condor daemons on a private network advertise themselves with the same IP address (though the broker will map the different ports to different nodes within the private network). Therefore, a given node in the public network needs to be told that if it is contacting this IP address, it should know that the IP address is really a GCB broker representing a node in the private network, so that the public network node can contact

the broker to arrange a single socket from the private node to the public one, instead of relying on forwarding packets between proxy sockets at the broker. Any other addresses, such as other public IP addresses, can be contacted directly, without going through a GCB broker. Similarly, other nodes within the same private network will still be advertising their address with their GCB broker's public IP address. So, nodes within the same private network also have to know that the public IP address of the broker is really a GCB broker, yet all other public IP addresses are valid for direct communication.

In general, all connections can be made directly, except to a host represented by a GCB broker. Furthermore, the default behavior of the GCB client library is to make a direct connection. The routing table is a (somewhat complicated) way to tell a given GCB installation what GCB brokers it might have to communicate with, and that it should directly communicate with anything else. In practice, the routing table should have a single entry for each GCB broker in the system. Future versions of GCB will be able to make use of more complicated routing behavior, which is why the full routing table infrastructure described below is implemented, even if the current version of GCB is not taking advantage of all of it.

Format of the GCB routing table

The routing table is a plain ASCII text file. Each line of the file contains one rule. Each rule consists of a *target* and a *method*. The target specifies destination IP address(es) to match, and the method defines what mechanism must be used to connect to the given target. The target must be a valid IP address string in the standard dotted notation, followed by a slash character (/), as well as an integer *mask*. The mask specifies how many bits of the destination IP address and target IP address must match. The method must be one of the strings

```
GCB
direct
```

GCB stops searching the table as soon as it finds a matching rule, therefore place more specific rules (rules with a larger value for the mask and without wildcards) before generic rules (rules with wildcards or smaller mask values). The default when no rule is matched is to use direct communication. Some examples and the corresponding routing tables may help clarify this syntax.

Simple GCB routing table example (1 private, 1 public)

Consider an example with a private network that has a set of nodes whose IP addresses are 192.168.2.*. Other nodes are in a public network whose IP addresses are 123.123.123.*. A GCB broker for the 192 network is running on IP address 123.123.123.123. In this case, the routing table for both the public and private nodes should be:

```
123.123.123.123/32 GCB
```

This rule states that for IP addresses where all 32 bits exactly match the address 123.123.123.123, first communicate with the GCB broker.

Since the default is to directly connect when no rule in the routing table matches a given target

IP, this single rule is all that is required. However, to illustrate how the routing table syntax works, the following routing table is equivalent:

```
123.123.123.123/32 GCB
*/0 direct
```

Any attempt to connect to 123.123.123.123 uses GCB, as it is the first rule in the file. All other IP addresses will connect directly. This table explicitly defines GCB's default behavior.

More complex GCB routing table example (2 private, 1 public)

As a more complicated case, consider a single Condor pool that spans one public network and two private networks. The two separate private networks each have machines with private addresses like 192.168.2.*. Identify one of these private networks as A, and the other one as B. The public network has nodes with IP addresses like 123.123.123.*. Assume that the GCB broker for nodes in the A network has IP address 123.123.123.65, and the GCB broker for the nodes in the B network has IP address 123.123.123.66. All of the nodes need to be able to talk to each other. In this case, nodes in private network A advertise themselves as 123.123.123.65, so any node, regardless of being in A, B, or the public network, must treat that IP address as a GCB broker. Similarly, nodes in private network B advertise themselves as 123.123.123.66, so any node, regardless of being in A, B, or the public network, must treat that IP address as a GCB broker. All other connections from any node can be made directly. Therefore, here is the appropriate routing table for all nodes:

```
123.123.123.65/32 GCB
123.123.123.66/32 GCB
```

Implications of GCB on Condor's Host/IP-based Security Configuration

When a message is received at a Condor daemon's command socket, Condor authenticates based on the IP address of the incoming socket. For more information about this host-based security in Condor, see section 3.6.9 on page 307. Because of the way GCB changes the IP addresses that are used and advertised by GCB-enabled clients, and since all nodes being represented by a GCB broker are represented by different ports on the broker node (a process known as *address leasing*), using GCB has implications for this process.

Depending on the communication pathway used by a GCB-enabled Condor client (either a tool or another Condor daemon) to connect to a given Condor server daemon, and where in the network each side of the connection resides, the IP address of the resulting socket actually used will be very different. In the case of a private client (that is, a client behind a firewall, which may or may not be using NAT and a fully private, non-routable IP address) attempting to connect to a server, there are three possibilities:

- For a direct connection to another node within the private network, the server will see the private IP address of the client.

- For a direct outbound connection to a public node: if NAT is being used, the server will see the IP address of the NAT server for the private network. If there is no NAT, and the firewall is blocking connections in only one direction, but not re-writing IP addresses, the server will see the client's real IP address.
- For a connection to a host in a different private network that must be relayed through the GCB broker, the server will see the IP address of the GCB broker representing the server. This is an instance of the private server case, as described below.

Therefore, any public server that wants to allow a command from a specific client must have any or all of the various IP addresses mentioned above within the appropriate `HOSTALLOW` settings. In practice, that means opening up the `HOSTALLOW` settings to include not just the actual IP addresses of each node, but also the IP address of the various GCB brokers in use, and potentially, the public IP address of the NAT host for each private network.

However, given that all private nodes which are represented by a given GCB broker could potentially make connections to any other host using the GCB broker's IP address (whenever proxy socket forwarding is being used), if a single private node is being granted a certain level of permission within the Condor pool, all of the private nodes using the same GCB broker will have the same level of permission. This is particularly important in the consideration of granting `HOSTALLOW_ADMINISTRATOR` or `HOSTALLOW_CONFIG` privileges to a private node represented by a GCB broker.

In the case of a public client attempting to connect to a private server, there are only two possible cases:

- the GCB broker can arrange a direct socket from the private server. The private server will see the real public IP address of the client.
- the GCB broker must forward packets from a proxy socket. This may happen because of a non-GCB aware public client, a misconfigured or missing GCB routing table, or a client in a different private network. The private server will see the IP address of its own GCB broker. In the case where the GCB broker runs on a node on the network boundary, the private server will see the GCB broker's private IP address (even if the GCB broker is also listening on the public interface and the leased addresses it provides use the public IP addresses). If the GCB broker is running entirely in the public network and cannot directly connect to the private nodes, the private server will see the remote connection as coming from the broker's public IP address.

This second case is particularly troubling. Since there are legitimate circumstances where a private server would need to use a forwarded proxy socket from its GCB broker, in general, the server should allow requests originating from its GCB broker. But, precisely because of the proxy forwarding, that implies that *any* client that can connect to the GCB broker would be allowed into the private server (if IP-based authorization was the only defense).

The final host-based security setting that requires special mention is `HOSTALLOW_NEGOTIATOR`. If the *condor_negotiator* for the pool is running on a private

node being represented by a GCB broker, there must be modifications to the default value. For the purposes of Condor's host-based security, the *condor_negotiator* acts as a client when communicating with each *condor_schedd* in the pool which has idle jobs that need to be matched with available resources. Therefore, all the possible cases of a private client attempting to connect to a given server apply to a private *condor_negotiator*. In practice, that means adding the public IP address of the broker, the real private IP address of the negotiator host, and possibly the public IP address of the NAT host for this private network to the `HOSTALLOW_NEGOTIATOR` setting. Unfortunately, this implies that *any* host behind the same NAT host or using the same GCB broker will be authorized as if it was the *condor_negotiator*.

Future versions of GCB and Condor will hopefully add some form of authentication and authorization to the GCB broker itself, to help alleviate these problems. Until then, sites using GCB are encouraged to use GSI strong authentication (since Kerberos also depends on IP addresses and is therefore incompatible with GCB) to rely on an authorization system that is not affected by address leasing. This is especially true for sites that (foolishly) choose to run their central manager on a private node.

Implications of GCB for Other Condor Configuration

Using GCB and address leasing has implications for Condor configuration settings outside of the Host/IP-based security settings. Each is described.

COLLECTORHOST If the *condor_collector* for the pool is running on a private node being represented by a GCB broker, `COLLECTOR_HOST` must be set to the host name or IP address of the GCB broker machine, *not* the real host name/IP address of the private node where the daemons are actually running. When the *condor_collector* on the private node attempts to `bind()` to its command port (9618 by default), it will request port 9618 on the GCB broker node, instead. The port is not a worry, but the host name or IP address is a worry. When public nodes want to communicate with the *condor_collector*, they must go through the GCB broker. In theory, other nodes inside the same private network could be told to directly use the private IP address of the *condor_collector* host, but that is unnecessary, and would probably lead to other confusion and configuration problems.

However, because the *condor_collector* is listening on a fixed port, and that single port is reserved on the GCB broker node, no two private nodes using the same broker can attempt to use the same port for their *condor_collector*. Therefore, any site that is attempting to set up multiple pools within the same private network is strongly encouraged to set up separate GCB brokers for each pool. Otherwise, one or both of the pools must use a non-standard port for the *condor_collector*, which adds yet more complication to an already complicated situation.

CKPT_SERVER_HOST Much like the case for `COLLECTOR_HOST` described above, a checkpoint server on a private node will have to lease a port on the GCB broker node. However, the checkpoint server also uses a fixed port, and unlike the *condor_collector*, there is no way to configure an alternate value. Therefore, only a single checkpoint server can be run behind a given GCB broker. The same solution works: if multiple checkpoint servers are required, multiple GCB brokers are deployed and configured. Furthermore, the host name of the GCB

broker should be used as the value for `CKPT_SERVER_HOST`, not the real IP address or host name of the private node where the *condor_ckpt_server* is running.

SEC_DEFAULT_AUTHENTICATION_METHODS KERBEROS may not be used for authentication on a GCB-enabled pool. The IP addresses used in various circumstances will not be the real IP addresses of the machines. Since Kerberos stores the IP address of each host as part of the Kerberos ticket, authentication will fail on a GCB-enabled pool.

Due to the complications and security limitations that arise from running a central manager on a private node represented by GCB (both regarding the `COLLECTOR_HOST` and `HOSTALLOW_NEGOTIATOR`), we recommend that sites avoid locating a central manager on a private host whenever possible.

3.7.4 Using TCP to Send Updates to the *condor_collector*

TCP sockets are reliable, connection-based sockets that guarantee the delivery of any data sent. However, TCP sockets are fairly expensive to establish, and there is more network overhead involved in sending and receiving messages.

UDP sockets are datagrams, and are not reliable. There is very little overhead in establishing or using a UDP socket, but there is also no guarantee that the data will be delivered. All previous Condor versions used UDP sockets to send updates to the *condor_collector*, and this did not cause problems.

Condor can be configured to use TCP sockets to send updates to the *condor_collector* instead of UDP datagrams. It is *not* intended for most sites. This feature is targeted at sites where UDP updates are lost because of the underlying network. Most Condor administrators that believe this is a good idea for their site are wrong. Do not enable this feature just because it sounds like a good idea. The only cases where an administrator would want this feature are if the ClassAd updates are consistently not getting to the *condor_collector*. An example where this may happen is if the pool is comprised of machines across a wide area network (WAN) where UDP packets are frequently dropped.

Configuration variables are set to enable the use of TCP sockets. There are two variables that an administrator must define to enable this feature:

UPDATE_COLLECTOR_WITH_TCP When set to `True`, the Condor daemons to use TCP to update the *condor_collector*, instead of the default UDP. Defaults to `False`.

COLLECTOR_SOCKET_CACHE_SIZE Specifies the number of TCP sockets cached at the *condor_collector*. The default value for this setting is 0, with no cache enabled.

The use of a cache allows Condor to leave established TCP sockets open, facilitating much better performance. Subsequent updates can reuse an already open socket. The work to establish a TCP connection may be lengthy, including authentication and setting up encryption. Therefore, Condor

requires that a socket cache be defined if TCP updates are to be used. TCP updates will be refused by the *condor_collector* daemon if a cache is not enabled.

Each Condor daemon will have 1 socket open to the *condor_collector*. So, in a pool with N machines, each of them running a *condor_master*, *condor_schedd*, and *condor_startd*, the *condor_collector* would need a socket cache that has at least 3*N entries. Machines running Personal Condor in the pool need an additional two entries (for the *condor_master* and *condor_schedd*) for each Personal Condor installation.

Every cache entry utilizes a file descriptor within the *condor_collector* daemon. Therefore, be careful not to define a cache that is larger than the number of file descriptors the underlying operating system allocates for a single process.

NOTE: At this time, `UPDATE_COLLECTOR_WITH_TCP`, only affects the main *condor_collector* for the site, not any sites that a *condor_schedd* might flock to.

3.8 The Checkpoint Server

A Checkpoint Server maintains a repository for checkpoint files. Using checkpoint servers reduces the disk requirements of submitting machines in the pool, since the submitting machines no longer need to store checkpoint files locally. Checkpoint server machines should have a large amount of disk space available, and they should have a fast connection to machines in the Condor pool.

If your spool directories are on a network file system, then checkpoint files will make two trips over the network: one between the submitting machine and the execution machine, and a second between the submitting machine and the network file server. If you install a checkpoint server and configure it to use the server's local disk, the checkpoint will travel only once over the network, between the execution machine and the checkpoint server. You may also obtain checkpointing network performance benefits by using multiple checkpoint servers, as discussed below.

NOTE: It is a good idea to pick very stable machines for your checkpoint servers. If individual checkpoint servers crash, the Condor system will continue to operate, although poorly. While the Condor system will recover from a checkpoint server crash as best it can, there are two problems that can (and will) occur:

1. A checkpoint cannot be sent to a checkpoint server that is not functioning. Jobs will keep trying to contact the checkpoint server, backing off exponentially in the time they wait between attempts. Normally, jobs only have a limited time to checkpoint before they are kicked off the machine. So, if the server is down for a long period of time, chances are that a lot of work will be lost by jobs being killed without writing a checkpoint.
2. If a checkpoint is not available from the checkpoint server, a job cannot be retrieved, and it will either have to be restarted from the beginning, or the job will wait for the server to come back online. This behavior is controlled with the `MAX_DISCARDED_RUN_TIME` parameter in the config file (see section 3.3.8 on page 172 for details). This parameter represents the

maximum amount of CPU time you are willing to discard by starting a job over from scratch if the checkpoint server is not responding to requests.

3.8.1 Preparing to Install a Checkpoint Server

The location of checkpoints changes upon the installation of a checkpoint server. A configuration change would cause currently queued jobs with checkpoints to not be able to find their checkpoints. This results in the jobs with checkpoints remaining indefinitely queued (never running) due to the lack of finding their checkpoints. It is therefore best to either remove jobs from the queues or let them complete before installing a checkpoint server. It is advisable to shut your pool down before doing any maintenance on your checkpoint server. See section ?? on page ?? for details on shutting down your pool.

A graduated installation of the checkpoint server may be accomplished by configuring submit machines as their queues empty.

3.8.2 Installing the Checkpoint Server Module

Files relevant to a checkpoint server are

```
sbin/condor_ckpt_server
sbin/condor_cleanckpts
etc/examples/condor_config.local.ckpt.server
```

`condor_ckpt_server` is the checkpoint server binary. `condor_cleanckpts` is a script that can be periodically run to remove stale checkpoint files from your server. The checkpoint server normally cleans all old files itself. However, in certain error situations, stale files can be left that are no longer needed. You may set up a cron job that calls `condor_cleanckpts` every week or so to automate the cleaning up of any stale files. The example configuration file give with the module is described below.

There are three steps necessary towards running a checkpoint server:

1. Configure the checkpoint server.
2. Start the checkpoint server.
3. Configure your pool to use the checkpoint server.

Configure the Checkpoint Server Place settings in the local configuration file of the checkpoint server. The file `etc/examples/condor_config.local.ckpt.server` contains the needed settings. Insert these into the local configuration file of your checkpoint server machine.

The `CKPT_SERVER_DIR` must be customized. The `CKPT_SERVER_DIR` attribute defines where your checkpoint files are to be located. It is better if this is on a very fast local file system (preferably a RAID). The speed of this file system will have a direct impact on the speed at which your checkpoint files can be retrieved from the remote machines.

The other optional settings are:

DAEMONLIST (Described in section 3.3.9). To have the checkpoint server managed by the *condor_master*, the `DAEMONLIST` entry must have `MASTER` and `CKPT_SERVER`. Add `STARTD` if you want to allow jobs to run on your checkpoint server. Similarly, add `SCHEDD` if you would like to submit jobs from your checkpoint server.

The rest of these settings are the checkpoint server-specific versions of the Condor logging entries, as described in section 3.3.4 on page 157.

CKPT_SERVER_LOG The `CKPT_SERVER_LOG` is where the checkpoint server log is placed.

MAX_CKPT_SERVER_LOG Sets the maximum size of the checkpoint server log before it is saved and the log file restarted.

CKPT_SERVER_DEBUG Regulates the amount of information printed in the log file. Currently, the only debug level supported is `D.ALWAYS`.

Start the Checkpoint Server To start the newly configured checkpoint server, restart Condor on that host to enable the *condor_master* to notice the new configuration. Do this by sending a *condor_restart* command from any machine with administrator access to your pool. See section 3.6.9 on page 307 for full details about IP/host-based security in Condor.

Configure the Pool to Use the Checkpoint Server After the checkpoint server is running, you change a few settings in your configuration files to let your pool know about your new server:

USE_CKPT_SERVER This parameter should be set to `TRUE` (the default).

CKPT_SERVER_HOST This parameter should be set to the full host name of the machine that is now running your checkpoint server.

It is most convenient to set these parameters in your global configuration file, so they affect all submission machines. However, you may configure each submission machine separately (using local configuration files) if you do not want all of your submission machines to start using the checkpoint server at one time. If `USE_CKPT_SERVER` is set to `FALSE`, the submission machine will not use a checkpoint server.

Once these settings are in place, send a *condor_reconfig* to all machines in your pool so the changes take effect. This is described in section ?? on page ??.

3.8.3 Configuring your Pool to Use Multiple Checkpoint Servers

It is possible to configure a Condor pool to use multiple checkpoint servers. The deployment of checkpoint servers across the network improves checkpointing performance. In this case, Condor machines are configured to checkpoint to the *nearest* checkpoint server. There are two main performance benefits to deploying multiple checkpoint servers:

- Checkpoint-related network traffic is localized by intelligent placement of checkpoint servers.
- Faster checkpointing implies that jobs spend less time checkpointing, more time doing useful work, jobs have a better chance of checkpointing successfully before returning a machine to its owner, and workstation owners see Condor jobs leave their machines quicker.

Once you have multiple checkpoint servers running in your pool, the following configuration changes are required to make them active.

First, `USE_CKPT_SERVER` should be set to `TRUE` (the default) on all submitting machines where Condor jobs should use a checkpoint server. Additionally, `STARTER_CHOOSES_CKPT_SERVER` should be set to `TRUE` (the default) on these submitting machines. When `TRUE`, this parameter specifies that the checkpoint server specified by the machine running the job should be used instead of the checkpoint server specified by the submitting machine. See section 3.3.8 on page 172 for more details. This allows the job to use the checkpoint server closest to the machine on which it is running, instead of the server closest to the submitting machine. For convenience, set these parameters in the global configuration file.

Second, set `CKPT_SERVER_HOST` on each machine. As described, this is set to the full host name of the checkpoint server machine. In the case of multiple checkpoint servers, set this in the local configuraton file. It is the host name of the nearest server to the machine.

Third, send a *condor_reconfig* to all machines in the pool so the changes take effect. This is described in section ?? on page ??.

After completing these three steps, the jobs in your pool will send checkpoints to the nearest checkpoint server. On restart, a job will remember where its checkpoint was stored and get it from the appropriate server. After a job successfully writes a checkpoint to a new server, it will remove any previous checkpoints left on other servers.

NOTE: If the configured checkpoint server is unavailable, the job will keep trying to contact that server as described above. It will not use alternate checkpoint servers. This may change in future versions of Condor.

3.8.4 Checkpoint Server Domains

The configuration described in the previous section ensures that jobs will always write checkpoints to their nearest checkpoint server. In some circumstances, it is also useful to configure Condor to localize checkpoint read transfers, which occur when the job restarts from its last checkpoint on a new machine. To localize these transfers, we want to schedule the job on a machine which is near the checkpoint server on which the job's checkpoint is stored.

We can say that all of the machines configured to use checkpoint server "A" are in "checkpoint server domain A." To localize checkpoint transfers, we want jobs which run on machines in a given checkpoint server domain to continue running on machines in that domain, transferring checkpoint files in a single local area of the network. There are two possible configurations which specify what a job should do when there are no available machines in its checkpoint server domain:

- The job can remain idle until a workstation in its checkpoint server domain becomes available.
- The job can try to immediately begin executing on a machine in another checkpoint server domain. In this case, the job transfers to a new checkpoint server domain.

These two configurations are described below.

The first step in implementing checkpoint server domains is to include the name of the nearest checkpoint server in the machine ClassAd, so this information can be used in job scheduling decisions. To do this, add the following configuration to each machine:

```
CkptServer = "$(CKPT_SERVER_HOST) "
STARTD_ATTRS = $(STARTD_ATTRS), CkptServer
```

For convenience, we suggest that you set these parameters in the global config file. Note that this example assumes that `STARTD_ATTRS` is defined previously in your configuration. If not, then you should use the following configuration instead:

```
CkptServer = "$(CKPT_SERVER_HOST) "
STARTD_ATTRS = CkptServer
```

Now, all machine ClassAds will include a `CkptServer` attribute, which is the name of the checkpoint server closest to this machine. So, the `CkptServer` attribute defines the checkpoint server domain of each machine.

To restrict jobs to one checkpoint server domain, we need to modify the jobs' `Requirements` expression as follows:

```
Requirements = ((LastCkptServer == TARGET.CkptServer) || (LastCkptServer == UNDEFINED))
```

This `Requirements` expression uses the `LastCkptServer` attribute in the job's ClassAd, which specifies where the job last wrote a checkpoint, and the `CkptServer` attribute in the machine ClassAd, which specifies the checkpoint server domain. If the job has not written a checkpoint yet, the `LastCkptServer` attribute will be `UNDEFINED`, and the job will be able to execute in any checkpoint server domain. However, once the job performs a checkpoint, `LastCkptServer` will be defined and the job will be restricted to the checkpoint server domain where it started running.

If instead we want to allow jobs to transfer to other checkpoint server domains when there are no available machines in the current checkpoint server domain, we need to modify the jobs' `Rank` expression as follows:

```
Rank = ((LastCkptServer == TARGET.CkptServer) || (LastCkptServer == UNDEFINED))
```

This `Rank` expression will evaluate to 1 for machines in the job's checkpoint server domain and 0 for other machines. So, the job will prefer to run on machines in its checkpoint server domain, but if no such machines are available, the job will run in a new checkpoint server domain.

You can automatically append the checkpoint server domain `Requirements` or `Rank` expressions to all `STANDARD` universe jobs submitted in your pool using `APPEND_REQ_STANDARD` or `APPEND_RANK_STANDARD`. See section 3.3.14 on page 205 for more details.

3.9 DaemonCore

This section is a brief description of *DaemonCore*. *DaemonCore* is a library that is shared among most of the Condor daemons which provides common functionality. Currently, the following daemons use *DaemonCore*:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*
- *condor_quill*
- *condor_dbmsd*
- *condor_gridmanager*
- *condor_credd*
- *condor_had*
- *condor_replication*
- *condor_job_router*
- *condor_lease_manager*

Most of *DaemonCore*'s details are not interesting for administrators. However, *DaemonCore* does provide a uniform interface for the daemons to various Unix signals, and provides a common set of command-line options that can be used to start up each daemon.

3.9.1 DaemonCore and Unix signals

One of the most visible features that DaemonCore provides for administrators is that all daemons which use it behave the same way on certain Unix signals. The signals and the behavior DaemonCore provides are listed below:

SIGHUP Causes the daemon to reconfigure itself.

SIGTERM Causes the daemon to gracefully shutdown.

SIGQUIT Causes the daemon to quickly shutdown.

Exactly what gracefully and quickly means varies from daemon to daemon. For daemons with little or no state (the *condor_kbdd*, *condor_collector* and *condor_negotiator*) there is no difference, and both SIGTERM and SIGQUIT signals result in the daemon shutting itself down quickly. For the *condor_master*, a graceful shutdown causes the *condor_master* to ask all of its children to perform their own graceful shutdown methods. The quick shutdown causes the *condor_master* to ask all of its children to perform their own quick shutdown methods. In both cases, the *condor_master* exits after all its children have exited. In the *condor_startd*, if the machine is not claimed and running a job, both the SIGTERM and SIGQUIT signals result in an immediate exit. However, if the *condor_startd* is running a job, a graceful shutdown results in that job writing a checkpoint, while a fast shutdown does not. In the *condor_schedd*, if there are no jobs currently running, there will be no *condor_shadow* processes, and both signals result in an immediate exit. However, with jobs running, a graceful shutdown causes the *condor_schedd* to ask each *condor_shadow* to gracefully vacate the job it is serving, while a quick shutdown results in a hard kill of every *condor_shadow*, with no chance to write a checkpoint.

For all daemons, a reconfigure results in the daemon re-reading its configuration file(s), causing any settings that have changed to take effect. See section 3.3 on page 142, Configuring Condor for full details on what settings are in the configuration files and what they do.

3.9.2 DaemonCore and Command-line Arguments

The second visible feature that DaemonCore provides to administrators is a common set of command-line arguments that all daemons understand. These arguments and what they do are described below:

- a string** Append a period character (' . ') concatenated with **string** to the file name of the log for this daemon, as specified in the configuration file.
- b** Causes the daemon to start up in the background. When a DaemonCore process starts up with this option, it disassociates itself from the terminal and forks itself, so that it runs in the background. This is the default behavior for Condor daemons.

-c filename Causes the daemon to use the specified **filename** as a full path and file name as its global configuration file. This overrides the `CONDOR_CONFIG` environment variable and the regular locations that Condor checks for its configuration file which are the `condor` user's home directory and the file `/etc/condor/condor_config`.

-d Use dynamic directories. The `$(LOG)`, `$(SPOOL)`, and `$(EXECUTE)` directories are all created by the daemon at run time, and they are named by appending the parent's IP address and PID to the value in the configuration file. These values are then inherited by all children of the daemon invoked with this **-d** argument. For the *condor_master*, all Condor processes will use the new directories. If a *condor_schedd* is invoked with the **-d** argument, then only the *condor_schedd* daemon and any *condor_shadow* daemons it spawns will use the dynamic directories (named with the *condor_schedd* daemon's PID).

Note that by using a dynamically-created spool directory named by the IP address and PID, upon restarting daemons, jobs submitted to the original *condor_schedd* daemon that were stored in the old spool directory will not be noticed by the new *condor_schedd* daemon, unless you manually specify the old, dynamically-generated `SPOOL` directory path in the configuration of the new *condor_schedd* daemon.

-f Causes the daemon to start up in the foreground. Instead of forking, the daemon runs in the foreground.

NOTE: When the *condor_master* starts up daemons, it does so with the **-f** option, as it has already forked a process for the new daemon. There will be a **-f** in the argument list for all Condor daemons that the *condor_master* spawns.

-k filename For non-Windows operating systems, causes the daemon to read out a PID from the specified **filename**, and send a `SIGTERM` to that process. The daemon started with this optional argument waits until the daemon it is attempting to kill has exited.

-l directory Overrides the value of `LOG` as specified in the configuration files. Primarily, this option is used with the *condor_kbdd* when it needs to run as the individual user logged into the machine, instead of running as root. Regular users would not normally have permission to write files into Condor's log directory. Using this option, they can override the value of `LOG` and have the *condor_kbdd* write its log file into a directory that the user has permission to write to.

-local-name name Specify a local name for this instance of the daemon. This local name will be used to look up configuration parameters. Section 3.3.1 contains details on how this local name will be used in the configuration.

-p port Causes the daemon to bind to the specified port as its command socket. The *condor_master* daemon uses this option to ensure that the *condor_collector* and *condor_negotiator* start up using well-known ports that the rest of Condor depends upon them using.

-pidfile filename Causes the daemon to write out its PID (process id number) to the specified **filename**. This file can be used to help shutdown the daemon without first searching through the output of the Unix *ps* command.

Since daemons run with their current working directory set to the value of `LOG`, if you don't specify a full path (one that begins with a `/`), the file will be placed in the `LOG` directory.

- q** Quiet output; write less verbose error messages to `stderr` when something goes wrong, and before regular logging can be initialized.
- r minutes** Causes the daemon to set a timer, upon expiration of which, it sends itself a `SIGTERM` for graceful shutdown.
- t** Causes the daemon to print out its error message to `stderr` instead of its specified log file. This option forces the **-f** option.
- v** Causes the daemon to print out version information and exit.

3.10 The High Availability of Daemons

In the case that a key machine no longer functions, Condor can be configured such that another machine takes on the key functions. This is called *High Availability*. While high availability is generally applicable, there are currently two specialized cases for its use: when the central manager (running the *condor_negotiator* and *condor_collector* daemons) becomes unavailable, and when the machine running the *condor_schedd* daemon (maintaining the job queue) becomes unavailable.

3.10.1 High Availability of the Job Queue

For a pool where all jobs are submitted through a single machine in the pool, and there are lots of jobs, this machine becoming nonfunctional means that jobs stop running. The *condor_schedd* daemon maintains the job queue. No job queue due to having a nonfunctional machine implies that no jobs can be run. This situation is worsened by using one machine as the single submission point. For each Condor job (taken from the queue) that is executed, a *condor_shadow* process runs on the machine where submitted to handle input/output functionality. If this machine becomes nonfunctional, none of the jobs can continue. The entire pool stops running jobs.

The goal of *High Availability* in this special case is to transfer the *condor_schedd* daemon to run on another designated machine. Jobs caused to stop without finishing can be restarted from the beginning, or can continue execution using the most recent checkpoint. New jobs can enter the job queue. Without *High Availability*, the job queue would remain intact, but further progress on jobs would wait until the machine running the *condor_schedd* daemon became available (after fixing whatever caused it to become unavailable).

Condor uses its flexible configuration mechanisms to allow the transfer of the *condor_schedd* daemon from one machine to another. The configuration specifies which machines are chosen to run the *condor_schedd* daemon. To prevent multiple *condor_schedd* daemons from running at the same time, a lock (semaphore-like) is held over the job queue. This synchronizes the situation in which control is transferred to a secondary machine, and the primary machine returns to functionality. Configuration variables also determine time intervals at which the lock expires, and periods of time that pass between polling to check for expired locks.

To specify a single machine that would take over, if the machine running the *condor_schedd* daemon stops working, the following additions are made to the local configuration of any and all machines that are able to run the *condor_schedd* daemon (becoming the single pool submission point):

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
VALID_SPOOL_FILES = SCHEDD.lock
```

Configuration macro `MASTER_HA_LIST` identifies the *condor_schedd* daemon as the daemon that is to be watched to make sure that it is running. Each machine with this configuration must have access to the lock (the job queue) which synchronizes which single machine does run the *condor_schedd* daemon. This lock and the job queue must both be located in a shared file space, and is currently specified only with a file URL. The configuration specifies the shared space (`SPOOL`), and the URL of the lock. *condor_preen* is not currently aware of the lock file and will delete it if it is placed in the `SPOOL` directory, so be sure to add `SCHEDD.lock` to `VALID_SPOOL_FILES`.

As Condor starts on machines that are configured to run the single *condor_schedd* daemon, the *condor_master* daemon of the first machine that looks at (polls) the lock and notices that no lock is held. This implies that no *condor_schedd* daemon is running. This *condor_master* daemon acquires the lock and runs the *condor_schedd* daemon. Other machines with this same capability to run the *condor_schedd* daemon look at (poll) the lock, but do not run the daemon, as the lock is held. The machine running the *condor_schedd* daemon renews the lock periodically.

If the machine running the *condor_schedd* daemon fails to renew the lock (because the machine is not functioning), the lock times out (becomes stale). The lock is released by the *condor_master* daemon if *condor_off* or *condor_off-schedd* is executed, or when the *condor_master* daemon knows that the *condor_schedd* daemon is no longer running. As other machines capable of running the *condor_schedd* daemon look at the lock (poll), one machine will be the first to notice that the lock has timed out or been released. This machine (correctly) interprets this situation as the *condor_schedd* daemon is no longer running. This machine's *condor_master* daemon then acquires the lock and runs the *condor_schedd* daemon.

See section 3.3.9, in the section on *condor_master* Configuration File Macros for details relating to the configuration variables used to set timing and polling intervals.

Working with Remote Job Submission

Remote job submission requires identification of the job queue, submitting with a command similar to:

```
% condor_submit -remote condor@example.com myjob.submit
```

This implies the identification of a single *condor_schedd* daemon, running on a single machine. With the high availability of the job queue, there are multiple *condor_schedd* daemons, of which

only one at a time is acting as the single submission point. To make remote submission of jobs work properly, set the configuration variable `SCHEDD_NAME` in the local configuration to have the same value for each potentially running *condor_schedd* daemon. In addition, the value chosen for the variable `SCHEDD_NAME` will need to include the at symbol (@), such that Condor will not modify the value set for this variable. See the description of `MASTER_NAME` in section 3.3.9 on page 177 for defaults and composition of valid values for `SCHEDD_NAME`. As an example, include in each local configuration a value similar to:

```
SCHEDD_NAME = had-schedd@
```

Then, with this sample configuration, the submit command appears as:

```
% condor_submit -remote had-schedd@ myjob.submit
```

3.10.2 High Availability of the Central Manager

Interaction with Flocking

The Condor high availability mechanisms discussed in this section currently do not work well in configurations involving flocking. The individual problems listed interact to make the situation worse. Because of these problems, we advise against the use of flocking to pools with high availability mechanisms enabled.

- The *condor_schedd* has a hard configured list of *condor_collector* and *condor_negotiator* daemons, and does not query redundant collectors to get the current *condor_negotiator*, as it does when communicating with its local pool. As a result, if the default *condor_negotiator* fails, the *condor_schedd* does not learn of the failure, and thus, talk to the new *condor_negotiator*.
- When the *condor_negotiator* is unable to communicate with a *condor_collector*, it utilizes the next *condor_collector* within the list. Unfortunately, it does not start over at the top of the list. When combined with the previous problem, a backup *condor_negotiator* will never get jobs from a flocked *condor_schedd*.

Introduction

The *condor_negotiator* and *condor_collector* daemons are the heart of the Condor matchmaking system. The availability of these daemons is critical to a Condor pool's functionality. Both daemons usually run on the same machine, most often known as the central manager. The failure of a central manager machine prevents Condor from matching new jobs and allocating new resources. High availability of the *condor_negotiator* and *condor_collector* daemons eliminates this problem.

Configuration allows one of multiple machines within the pool to function as the central manager. While there may be many active *condor_collector* daemons, only a single, active *condor_negotiator* daemon will be running. The machine with the *condor_negotiator* daemon running

is the active central manager. The other potential central managers each have a *condor_collector* daemon running; these are the idle central managers.

All submit and execute machines are configured to report to all potential central manager machines.

Each potential central manager machine runs the high availability daemon, *condor_had*. These daemons communicate with each other, constantly monitoring the pool to ensure that one active central manager is available. If the active central manager machine crashes or is shut down, these daemons detect the failure, and they agree on which of the idle central managers is to become the active one. A protocol determines this.

In the case of a network partition, idle *condor_had* daemons within each partition detect (by the lack of communication) a partitioning, and then use the protocol to choose an active central manager. As long as the partition remains, and there exists an idle central manager within the partition, there will be one active central manager within each partition. When the network is repaired, the protocol returns to having one central manager.

Through configuration, a specific central manager machine may act as the primary central manager. While this machine is up and running, it functions as the central manager. After a failure of this primary central manager, another idle central manager becomes the active one. When the primary recovers, it again becomes the central manager. This is a recommended configuration, if one of the central managers is a reliable machine, which is expected to have very short periods of instability. An alternative configuration allows the promoted active central manager (in the case that the central manager fails) to stay active after the failed central manager machine returns.

This high availability mechanism operates by monitoring communication between machines. Note that there is a significant difference in communications between machines when

1. a machine is down
2. a specific daemon (the *condor_had* daemon in this case) is not running, yet the machine is functioning

The high availability mechanism distinguishes between these two, and it operates based only on first (when a central manager machine is down). A lack of executing daemons does *not* cause the protocol to choose or use a new active central manager.

The central manager machine contains state information, and this includes information about user priorities. The information is kept in a single file, and is used by the central manager machine. Should the primary central manager fail, a pool with high availability enabled would lose this information (and continue operation, but with re-initialized priorities). Therefore, the *condor_replication* daemon exists to replicate this file on all potential central manager machines. This daemon promulgates the file in a way that is safe from error, and more secure than dependence on a shared file system copy.

The *condor_replication* daemon runs on each potential central manager machine as well as on the active central manager machine. There is a unidirectional communication between the *condor_had* daemon and the *condor_replication* daemon on each machine.

Configuration

The high availability of central manager machines is enabled through configuration. It is disabled by default. All machines in a pool must be configured appropriately in order to make the high availability mechanism work. See section 3.3.29, for definitions of these configuration variables.

The stabilization period is the time it takes for the *condor_had* daemons to detect a change in the pool state such as an active central manager failure or network partition, and recover from this change. It may be computed using the following formula:

$$\text{stabilization period} = 12 * (\text{number of central managers}) * \\ \$(\text{HAD_CONNECTION_TIMEOUT})$$

To disable the high availability of central managers mechanism, it is sufficient to remove HAD, REPLICATION, and NEGOTIATOR from the DAEMON_LIST configuration variable on all machines, leaving only one *condor_negotiator* in the pool.

To shut down a currently operating high availability mechanism, follow the given steps. All commands must be invoked from a host which has administrative permissions on all central managers. The first three commands kill all *condor_had*, *condor_replication*, and all running *condor_negotiator* daemons. The last command is invoked on the host where the single *condor_negotiator* daemon is to run.

1. `condor_off -all -neg`
2. `condor_off -all -subsystem -replication`
3. `condor_off -all -subsystem -had`
4. `condor_on -neg`

When configuring *condor_had* to control the *condor_negotiator*, if the default backoff constant value is too small, it can result in a churning of the *condor_negotiator*, especially in cases in which the primary negotiator is unable to run due to misconfiguration. In these cases, the *condor_master* will kill the *condor_had* after the *condor_negotiator* exists, wait a short period, then restart *condor_had*. The *condor_had* will then win the election, so the secondary *condor_negotiator* will be killed, and the primary will be restarted, only to exit again. If this happens to quickly, neither *condor_negotiator* will run long enough to complete a negotiation cycle, resulting in no jobs getting started. Increasing this value via MASTER_HAD_BACKOFF_CONSTANT to be larger than a typical negotiation cycle can help solve this problem.

To run a high availability pool without the replication feature, do the following operations:

1. Set the HAD_USE_REPLICATION configuration variable to `false`, and thus disable the replication on configuration level.
2. Remove REPLICATION from both DAEMON_LIST and DC_DAEMON_LIST in the configuration file.

Sample Configuration

This section provides sample configurations for high availability. The two parts to this are the configuration for the potential central manager machines, and the configuration for the machines within the pool that will *not* be central managers.

This is a sample configuration relating to the high availability of central managers. This is for the potential central manager machines.

```
#####
# A sample configuration file for central managers, to enable the      #
# the high availability mechanism.                                     #
#####

# unset these two macros
NEGOTIATOR_HOST=
CONDOR_HOST=

#####
## THE FOLLOWING MUST BE IDENTICAL ON ALL POTENTIAL CENTRAL MANAGERS.  #
#####
## For simplicity in writing other expressions, define a variable
## for each potential central manager in the pool.
## These are samples.
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
## A list of all potential central managers in the pool.
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

## Define the port number on which the condor_had daemon will
## listen. The port must match the port number used
## for when defining HAD_LIST. This port number is
## arbitrary; make sure that there is no port number collision
## with other applications.
HAD_PORT = 51450
HAD_ARGS = -p $(HAD_PORT)

## The following macro defines the port number condor_replication will listen
## on on this machine. This port should match the port number specified
## for that replication daemon in the REPLICATION_LIST
## Port number is arbitrary (make sure no collision with other applications)
## This is a sample port number
REPLICATION_PORT = 41450
REPLICATION_ARGS = -p $(REPLICATION_PORT)

## The following list must contain the same addresses
## as HAD_LIST. In addition, for each hostname, it should specify
## the port number of condor_replication daemon running on that host.
## This parameter is mandatory and has no default value
REPLICATION_LIST = \
$(CENTRAL_MANAGER1):$(REPLICATION_PORT), \
$(CENTRAL_MANAGER2):$(REPLICATION_PORT)

## The following list must contain the same addresses in the same order
## as COLLECTOR_HOST. In addition, for each hostname, it should specify
## the port number of condor_had daemon running on that host.
```

```

## The first machine in the list will be the PRIMARY central manager
## machine, in case HAD_USE_PRIMARY is set to true.
HAD_LIST = \
$(CENTRAL_MANAGER1):$(HAD_PORT), \
$(CENTRAL_MANAGER2):$(HAD_PORT)

## HAD connection time.
## Recommended value is 2 if the central managers are on the same subnet.
## Recommended value is 5 if Condor security is enabled.
## Recommended value is 10 if the network is very slow, or
## to reduce the sensitivity of HA daemons to network failures.
HAD_CONNECTION_TIMEOUT = 2

##If true, the first central manager in HAD_LIST is a primary.
HAD_USE_PRIMARY = true

##-----
## Host/IP access levels
##-----

## What machines have administrative rights for your pool? This
## defaults to your central manager. You should set it to the
## machine(s) where whoever is the condor administrator(s) works
## (assuming you trust all the users who log into that/those
## machine(s), since this is machine-wide access you're granting).
HOSTALLOW_ADMINISTRATOR = $(COLLECTOR_HOST)

## Negotiator access. Machines listed here are trusted central
## managers. You should normally not have to change this.
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)

#####
## THE PARAMETERS BELOW ARE ALLOWED TO BE DIFFERENT ON EACH #
## CENTRAL MANAGERS #
## THESE ARE MASTER SPECIFIC PARAMETERS
#####

## The location of executable files
HAD = $(SBIN)/condor_had
REPLICATION = $(SBIN)/condor_replication
TRANSFERER = $(SBIN)/condor_transferer

## the master should start at least these five daemons
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
## DC_Daemon list should contain at least these five
DC_DAEMON_LIST = +HAD, REPLICATION

## Enables/disables the replication feature of HAD daemon
## Default: no
HAD_USE_REPLICATION = true

## Name of the file from the SPOOL directory that will be replicated
## Default: $(SPOOL)/Accountantnew.log
STATE_FILE = $(SPOOL)/Accountantnew.log

```

```

## Period of time between two successive awakenings of the replication daemon
## Default: 300
REPLICATION_INTERVAL = 300

# Period of time, in which transferer daemons have to accomplish the
## downloading/uploading process
## Default: 300
MAX_TRANSFERER_LIFETIME = 300

## Period of time between two successive sends of ClassAds to the collector by HAD
## Default: 300
HAD_UPDATE_INTERVAL = 300

## The HAD controls the negotiator, and should have a larger
## backoff constant
MASTER_NEGOTIATOR_CONTROLLER = HAD
MASTER_HAD_BACKOFF_CONSTANT = 360

## The size of the log file
MAX_HAD_LOG = 640000
## debug level
HAD_DEBUG = D_COMMAND
## location of the condor_had log file
HAD_LOG = $(LOG)/HADLog

## The size of replication log file
MAX_REPLICATION_LOG = 640000
## Replication debug level
REPLICATION_DEBUG = D_COMMAND
## Replication log file
REPLICATION_LOG = $(LOG)/ReplicationLog

## The size of transferer log file
MAX_TRANSFERER_LOG = 640000
## Replication debug level
TRANSFERER_DEBUG = D_COMMAND
## Replication log file
TRANSFERER_LOG = $(LOG)/TransferLog

```

Machines that are not potential central managers also require configuration. The following is a sample configuration relating to high availability for machines that will *not* be central managers.

```

#####
# Sample configuration relating to high availability for machines      #
# that DO NOT run the condor_had daemon.                             #
#####

#unset these variables
NEGOTIATOR_HOST =
CONDOR_HOST =

## For simplicity define a variable for each potential central manager
## in the pool.
CENTRAL_MANAGER1 = cml.cs.technion.ac.il

```

```

CENTRAL_MANAGER2 = cm2.cs.technion.ac.il
## List of all potential central managers in the pool
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

##-----
## Host/IP access levels
##-----

## Negotiator access. Machines listed here are trusted central
## managers. You should normally not need to change this.
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)

## Now, with flocking (and HA) we need to let the SCHEDD trust the other
## negotiators we are flocking with as well. You should normally
## not need to change this.
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST)

```

3.11 Quill

Quill is an optional component of Condor that maintains a mirror of Condor operational data in a relational database. The *condor-quill* daemon updates the data in the relation database, and the *condor-dbmsd* daemon maintains the database itself.

3.11.1 Installation and Configuration

Quill uses the *PostgreSQL* database management system. Quill uses the *PostgreSQL* server as its back end and client library, *libpq* to talk to the server. We **strongly recommend** the use of version 8.2 or later due to its integrated facilities of certain key database maintenance tasks, and stronger security features.

Obtain *PostgreSQL* from

<http://www.postgresql.org/ftp/source/>

Installation instructions are detailed in: <http://www.postgresql.org/docs/8.2/static/installation.html>

Configure *PostgreSQL* after installation:

1. Initialize the database with the *PostgreSQL* command `initdb`.
2. Configure to accept TCP/IP connections. For *PostgreSQL* version 8, use the `listen_addresses` variable in `postgresql.conf` file as a guide. For example, `listen_addresses = '*'` means listen on any IP interface.
3. Configure automatic vacuuming. Ensure that these variables with these defaults are commented in and/or set properly in the `postgresql.conf` configuration file:

```
# Turn on/off automatic vacuuming
```

```

autovacuum = on

# time between autovacuum runs, in secs
autovacuum_naptime = 60

# min # of tuple updates before vacuum
autovacuum_vacuum_threshold = 1000

# min # of tuple updates before analyze
autovacuum_analyze_threshold = 500

# fraction of rel size before vacuum
autovacuum_vacuum_scale_factor = 0.4

# fraction of rel size before analyze
autovacuum_analyze_scale_factor = 0.2

# default vacuum cost delay for
#   autovac, -1 means use
#   vacuum_cost_delay
autovacuum_vacuum_cost_delay = -1

# default vacuum cost limit for
#   autovac, -1 means use
#   vacuum_cost_limit
autovacuum_vacuum_cost_limit = -1

```

4. Configure *PostgreSQL* to accept TCP/IP connections from specific hosts. Modify the `pg_hba.conf` file (which usually resides in the *PostgreSQL* server's data directory). Access is required by the *condor-quill* daemon, as well as the database users "**quillreader**" and "**quillwriter**". For example, to give database users "**quillreader**" and "**quillwriter**" password-enabled access to all databases on current machine from any machine in the 128.105.0.0/16 subnet, add the following:

```

host all quillreader 128.105.0.0 255.255.0.0 md5
host all quillwriter 128.105.0.0 255.255.0.0 md5

```

Note that in addition to the database specified by the configuration variable `QUILL_DB_NAME`, the *condor-quill* daemon also needs access to the database "template1". In order to create the database in the first place, the *condor-quill* daemon needs to connect to the database.

5. Start the *PostgreSQL* server service. See the installation instructions for the appropriate method to start the service at <http://www.postgresql.org/docs/8.2/static/installation.html>
6. The *condor-quill* and *condor-dbmsd* daemons and client tools connect to the database as users "**quillreader**" and "**quillwriter**". These are database users, not operating system users. The two types of users are quite different from each other. If these database users do not exist, add them using the *createuser* command supplied with the installation. Assign them with

appropriate passwords; these passwords will be used by the Quill tools to connect to the database in a secure way. User “**quillreader**” should not be allowed to create more databases nor create more users. User “**quillwriter**” should not be allowed to create more users, however it should be allowed to create more databases. The following commands create the two users with the appropriate permissions, and be ready to enter the corresponding passwords when prompted.

```
/path/to/postgreSQL/bin/directory/createuser quillreader \
--no-createdb --no-createrole --pwprompt

/path/to/postgreSQL/bin/directory/createuser quillwriter \
--createdb --no-createrole --pwprompt
```

Answer “no” to the question about the ability for role creation.

7. Create a database for Quill to store data in with the `createdb` command. Create this database with the “**quillwriter**” user as the owner. A sample command to do this is

```
createdb -O quillwriter quill
```

`quill` is the database name to use with the `QUILL_DB_NAME` configuration variable.

8. The `condor_quill` and `condor_dbmsd` daemons need read and write access to the database. They connect as user “**quillwriter**”, which has owner privileges to the database. Since this gives all access to the “**quillwriter**” user, its password cannot be stored in a public place (such as in a `ClassAd`). For this reason, the “**quillwriter**” password is stored in a file named `.pgpass` in the Condor spool directory. Appropriate protections on this file guarantee secure access to the database. This file must be created and protected by the site administrator; if this file does not exist as and where expected, the `condor_quill` and `condor_dbmsd` daemons log an error and exit. The `.pgpass` file contains exactly one line. The line has fields separated by colons. The first field may be either the machine name and fully qualified domain, or it may be a dotted quad IP address. This is followed by four fields containing: the TCP port number, the name of the database, the “**quillwriter**” user name, and the password. The form used in the first field must exactly match the value set for the configuration variable `QUILL_DB_IP_ADDR`. Condor uses a string comparison between the two, and it does not resolve the host names to compare IP addresses. Example:

```
machinename.cs.wisc.edu:5432:quill:quillwriter:password
```

After the *PostgreSQL* database is initialized and running, the Quill schema must be loaded into it. First, load the `plsql` programming language into the server:

```
createlang plpgsql [databasename]
```

Then, load the Quill schema from the `sql` files in the `sql` subdirectory of the Condor release directory:

```
psql [databasename] [username] < common_createddl.sql
psql [databasename] [username] < pgsqldb_createddl.sql
```

where [username] will be quillwriter.

After *PostgreSQL* is configured and running, Condor must also be configured to use Quill, since by default Quill is configured to be off.

Add the file `.pgpass` to the `VALID_SPOOL_FILES` variable, since *condor_preen* must be told not to delete this file. This step may not be necessary, depending on which version of Condor you are upgrading from.

Set up configuration variables that are specific to the installation, and check that the `HISTORY` variable is set.

```
HISTORY                = $(SPOOL)/history
QUILL_ENABLED          = TRUE
QUILL_USE_SQL_LOG      = FALSE
QUILL_NAME             = some-unique-quill-name.cs.wisc.edu
QUILL_DB_USER          = quillwriter
QUILL_DB_NAME          = database-for-some-unique-quill-name
QUILL_DB_IP_ADDR       = databaseIPAddress:port
# the following parameter's units is in seconds
QUILL_POLLING_PERIOD   = 10
QUILL_HISTORY_DURATION = 30
QUILL_MANAGE_VACUUM    = FALSE
QUILL_IS_REMOTELY_QUERYABLE = TRUE
QUILL_DB_QUERY_PASSWORD = password-for-database-user-quillreader
QUILL_ADDRESS_FILE     = $(LOG)/.quill_address
QUILL_DB_TYPE          = PGSQL
# The Purge and Reindex intervals are in seconds
DATABASE_PURGE_INTERVAL = 86400
DATABASE_REINDEX_INTERVAL = 86400
# The History durations are all in days
QUILL_RESOURCE_HISTORY_DURATION = 7
QUILL_RUN_HISTORY_DURATION = 7
QUILL_JOB_HISTORY_DURATION = 3650
#The DB Size limit is in gigabytes
QUILL_DBSIZE_LIMIT     = 20
QUILL_MAINTAIN_DB_CONN = TRUE
SCHEDD_SQLLOG          = $(LOG)/schedd_sql.log
SCHEDD_DAEMON_AD_FILE  = $(LOG)/.schedd_classad
```

The default Condor configuration file should already contain definitions for `QUILL` and `QUILL_LOG`. When upgrading from a previous version that did not have Quill to a new one that does, define these two configuration variables.

One machine should run the *condor_dbmsd* daemon. On this machine, add it to the `DAEMON_LIST` configuration variable. All Quill-enabled machines should also run the *condor_quill* daemon. The machine running the *condor_dbmsd* daemon can also run a *condor_quill* daemon. An example `DAEMON_LIST` for a machine running both daemons, and acting as both a submit machine and a central manager might look like the following:

```
DAEMON_LIST = MASTER, SCHEDD, COLLECTOR, NEGOTIATOR, DBMSD, QUILL
```

The *condor_dbmsd* daemon will need configuration file entries common to all daemons. If not already in the configuration file, add the following entries:

```
DBMSD = $(SBIN)/condor_dbmsd
DBMSD_ARGS = -f
DBMSD_LOG = $(LOG)/DbmsdLog
MAX_DBMSD_LOG = 10000000
```

Descriptions of these and other configuration variables are in section 3.3.30. Here are further brief details:

QUILL_DB_NAME and QUILL_DB_IP_ADDR These two variables are used to determine the location of the database server that this Quill would talk to, and the name of the database that it creates. More than one Quill server can talk to the same database server. This can be accomplished by letting all the *QUILL_DB_IP_ADDR* values point to the same database server.

QUILL_DB_USER This is the *PostgreSQL* user that Quill will connect as to the database. We recommend “**quillwriter**” for this setting. There is no default setting for *QUILL_DB_USER*, so it must be specified in the configuration file.

QUILLNAME Each *condor-quill* daemon in the pool has to be uniquely named.

QUILL_POLLING_PERIOD This controls the frequency with which Quill polls the *job-queue.log* file. By default, it is 10 seconds. Since Quill works by periodically sniffing the log file for updates and then sending those updates to the database, this variable controls the trade off between the currency of query results and Quill’s load on the system, which is usually negligible.

QUILL_RESOURCE_HISTORY_DURATION, QUILL_RUN_HISTORY_DURATION, and QUILL_JOB_HISTORY_DURATION

These three variables control the deletion of historical information from the database. *QUILL_RESOURCE_HISTORY_DURATION* is the number of days historical information about the state of a resource will be kept in the database. The default for resource history is 7 days. An example of a resource is the ClassAd for a compute slot. *QUILL_RUN_HISTORY_DURATION* is the number of days after completion that auxiliary information about a given job will stay in the database. This includes user log events, file transfers performed by the job, the matches that were made for a job, et cetera. The default for run history is 7 days. *QUILL_JOB_HISTORY_DURATION* is the number of days after completion that a given job will stay in the database. A more precise definition is the number of days since the history ad got into the history database; those two might be different, if a job is completed but stays in the queue for a while. The default for job history is 3,650 days (about 10 years.)

DATABASE_PURGE_INTERVAL As scanning the entire database for old jobs can be expensive, the other variable *DATABASE_PURGE_INTERVAL* is the number of seconds between two successive scans. *DATABASE_PURGE_INTERVAL* is set to 86400 seconds, or one day.

DATABASE_REINDEX_INTERVAL *PostgreSQL* does not aggressively maintain the index structures for deleted tuples. This can lead to bloated index structures. Quill can periodically reindex the database, which is controlled by the variable `DATABASE_REINDEX_INTERVAL`. `DATABASE_PURGE_INTERVAL` is set to 86400 seconds, or one day.

QUILL_DBSIZE_LIMIT Quill can estimate the size of the database, and send email to the Condor administrator if the database size exceeds this threshold. The estimate is checked after every `DATABASE_PURGE_INTERVAL`. The limit is given as gigabytes, and the default is 20.

QUILL_MAINTAIN_DB_CONN Quill can maintain an open connection to the database server, which speeds up updates to the database. However, each open connection consumes resources at the database server. The default is `TRUE`, but for large pools we recommend setting this `FALSE`.

QUILL_MANAGE_VACUUM Set to `False` by default, this variable determines whether Quill is to perform vacuuming tasks on its tables or not. Vacuuming is a maintenance task that needs to be performed on tables in *PostgreSQL*. The frequency with which a table is vacuumed typically depends on the number of updates (inserts/deletes) performed on the table. Fortunately, with *PostgreSQL* version 8.1, vacuuming tasks can be configured to be performed automatically by the database server. We recommend that users upgrade to 8.1 and use the integrated vacuuming facilities of the database server, instead of having Quill do them. If the user does prefer having Quill perform those vacuuming tasks, it can be achieved by setting this variable to `ExprTrue`. However, it cannot be overstated that Quill's vacuuming policy is quite rudimentary as compared to the integrated facilities of the database server, and under high update workloads, can prove to be a bottleneck on the Quill daemon. As such, setting this variable to `ExprTrue` results in some warning messages in the log file regarding this issue.

QUILL_IS_REMOTELY_QUERYABLE Thanks to *PostgreSQL*, one can now remotely query both the job queue and the history tables. This variable controls whether this remote querying feature should be enabled. By default it is `True`. Note that even if this is `False`, one can still query the job queue at the remote *condor_schedd* daemon. This variable only controls whether the database tables are remotely queryable.

QUILL_DB_QUERY_PASSWORD In order for the query tools to connect to a database, they need to provide the password that is assigned to the database user “*quillreader*”. This variable is then advertised by the *condor_quill* daemon to the *condor_collector*. This facility enables remote querying: remote *condor_q* query tools first ask the *condor_collector* for the password associated with a particular Quill database, and then query that database. Users who do not have access to the *condor_collector* cannot view the password, and as such cannot query the database. Again, this password only provides read access to the database.

QUILL_ADDRESS_FILE When Quill starts up, it can place its address (IP and port) into a file. This way, tools running on the local machine do not need to query the central manager to find Quill. This feature can be turned off by commenting out the variable.

3.11.2 Four Usage Examples

1. Query a remote Quill daemon on `regular.cs.wisc.edu` for all the jobs in the queue

```
condor_q -name quill@regular.cs.wisc.edu
condor_q -name schedd@regular.cs.wisc.edu
```

There are two ways to get to a Quill daemon: directly using its name as specified in the `QUILL_NAME` configuration variable, or indirectly by querying the `condor_schedd` daemon using its name. In the latter case, `condor_q` will detect if that `condor_schedd` daemon is being serviced by a database, and if so, directly query it. In both cases, the IP address and port of the database server hosting the data of this particular remote Quill daemon can be figured out by the `QUILL_DB_IP_ADDR` and `QUILL_DB_NAME` variables specified in the `QUILL_AD` sent by the quill daemon to the collector and in the `SCHEDD_AD` sent by the `condor_schedd` daemon.

2. Query a remote Quill daemon on `regular.cs.wisc.edu` for all historical jobs belonging to owner `einstein`.

```
condor_history -name quill@regular.cs.wisc.edu einstein
```

3. Query the local Quill daemon for the average time spent in the queue for all non-completed jobs.

```
condor_q -avgqueueetime
```

The average queue time is defined as the average of `(currenttime - jobsubmissiontime)` over all jobs which are neither completed (`JobStatus == 4`) or removed (`JobStatus == 3`).

4. Query the local Quill daemon for all historical jobs completed since Apr 1, 2005 at 13h 00m.

```
condor_history -completedsince '04/01/2005 13:00'
```

It fetches all jobs which got into the 'Completed' state on or after the specified time stamp. It use the *PostgreSQL* date/time syntax rules, as it encompasses most format options. See <http://www.postgresql.org/docs/8.2/static/datatype-datetime.html> for the various time stamp formats.

3.11.3 Quill and Security

There are several layers of security in Quill, some provided by Condor and others provided by the database. First, all accesses to the database are password-protected.

1. The query tools, `condor_q` and `condor_history` connect to the database as user “**quillreader**”. The password for this user can vary from one database to another and as such, each Quill daemon advertises this password to the collector. The query tools then obtain this password from the collector and connect successfully to the database. Access to the database by the

“**quillreader**” user is read-only, as this is sufficient for the query tools. The *condor_quill* daemon ensures this protected access using the sql GRANT command when it first creates the tables in the database. Note that access to the “**quillreader**” password itself can be blocked by blocking access to the collector, a feature already supported in Condor.

2. The *condor_quill* and *condor_dbmsd* daemons, on the other hand, need read and write access to the database. As such, they connect as user “**quillwriter**”, who has owner privileges to the database. Since this gives all access to the “**quillwriter**” user, this password cannot be stored in a public place (such as the collector). For this reason, the “**quillwriter**” password is stored in a file called `.pgpass` in the Condor spool directory. Appropriate protections on this file guarantee secure access to the database. This file must be created and protected by the site administrator; if this file does not exist as and where expected, the *condor_quill* daemon logs an error and exits.
3. The `IsRemotelyQueryable` attribute in the Quill ClassAd advertised by the Quill daemon to the collector can be used by site administrators to disallow the database from being read by all remote Condor query tools.

3.11.4 Quill and Its RDBMS Schema

Notes:

- The type “timestamp(*precision*) with timezone” is abbreviated “ts(*precision*) w tz.”
- The column O. Type is an abbreviation for Oracle Type.
- The column P. Type is an abbreviation for PostgreSQL Type.

Although the current version of Condor does not support Oracle, we anticipate supporting it in the future, so Oracle support in this schema document is for future reference.

Administrative Tables

Attributes of currencies Table			
Name	O. Type	P. Type	Description
datasource	varchar(4000)	varchar(4000)	Identifier of the data source.
lastupdate	ts(3) w tz	ts(3) w tz	Time of the last update sent to the database from the data source.

Attributes of error_sqllogs Table			
Name	O. Type	P. Type	Description
logname	varchar(100)	varchar(100)	Name of the SQL log file causing a SQL error.
host	varchar(50)	varchar(50)	The host where the SQL log resides.
lastmodified	ts(3) w tz	ts(3) w tz	The last modified time of the SQL log.
errorsql	varchar(4000)	text	The SQL statement causing an error.
logbody	clob	text	The body of the SQL log.
errormessage	varchar(4000)	varchar(4000)	The description of the error.
INDEX: Index named error_sqllog_idx on (logname, host, lastmodified)			

Attributes of maintenance_log Table			
Name	O. Type	P. Type	Description
eventts	ts(3) w tz	ts(3) w tz	Time the event occurred.
eventmsg	varchar(4000)	varchar(4000)	Message describing the event.

Attributes of quilldbmonitor Table			
Name	O. Type	P. Type	Description
dbsize	integer	integer	Size of the database in megabytes.

Attributes of quill_schema_version Table			
Name	O. Type	P. Type	Description
major	int	int	Major version number.
minor	int	int	Minor version number.
back_to_major	int	int	The major number of the old version this version is compatible to.
back_to_minor	int	int	The minor number of the old version this version is compatible to.

Attributes of throws Table			
Name	O. Type	P. Type	Description
filename	varchar(4000)	varchar(4000)	The name of the log that was truncated.
machine_id	varchar(4000)	varchar(4000)	The machine where the truncated log resides.
log_size	numeric(38)	numeric(38)	The size of the truncated log.
throwtime	ts(3) w tz	ts(3) w tz	The time when the truncation occurred.

Daemon Tables

Attributes of daemons_horizontal Table			
Name	O. Type	P. Type	Description
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. "Master"
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
monitorselftime	ts(3) w tz	ts(3) w tz	The time when the daemon last collected information about itself.
monitorselfcpuusage	numeric(38)	numeric(38)	The amount of CPU this daemon has used.
monitorselfimagesize	numeric(38)	numeric(38)	The amount of virtual memory this daemon has used.
monitorselfresidentsetsize	numeric(38)	numeric(38)	The amount of physical memory this daemon has used.
monitorselfage	integer	integer	How long the daemon has been running.
updatesequencenumber	integer	integer	The sequence number associated with the update.
updatetotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
updateshistory	varchar(4000)	varchar(4000)	Bitmask of the last 32 updates.
lastreportedtime_epoch	integer	integer	The equivalent epoch time of last heard from.
PRIMARY KEY: (mytype, name)			
NOT NULL: mytype and name cannot be null			

Attributes of daemons_horizontal_history Table			
Name	O. Type	P. Type	Description
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. "Master"
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
monitorselftime	ts(3) w tz	ts(3) w tz	The time when the daemon last collected information about itself.
monitorselfcpuusage	numeric(38)	numeric(38)	The amount of CPU this daemon has used.
monitorselfimagesize	numeric(38)	numeric(38)	The amount of virtual memory this daemon has used.
monitorselfresidentsetsize	numeric(38)	numeric(38)	The amount of physical memory this daemon has used.
monitorselfage	integer	integer	How long the daemon has been running.
updatesequencenumber	integer	integer	The sequence number associated with the update.
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
updateshistory	varchar(4000)	varchar(4000)	Bitmask of the last 32 updates.
endtime	ts(3) w tz	ts(3) w tz	End of when the ClassAd is valid.

Attributes of daemons_vertical Table			
Name	O. Type	P. Type	Description
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. "Master"
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
attr	varchar(4000)	varchar(4000)	Attribute name.
val	clob	text	Attribute value.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
PRIMARY KEY: (mytype, name, attr)			
NOT NULL: mytype, name, and attr cannot be null			

Attributes of daemons_vertical_history Table			
Name	O. Type	P. Type	Description
mytype	varchar(100)	varchar(100)	The type of daemon ClassAd, e.g. "Master"
name	varchar(500)	varchar(500)	The name identifier of the daemon ClassAd.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the daemon last reported to Quill.
attr	varchar(4000)	varchar(4000)	Attribute name.
val	clob	text	Attribute value.
endtime	ts(3) w tz	ts(3) w tz	End of when the ClassAd is valid.

Attributes of submitters_horizontal table			
Name	O. Type	P. Type	Description
name	varchar(500)	varchar(500)	Name of the submitter ClassAd.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd where the submitter ad is from.
lastreportedtime	ts(3) w tz	ts(3) w tz	Last time a submitter ClassAd was sent to Quill.
idlejobs	integer	integer	Number of idle jobs of the submitter.
runningjobs	integer	integer	Number of running jobs of the submitter.
heldjobs	integer	integer	Number of held jobs of the submitter.
flockedjobs	integer	integer	Number of flocked jobs of the submitter.

Attributes of submitters_horizontal_history table			
Name	O. Type	P. Type	Description
name	varchar(500)	varchar(500)	Name of the submitter ClassAd.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd where the submitter ad is from.
lastreportedtime	ts(3) w tz	ts(3) w tz	Last time a submitter ClassAd was sent to Quill.
idlejobs	integer	integer	Number of idle jobs of the submitter.
runningjobs	integer	integer	Number of running jobs of the submitter.
heldjobs	integer	integer	Number of held jobs of the submitter.
flockedjobs	integer	integer	Number of flocked jobs of the submitter.
endtime	ts(3) w tz	ts(3) w tz	End of when the ClassAd is valid.

Files Tables

Attributes of files Table			
Name	O. Type	P. Type	Description
file_id	int	int	Unique numeric identifier of the file.
name	varchar(4000)	varchar(4000)	File name.
host	varchar(4000)	varchar(4000)	Name of machine where the file is located.
path	varchar(4000)	varchar(4000)	Directory path to the file.
acl_id	integer	integer	Not yet used, null.
lastmodified	ts(3) w tz	ts(3) w tz	Timestamp of the file.
filesize	numeric(38)	numeric(38)	Size of the file in bytes.
checksum	varchar(32)	varchar(32)	MD5 checksum of the file.
PRIMARY KEY: file_id			
NOT NULL: file_id cannot be null			

Attributes of fileusages Table			
Name	O. Type	P. Type	Description
globaljobid	varchar(4000)	varchar(4000)	Global identifier of the job that used the file.
file_id	int	int	Numeric identifier of the file.
usagetype	varchar(4000)	varchar(4000)	Type of use of the file by the job, e.g., input, output, command.
REFERENCE: file_id references files(file_id)			

Attributes of transfers Table			
Name	O. Type	P. Type	Description
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
src_name	varchar(4000)	varchar(4000)	Name of the file on the source machine.
src_host	varchar(4000)	varchar(4000)	Name of the source machine.
src_port	integer	integer	Source port number used for the transfer.
src_path	varchar(4000)	varchar(4000)	Path to the file on the source machine.
src_daemon	varchar(30)	varchar(30)	Condor demon performing the transfer on the source machine.
src_protocol	varchar(30)	varchar(30)	The protocol used on the source machine.
src_credential_id	integer	integer	Not yet used, null.
src_acl_id	integer	integer	Not yet used, null.
dst_name	varchar(4000)	varchar(4000)	Name of the file on the destination machine.
dst_host	varchar(4000)	varchar(4000)	Name of the destination machine.
dst_port	integer	integer	Destination port number used for the transfer.
dst_path	varchar(4000)	varchar(4000)	Path to the file on the destination machine.
dst_daemon	varchar(30)	varchar(30)	Condor daemon receiving the transfer on the destination machine.
dst_protocol	varchar(30)	varchar(30)	The protocol used on the destination machine.
dst_credential_id	integer	integer	Not yet used, null.
dst_acl_id	integer	integer	Not yet used, null.
transfer_intermediary_id	integer	integer	Not yet used, null; will use someday if a proxy is used.
transfer_size_bytes	numeric(38)	numeric (38)	Size of the data transferred in bytes.
elapsed	numeric(38)	numeric(38)	Number of seconds that elapsed during the transfer.
checksum	varchar(256)	varchar(256)	Checksum of the file.
transfer_time	ts(3) w tz	ts(3) w tz	Time when the transfer took place.
last_modified	ts(3) w tz	ts(3) w tz	Last modified time for the file that was transferred.
is_encrypted	varchar(5)	varchar(5)	(boolean) True if the file is encrypted.
delegation_method_id	integer	integer	Not yet used, null.
completion_code	integer	integer	Indicates whether the transfer failed or succeeded.

Interface Tables

Attributes of cdb_users Table			
Name	O. Type	P. Type	Description
userid	varchar(30)	varchar(30)	Unique identifier of the user
password	character(32)	character(32)	Encrypted password
admin	varchar(5)	varchar(5)	(boolean) True if the user has administrator privileges

Attributes of l_eventtype Table			
Name	O. Type	P. Type	Description
eventtype	integer	integer	Numeric type code of the event.
description	varchar(4000)	varchar(4000)	Description of the type of event associated with the event-type code.

Attributes of l_jobstatus Table			
Name	O. Type	P. Type	Description
jobstatus	integer	integer	Numeric code for job status.
abbrev	char(1)	char(1)	Single letter code for job status.
description	varchar(4000)	varchar(4000)	Description of job status.
PRIMARY KEY: jobstatus			
NOT NULL: jobstatus cannot be null			

Jobs Tables

Attributes of clusterads_horizontal Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd the job is submitted to.
cluster_id	integer	integer	Cluster identifier for the job.
owner	varchar(30)	varchar(30)	User who submitted the job.
jobstatus	integer	integer	Current status of the job.
jobprio	integer	integer	Priority for this job.
imagesize	numeric(38)	numeric(38)	Estimate of memory image size of the job in kilobytes.
qdate	ts(3) w tz	ts(3) w tz	Time the job was submitted to the job queue.
remoteusercpu	numeric(38)	numeric(38)	Total number of seconds of user CPU time the job used on remote machines.
remotewallclocktime	numeric(38)	numeric(38)	Committed cumulative number of seconds the job has been allocated to a machine.
cmd	clob	text	Path to and filename of the job to be executed.
args	clob	text	Arguments passed to the job.
jobuniverse	integer	integer	The Condor universe used by the job.
PRIMARY KEY: (scheddname, cluster_id)			
NOT NULL: scheddname and cluster_id cannot be null			

Attributes of clusterads_vertical Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that the job is submitted to.
cluster_id	integer	integer	Cluster identifier for the job.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.
PRIMARY KEY: (scheddname, cluster_id, attr)			

Attributes of jobs_horizontal_history Table – Part 1 of 3			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
scheddbirthdate	integer	integer	The birth date of the schedd where the job is submitted.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
qdate	ts(3) w tz	ts(3) w tz	Time the job was submitted to the job queue.
owner	varchar(30)	varchar(30)	User who submitted the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
numckpts	integer	integer	Number of checkpoints written by the job during its lifetime.
numrestarts	integer	integer	Number of restarts from a checkpoint attempted by the job in its lifetime.
numsystemholds	integer	integer	Number of times Condor-G placed the job on hold.
condorversion	varchar(4000)	varchar(4000)	Version of Condor that ran the job.
condorplatform	varchar(4000)	varchar(4000)	Platform of the computer where the schedd runs.
rootdir	varchar(4000)	varchar(4000)	Root directory on the system where the job is submitted from.
iwd	varchar(4000)	varchar(4000)	Initial working directory of the job.
jobuniverse	integer	integer	The Condor universe used by the job.
cmd	clob	text	Path to and filename of the job to be executed.
minhosts	integer	integer	Minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.
maxhosts	integer	integer	Maximum number of hosts this job would like to claim.
jobprio	integer	integer	Priority for this job.
negotiation_user_name	varchar(4000)	varchar(4000)	User name in which the job is negotiated.
env	clob	text	Environment under which the job ran.
userlog	varchar(4000)	varchar(4000)	User log where the job events are written to.
coresize	numeric(38)	numeric(38)	Maximum allowed size of the core file.
Table Continues on Next Page			

Attributes of jobs_horizontal_history Table – Part 2 of 3			
Name	O. Type	P. Type	Description
killsig	varchar(4000)	varchar(4000)	Signal to be sent if the job is put on hold.
stdin	varchar(4000)	varchar(4000)	The file used as stdin.
transferin	varchar(5)	varchar(5)	(boolean) For globus universe jobs. True if input should be transferred to the remote machine.
stdout	varchar(4000)	varchar(4000)	The file used as stdout.
transferout	varchar(5)	varchar(5)	(boolean) For globus universe jobs. True if output should be transferred back to the submit machine.
stderr	varchar(4000)	varchar(4000)	The file used as stderr.
transfererr	varchar(5)	varchar(5)	(boolean) For globus universe jobs. True if error output should be transferred back to the submit machine.
shouldtransferfiles	varchar(4000)	varchar(4000)	Whether Condor should transfer files to and from the machine where the job runs.
transferfiles	varchar(4000)	varchar(4000)	Deprecated. Similar to shouldtransferfiles.
executablesize	numeric(38)	numeric(38)	Size of the executable in kilobytes.
diskusage	integer	integer	Size of the executable and input files to be transferred.
filesystemdomain	varchar(4000)	varchar(4000)	Name of the networked file system used by the job.
args	clob	text	Arguments passed to the job.
lastmatchtime	ts(3) w tz	ts(3) w tz	Time when the job was last successfully matched with a resource.
numjobmatches	integer	integer	Number of times the negotiator matches the job with a resource.
jobstartdate	ts(3) w tz	ts(3) w tz	Time when the job first began running.
jobcurrentstartdate	ts(3) w tz	ts(3) w tz	Time when the job's current run started.
jobruncount	integer	integer	Number of times a shadow has been started for the job.
filereadcount	numeric(38)	numeric(38)	Number of read(2) calls the job made (only standard universe).
filereadbytes	numeric(38)	numeric(38)	Number of bytes read by the job (only standard universe).
filewritecount	numeric(38)	numeric(38)	Number of write calls the job made (only standard universe).
filewritebytes	numeric(38)	numeric(38)	Number of bytes written by the job (only standard universe).
Table Continues on Next Page			

Attributes of jobs.horizontal_history Table – Part 3 of 3			
Name	O. Type	P. Type	Description
fileseekcount	numeric(38)	numeric(38)	Number of seek calls that this job made (only standard universe).
totalsuspensions	integer	integer	Number of times the job has been suspended during its lifetime
imagesize	numeric(38)	numeric(38)	Estimate of memory image size of the job in kilobytes.
exitstatus	integer	integer	No longer used by Condor.
localusercpu	numeric(38)	numeric(38)	Number of seconds of user CPU time the job used on the submit machine.
localsyscpu	numeric(38)	numeric(38)	Number of seconds of system CPU time the job used on the submit machine.
remoteusercpu	numeric(38)	numeric(38)	Number of seconds of user CPU time the job used on remote machines.
remotesyscpu	numeric(38)	numeric(38)	Number of seconds of system CPU time the job used on remote machines.
bytessent	numeric(38)	numeric(38)	Number of bytes sent to the job.
bytesrecvd	numeric(38)	numeric(38)	Number of bytes received by the job.
rsbytessent	numeric(38)	numeric(38)	Number of remote system call bytes sent to the job.
rsbytesrecvd	numeric(38)	numeric(38)	Number of remote system call bytes received by the job.
exitcode	integer	integer	Exit return code of the user job. Used when a job exits by means other than a signal.
jobstatus	integer	integer	Current status of the job.
enteredcurrentstatus	ts(3) w tz	ts(3) w tz	Time the job entered into its current status.
remotewallclocktime	numeric(38)	numeric(38)	Cumulative number of seconds the job has been allocated to a machine.
lastremotehost	varchar(4000)	varchar(4000)	The remote host for the last run of the job.
completiondate	ts(3) w tz	ts(3) w tz	Time when the job completed; 0 if job has not yet completed.
enteredhistorytable	ts(3) w tz	ts(3) w tz	Time when the job entered the history table.
PRIMARY KEY: (scheddname, scheddbirthdate, cluster_id, proc_id)			
NOT NULL: scheddname, scheddbirthdate, cluster_id, and proc_id cannot be null			
INDEX: Index named hist_h_i_owner on owner			

Attributes of jobs_vertical_history Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
scheddbirthdate	integer	integer	The birth date of the schedd where the job is submitted.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.
PRIMARY KEY: (scheddname, scheddbirthdate, cluster_id, proc_id, attr)			
NOT NULL: scheddname, scheddbirthdate, cluster_id, proc_id, and attr cannot be null			

Attributes of procads_horizontal Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
jobstatus	integer	integer	Current status of the job.
imagesize	numeric(38)	numeric(38)	Estimate of memory image size of the job in kilobytes.
remoteusercpu	numeric(38)	numeric(38)	Total number of seconds of user CPU time the job used on remote machines.
remotewallclocktime	numeric(38)	numeric(38)	Cumulative number of seconds the job has been allocated to a machine.
remotehost	varchar(4000)	varchar(4000)	Name of the machine running the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
jobprio	integer	integer	Priority of the job.
args	clob	text	Arguments passed to the job.
shadowbday	ts(3) w tz	ts(3) w tz	The time when the shadow was started.
enteredcurrentstatus	ts(3) w tz	ts(3) w tz	Time the job entered its current status.
numrestarts	integer	integer	Number of times the job has restarted.
PRIMARY KEY: (scheddname, cluster_id, proc_id)			
NOT NULL: scheddname, cluster_id, and proc_id cannot be null			

Attributes of procads_vertical Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.

Machines Tables

Attributes of machines_horizontal Table – Part 1 of 2			
Name	O. Type	P. Type	Description
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
opsys	varchar(4000)	varchar(4000)	Operating system running on the machine.
arch	varchar(4000)	varchar(4000)	Architecture of the machine.
state	varchar(4000)	varchar(4000)	Condor state of the machine.
activity	varchar(4000)	varchar(4000)	Condor job activity on the machine.
keyboardidle	integer	integer	Number of seconds since activity has been detected on any keyboard or mouse associated with the machine.
consoleidle	integer	integer	Number of seconds since activity has been detected on the console keyboard or mouse.
loadavg	real	real	Current load average of the machine.
condorloadavg	real	real	Portion of load average generated by Condor
totalloadavg	real	real	
virtualmemory	integer	integer	Amount of currently available virtual memory in kilobytes.
memory	integer	integer	Amount of RAM in megabytes.
totalvirtualmemory	integer	integer	
cpubusytime	integer	integer	Time in seconds since cpubusy became true.
cpubusy	varchar(5)	varchar(5)	(boolean) True when the CPU is busy.
currentrank	real	real	The machine owner's affinity for running the Condor job which it is currently hosting.
clockmin	integer	integer	Number of minutes passed since midnight.
clockday	integer	integer	The day of the week.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the Condor central manager last received a status update from this machine.
enteredcurrentactivity	ts(3) w tz	ts(3) w tz	Time when the machine entered the current activity.
enteredcurrentstate	ts(3) w tz	ts(3) w tz	Time when the machine entered the current state.
updatesequencenumber	integer	integer	Each update includes a sequence number.
Table Continues on Next Page			

Attributes of machines_horizontal Table – Part 2 of 2			
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
lastreportedtime_epoch	integer	integer	The equivalent epoch time of lastreported-time.
PRIMARY KEY: machine_id			

Attributes of machines_horizontal_history Table – Part 1 of 2			
Name	O. Type	P. Type	Description
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
opsys	varchar(4000)	varchar(4000)	Operating system running on the machine.
arch	varchar(4000)	varchar(4000)	Architecture of the machine.
state	varchar(4000)	varchar(4000)	Condor state of the machine.
activity	varchar(4000)	varchar(4000)	Condor job activity on the machine.
keyboardidle	integer	integer	Number of seconds since activity has been detected on any keyboard or mouse associated with the machine.
consoleidle	integer	integer	Number of seconds since activity has been detected on the console keyboard or mouse.
loadavg	real	real	Current load average of the machine.
condorloadavg	real	real	Portion of load average generated by Condor
totalloadavg	real	real	
virtualmemory	integer	integer	Amount of currently available virtual memory in kilobytes.
memory	integer	integer	Amount of RAM in megabytes.
totalvirtualmemory	integer	integer	
cpubusytime	integer	integer	Time in seconds since cpuisbusy became true.
cpuisbusy	varchar(5)	varchar(5)	(boolean) True when the CPU is busy.
currentrank	real	real	The machine owner's affinity for running the Condor job which it is currently hosting.
clockmin	integer	integer	Number of minutes passed since midnight.
clockday	integer	integer	The day of the week.
lastreportedtime	ts(3) w tz	ts(3) w tz	Time when the Condor central manager last received a status update from this machine.
enteredcurrentactivity	ts(3) w tz	ts(3) w tz	Time when the machine entered the current activity.
enteredcurrentstate	ts(3) w tz	ts(3) w tz	Time when the machine entered the current state.
updatesequencenumber	integer	integer	Each update includes a sequence number.
Table Continues on Next Page			

Attributes of machines_horizontal_history Table – Part 2 of 2			
Name	O. Type	P. Type	Description
updatestotal	integer	integer	The number of updates received from the daemon.
updatessequenced	integer	integer	The number of updates that were in order.
updateslost	integer	integer	The number of updates that were lost.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
end_time	ts(3) w tz	ts(3) w tz	The end of when the ClassAd is valid.

Attributes of machines_vertical Table			
Name	O. Type	P. Type	Description
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
attr	varchar(2000)	varchar(2000)	Attribute name.
val	clob	text	Attribute value.
start_time	ts(3) w tz	ts(3) w tz	Time when this attribute–value pair became valid.
PRIMARY KEY: (machine_id, attr)			
NOT NULL: machine_id and attr cannot be null			

Attributes of machines_vertical_history Table			
Name	O. Type	P. Type	Description
machine_id	varchar(4000)	varchar(4000)	Unique identifier of the machine.
attr	varchar(4000)	varchar(4000)	Attribute name.
val	clob	text	Attribute value.
start_time	ts(3) w tz	ts(3) w tz	Time when this attribute–value pair became valid.
end_time	ts(3) w tz	ts(3) w tz	Time when this attribute–value pair became invalid.

Matchmaking Tables

Attributes of matches Table			
Name	O. Type	P. Type	Description
match_time	ts(3) w tz	ts(3) w tz	Time the match was made.
username	varchar(4000)	varchar(4000)	User who submitted the job.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that the job is submitted to.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.
machine_id	varchar(4000)	varchar(4000)	Identifier of the machine the job matched with.
remote_user	varchar(4000)	varchar(4000)	User that was preempted.
remote_priority	real	real	The preempted user's priority.

Attributes of rejects Table			
Name	O. Type	P. Type	Description
reject_time	ts(3) w tz	ts(3) w tz	Time when the job was rejected.
username	varchar(4000)	varchar(4000)	User who submitted the job.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.

Runtime Tables

Attributes of events Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Global identifier of the job that generated the event.
run_id	numeric(12,0)	numeric(12,0)	Identifier of the run that the event is associated with.
eventtype	integer	integer	Numeric type code of the event.
eventtime	ts(3) w tz	ts(3) w tz	Time the event occurred.
description	varchar(4000)	varchar(4000)	Description of the event.

Attributes of generic_messages Table			
Name	O. Type	P. Type	Description
eventtype	varchar(4000)	varchar(4000)	The type of event.
eventkey	varchar(4000)	varchar(4000)	The key of the event.
eventtime	ts(3) w tz	ts(3) w tz	The time of the event.
eventloc	varchar(4000)	varchar(4000)	The location of the event.
attrname	varchar(4000)	varchar(4000)	The attribute name.
attrval	clob	text	The attribute value.
attrtype	varchar(4000)	varchar(4000)	The attribute type.

Attributes of runs Table			
Name	O. Type	P. Type	Description
run_id	numeric(12)	numeric(12)	Unique identifier of the run.
machine_id	varchar(4000)	varchar(4000)	Identifier of the machine where the job ran.
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
spid	integer	integer	Subprocess identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Identifier of the job that was run.
startts	ts(3) w tz	ts(3) w tz	Time when the job started.
endts	ts(3) w tz	ts(3) w tz	Time when the job ended.
endtype	smallint	smallint	The type of ending event.
endmessage	varchar(4000)	varchar(4000)	The ending message.
wascheckpointed	varchar(7)	varchar(7)	Whether the run was checkpointed.
imagesize	numeric(38)	numeric(38)	The image size of the executable.
runlocalusageuser	integer	integer	The time the job spent in usermode on execute machines (only standard universe).
runlocalusagesystem	integer	integer	The time the job was in system calls.
runremoteusageuser	integer	integer	The time the shadow spent working for the job.
runremoteusagesystem	integer	integer	The time the shadow spent in system calls for the job.
runbytessent	numeric(38)	numeric(38)	Number of bytes sent to the run.
runbytesreceived	numeric(38)	numeric(38)	Number of bytes received from the run.
PRIMARY KEY: run_id			
NOT NULL: run_id cannot be null			

System Tables

Attributes of dummy_single_row_table Table			
Name	O. Type	P. Type	Description
a	varchar(1)	varchar(1)	A dummy column.

Attributes of history_jobs_to_purge Table			
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
cluster_id	integer	integer	Cluster identifier for the job.
proc_id	integer	integer	Process identifier for the job.
globaljobid	varchar(4000)	varchar(4000)	Unique global identifier for the job.

Attributes of jobqueuepollinginfo Table			
Name	O. Type	P. Type	Description
scheddname	varchar(4000)	varchar(4000)	Name of the schedd that submitted the job.
last_file_mtime	integer	integer	The last modification time of the file.
last_file_size	numeric(38)	numeric(38)	The last size of the file in bytes.
last_next_cmd_offset	integer	integer	The last offset for the next command.
last_cmd_offset	integer	integer	The last offset of the current command.
last_cmd_type	smallint	smallint	The last type of command.
last_cmd_key	varchar(4000)	varchar(4000)	The last key of the command.
last_cmd_mytype	varchar(4000)	varchar(4000)	The last my ClassAd type of the command.
last_cmd_targettype	varchar(4000)	varchar(4000)	The last target ClassAd type.
last_cmd_name	varchar(4000)	varchar(4000)	The attribute name of the command.
last_cmd_value	varchar(4000)	varchar(4000)	The attribute value of the command.

3.12 Setting Up for Special Environments

The following sections describe how to set up Condor for use in special environments or configurations. See section ?? on page ?? for installation instructions on the various Contrib modules that can be optionally downloaded and installed.

3.12.1 Using Condor with AFS

If you are using AFS at your site, be sure to read section 3.3.7 on “Shared Filesystem Config Files Entries” for details on configuring your machines to interact with and use shared filesystems, AFS in particular.

Condor does not currently have a way to authenticate itself to AFS. This is true of the Condor daemons that would like to authenticate as AFS user Condor, and the *condor_shadow*, which would like to authenticate as the user who submitted the job it is serving. Since neither of these things can happen yet, there are a number of special things people who use AFS with Condor must do. Some of this must be done by the administrator(s) installing Condor. Some of this must be done by Condor users who submit jobs.

AFS and Condor for Administrators

The most important thing is that since the Condor daemons can't authenticate to AFS, the `LOCAL_DIR` (and it's subdirectories like "log" and "spool") for each machine must be either writable to unauthenticated users, or must not be on AFS. The first option is a **VERY** bad security hole so you should **NOT** have your local directory on AFS. If you've got NFS installed as well and want to have your `LOCAL_DIR` for each machine on a shared file system, use NFS. Otherwise, you should put the `LOCAL_DIR` on a local partition on each machine in your pool. This means that you should run *condor_configure* to install your release directory and configure your pool, setting the `LOCAL_DIR` parameter to some local partition. When that's complete, log into each machine in your pool and run *condor_init* to set up the local Condor directory.

The `RELEASE_DIR`, which holds all the Condor binaries, libraries and scripts can and probably should be on AFS. None of the Condor daemons need to write to these files, they just need to read them. So, you just have to make your `RELEASE_DIR` world readable and Condor will work just fine. This makes it easier to upgrade your binaries at a later date, which means that your users can find the Condor tools in a consistent location on all the machines in your pool, and that you can have the Condor config files in a centralized location. This is what we do at UW-Madison's CS department Condor pool and it works quite well.

Finally, you might want to setup some special AFS groups to help your users deal with Condor and AFS better (you'll want to read the section below anyway, since you're probably going to have to explain this stuff to your users). Basically, if you can, create an AFS group that contains all unauthenticated users but that is restricted to a given host or subnet. You're supposed to be able to make these host-based ACLs with AFS, but we've had some trouble getting that working here at UW-Madison. What we have instead is a special group for all machines in our department. So, the users here just have to make their output directories on AFS writable to any process running on any of our machines, instead of any process on any machine with AFS on the Internet.

AFS and Condor for Users

The *condor_shadow* process runs on the machine where you submitted your Condor jobs and performs all file system access for your jobs. Because this process isn't authenticated to AFS as the user who submitted the job, it will not normally be able to write any output. So, when you submit jobs, any directories where your job will be creating output files will need to be world writable (to non-authenticated AFS users). In addition, if your program writes to `stdout` or `stderr`, or you're using a user log for your jobs, those files will need to be in a directory that's world-writable.

Any input for your job, either the file you specify as input in your submit file, or any files your program opens explicitly, needs to be world-readable.

Some sites may have special AFS groups set up that can make this unauthenticated access to your files less scary. For example, there's supposed to be a way with AFS to grant access to any unauthenticated process on a given host. That way, you only have to grant write access to unauthenticated processes on your submit machine, instead of any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running on your submit machine. Ask your AFS administrators about the existence of such AFS groups and details of how to use them.

The other solution to this problem is to just not use AFS at all. If you have disk space on your submit machine in a partition that is not on AFS, you can submit your jobs from there. While the *condor_shadow* is not authenticated to AFS, it does run with the effective UID of the user who submitted the jobs. So, on a local (or NFS) file system, the *condor_shadow* will be able to access your files normally, and you won't have to grant any special permissions to anyone other than yourself. If the Condor daemons are not started as root however, the shadow will not be able to run with your effective UID, and you'll have a similar problem as you would with files on AFS. See the section on "Running Condor as Non-Root" for details.

3.12.2 Configuring Condor for Multiple Platforms

A single, global configuration file may be used for all platforms in a Condor pool, with only platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing changes of platform-independent, global settings in one place, instead of separately for each platform. This is made possible by treating the `LOCAL_CONFIG_FILE` configuration variable as a list of files, instead of a single file. Of course, this only helps when using a shared file system for the machines in the pool, so that multiple machines can actually share a single set of configuration files.

With multiple platforms, put all platform-independent settings (the vast majority) into the regular `condor_config` file, which would be shared by all platforms. This global file would be the one that is found with the `CONDOR_CONFIG` environment variable, the user `condor`'s home directory, or `/etc/condor/condor_config`.

Then set the `LOCAL_CONFIG_FILE` configuration variable from that global configuration file to specify both a platform-specific configuration file and optionally, a local, machine-specific configuration file (this parameter is described in section 3.3.3 on "Condor-wide Configuration File Entries").

The order of file specification in the `LOCAL_CONFIG_FILE` configuration variable is important, because settings in files at the beginning of the list are overridden if the same settings occur in files later within the list. So, if specifying the platform-specific file and then the machine-specific file, settings in the machine-specific file would override those in the platform-specific file (as is likely desired).

Utilizing a Platform-Specific Configuration File

The name of platform-specific configuration files may be specified by using the `ARCH` and `OPSYS` parameters, as are defined automatically by Condor. For example, for Intel Linux machines, and Sparc Solaris 2.6 machines, the files ought to be named:

```
condor_config.INTEL.LINUX
condor_config.SUN4x.SOLARIS26
```

Then, assuming these three files are in the directory defined by the `ETC` configuration macro, and machine-specific configuration files are in the same directory, named by each machine's host name, the `LOCAL_CONFIG_FILE` configuration macro should be:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.$(ARCH).$(OPSYS), \
                    $(ETC)/$(HOSTNAME).local
```

Alternatively, when using AFS, an “@sys link” may be used to specify the platform-specific configuration file, and let AFS resolve this link differently on different systems. For example, consider a soft link named `condor_config.platform` that points to `condor_config.@sys`. In this case, the files might be named:

```
condor_config.i386_linux2
condor_config.sun4x_56
condor_config.sgi_64
condor_config.platform -> condor_config.@sys
```

and the `LOCAL_CONFIG_FILE` configuration variable would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.platform, \
                    $(ETC)/$(HOSTNAME).local
```

Platform-Specific Configuration File Settings

The configuration variables that are truly platform-specific are:

RELEASEDIR Full path to the installed Condor binaries. While the configuration files may be shared among different platforms, the binaries certainly cannot. Therefore, maintain separate release directories for each platform in the pool. See section 3.3.3 on “Condor-wide Configuration File Entries” for details.

MAIL The full path to the mail program. See section 3.3.3 on “Condor-wide Configuration File Entries” for details.

CONSOLEDEVICES Which devices in `/dev` should be treated as console devices. See section 3.3.10 on “`condor_startd` Configuration File Entries” for details.

DAEMONLIST Which daemons the *condor_master* should start up. The reason this setting is platform-specific is to distinguish the *condor_kbdd*. On Alphas running Digital Unix, it was needed, and it is not needed on other platforms. See section 3.3.9 on for details.

Reasonable defaults for all of these configuration variables will be found in the default configuration files inside a given platform’s binary distribution (except the `RELEASE_DIR`, since the location of the Condor binaries and libraries is installation specific). With multiple platforms, use one of the `condor_config` files from either running *condor_configure* or from the `<release_dir>/etc/examples/condor_config.generic` file, take these settings out, save them into a platform-specific file, and install the resulting platform-independent file as the global configuration file. Then, find the same settings from the configuration files for any other platforms to be set up, and put them in their own platform-specific files. Finally, set the `LOCAL_CONFIG_FILE` configuration variable to point to the appropriate platform-specific file, as described above.

Not even all of these configuration variables are necessarily going to be different. For example, if an installed mail program understands the `-s` option in `/usr/local/bin/mail` on all platforms, the `MAIL` macro may be set to that in the global configuration file, and not define it anywhere else. For a pool with only Digital Unix, the `DAEMONLIST` will be the same for each, so there is no reason not to put that in the global configuration file.

Other Uses for Platform-Specific Configuration Files

It is certainly possible that an installation may want other configuration variables to be platform-specific as well. Perhaps a different policy is desired for one of the platforms. Perhaps different people should get the e-mail about problems with the different platforms. There is nothing hard-coded about any of this. What is shared and what should not shared is entirely configurable.

Since the `LOCAL_CONFIG_FILE` macro can be an arbitrary list of files, an installation can even break up the global, platform-independent settings into separate files. In fact, the global configuration file might only contain a definition for `LOCAL_CONFIG_FILE`, and all other configuration variables would be placed in separate files.

Different people may be given different permissions to change different Condor settings. For example, if a user is to be able to change certain settings, but nothing else, those settings may be placed in a file which was early in the `LOCAL_CONFIG_FILE` list, to give that user write permission on that file, then include all the other files after that one. In this way, if the user was trying to change settings she/he should not, they would simply be overridden.

This mechanism is quite flexible and powerful. For very specific configuration needs, they can probably be met by using file permissions, the `LOCAL_CONFIG_FILE` configuration variable, and imagination.

3.12.3 Full Installation of `condor_compile`

In order to take advantage of two major Condor features: checkpointing and remote system calls, users of the Condor system need to relink their binaries. Programs that are not relinked for Condor can run in Condor's "vanilla" universe just fine, however, they cannot checkpoint and migrate, or run on machines without a shared filesystem.

To relink your programs with Condor, we provide a special tool, `condor_compile`. As installed by default, `condor_compile` works with the following commands: `gcc`, `g++`, `g77`, `cc`, `acc`, `c89`, `CC`, `f77`, `fort77`, `ld`. On Solaris and Digital Unix, `f90` is also supported. See the `condor_compile(1)` man page for details on using `condor_compile`.

However, you can make `condor_compile` work transparently with all commands on your system whatsoever, including `make`.

The basic idea here is to replace the system linker (`ld`) with the Condor linker. Then, when a program is to be linked, the condor linker figures out whether this binary will be for Condor, or for a normal binary. If it is to be a normal compile, the old `ld` is called. If this binary is to be linked for condor, the script performs the necessary operations in order to prepare a binary that can be used with condor. In order to differentiate between normal builds and condor builds, the user simply places `condor_compile` before their build command, which sets the appropriate environment variable that lets the condor linker script know it needs to do its magic.

In order to perform this full installation of `condor_compile`, the following steps need to be taken:

1. Rename the system linker from `ld` to `ld.real`.
2. Copy the condor linker to the location of the previous `ld`.
3. Set the owner of the linker to root.
4. Set the permissions on the new linker to 755.

The actual commands that you must execute depend upon the system that you are on. The location of the system linker (`ld`), is as follows:

Operating System	Location of <code>ld</code> (<code>ld-path</code>)
Linux	<code>/usr/bin</code>
Solaris 2.X	<code>/usr/ccs/bin</code>
OSF/1 (Digital Unix)	<code>/usr/lib/cmplrs/cc</code>

On these platforms, issue the following commands (as root), where `ld-path` is replaced by the path to your system's `ld`.

```
mv [[ld-path]]/ld [[ld-path]]/ld.real
cp /usr/local/condor/lib/ld [[ld-path]]/ld
chown root [[ld-path]]/ld
chmod 755 [[ld-path]]/ld
```

If you remove Condor from your system latter on, linking will continue to work, since the condor linker will always default to compiling normal binaries and simply call the real ld. In the interest of simplicity, it is recommended that you reverse the above changes by moving your ld.real linker back to it's former position as ld, overwriting the condor linker.

NOTE: If you ever upgrade your operating system after performing a full installation of *condor_compile*, you will probably have to re-do all the steps outlined above. Generally speaking, new versions or patches of an operating system might replace the system ld binary, which would undo the full installation of *condor_compile*.

3.12.4 The *condor_kbdd*

The Condor keyboard daemon (*condor_kbdd*) monitors X events on machines where the operating system does not provide a way of monitoring the idle time of the keyboard or mouse. It is not needed for most platforms, as Condor has other ways of detecting keyboard and mouse activity.

Although great measures have been taken to make this daemon as robust as possible, the X window system was not designed to facilitate such a need, and thus is less then optimal on machines where many users log in and out on the console frequently.

In order to work with X authority, the system by which X authorizes processes to connect to X servers, the *condor_kbdd* needs to run with super user privileges. Currently, the daemon assumes that X uses the HOME environment variable in order to locate a file named *.Xauthority*, which contains keys necessary to connect to an X server. The keyboard daemon attempts to set this environment variable to various users home directories in order to gain a connection to the X server and monitor events. This may fail to work on your system, if you are using a non-standard approach. If the keyboard daemon is not allowed to attach to the X server, the state of a machine may be incorrectly set to idle when a user is, in fact, using the machine.

In some environments, the *condor_kbdd* will not be able to connect to the X server because the user currently logged into the system keeps their authentication token for using the X server in a place that no local user on the current machine can get to. This may be the case for AFS where the user's *.Xauthority* file is in an AFS home directory. There may also be cases where the *condor_kbdd* may not be run with super user privileges because of political reasons, but it is still desired to be able to monitor X activity. In these cases, change the XDM configuration in order to start up the *condor_kbdd* with the permissions of the currently logging in user. Although your situation may differ, if you are running X11R6.3, you will probably want to edit the files in */usr/X11R6/lib/X11/xdm*. The *.xsession* file should have the keyboard daemon start up at the end, and the *.Xreset* file should have the keyboard daemon shut down. The **-l** option can be used to write the daemon's log file to a place where the user running the daemon has permission to write a file. We recommend something akin to *\$HOME/.kbdd.log*, since this is a place where every user can write, and it will not get in the way. The **-pidfile** and **-k** options allow for easy shut down of the daemon by storing the process id in a file. It will be necessary to add lines to the XDM configuration that look something like:

```
condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the *condor_kbdd* as the user who is currently logging in and write the log to a file in the directory `$HOME/.kbdd.log/`. Also, this will save the process id of the daemon to `~/ .kbdd.pid`, so that when the user logs out, XDM can do:

```
condor_kbdd -k $HOME/.kbdd.pid
```

This will shut down the process recorded in `~/ .kbdd.pid` and exit.

To see how well the keyboard daemon is working, review the log for the daemon and look for successful connections to the X server. If there are none, the *condor_kbdd* is unable to connect to the machine's X server.

3.12.5 Configuring The CondorView Server

The CondorView server is an alternate use of the *condor_collector* that logs information on disk, providing a persistent, historical database of pool state. This includes machine state, as well as the state of jobs submitted by users.

An existing *condor_collector* may act as the CondorView collector through configuration. This is the simplest situation, because the only change needed is to turn on the logging of historical information. The alternative of configuring a new *condor_collector* to act as the CondorView collector is slightly more complicated, while it offers the advantage that the same CondorView collector may be used for several pools as desired, to aggregate information into one place.

The following sections describe how to configure a machine to run a CondorView server and to configure a pool to send updates to it.

Configuring a Machine to be a CondorView Server

To configure the CondorView collector, a few configuration variables are added or modified for the *condor_collector* chosen to act as the CondorView collector. These configuration variables are described in section 3.3.16 on page 208. Here are brief explanations of the entries that must be customized:

POOL_HISTORY_DIR The directory where historical data will be stored. This directory must be writable by whatever user the CondorView collector is running as (usually the user *condor*). There is a configurable limit to the maximum space required for all the files created by the CondorView server called (`POOL_HISTORY_MAX_STORAGE`).

NOTE: This directory should be separate and different from the *spool* or *log* directories already set up for Condor. There are a few problems putting these files into either of those directories.

KEEP_POOL_HISTORY A boolean value that determines if the CondorView collector should store the historical information. It is `False` by default, and must be specified as `True` in the local configuration file to enable data collection.

Once these settings are in place in the configuration file for the CondorView server host, create the directory specified in `POOL_HISTORY_DIR` and make it writable by the user the CondorView collector is running as. This is the same user that owns the `CollectorLog` file in the `log` directory. The user is usually `condor`.

If using the existing *condor_collector* as the CondorView collector, no further configuration is needed. To run a different *condor_collector* to act as the CondorView collector, configure Condor to automatically start it.

If using a separate host for the CondorView collector, to start it, add the value `COLLECTOR` to `DAEMON_LIST`, and restart Condor on that host. To run the CondorView collector on the same host as another *condor_collector*, ensure that the two *condor_collector* daemons use different network ports. Here is an example configuration in which the main *condor_collector* and the CondorView collector are started up by the same *condor_master* daemon on the same machine. In this example, the CondorView collector uses port 12345.

```
VIEW_SERVER = $(COLLECTOR)
VIEW_SERVER_ARGS = -f -p 12345
VIEW_SERVER_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog"
DAEMON_LIST = MASTER, NEGOTIATOR, COLLECTOR, VIEW_SERVER
```

For this change to take effect, restart the *condor_master* on this host. This may be accomplished with the *condor_restart* command, if the command is run with administrator access to the pool.

Configuring a Pool to Report to the CondorView Server

For the CondorView server to function, configure the existing collector to forward ClassAd updates to it. This configuration is only necessary if the CondorView collector is a different collector from the existing *condor_collector* for the pool. All the Condor daemons in the pool send their ClassAd updates to the regular *condor_collector*, which in turn will forward them on to the CondorView server.

Define the following configuration variable:

```
CONDOR_VIEW_HOST = full.hostname[:portnumber]
```

where `full.hostname` is the full host name of the machine running the CondorView collector. The full host name is optionally followed by a colon and port number. This is only necessary if the CondorView collector is configured to use a port number other than the default.

Place this setting in the configuration file used by the existing *condor_collector*. It is acceptable to place it in the global configuration file. The CondorView collector will ignore this setting (as it should) as it notices that it is being asked to forward ClassAds to itself.

Once the CondorView server is running with this change, send a *condor_reconfig* command to the main *condor_collector* for the change to take effect, so it will begin forwarding updates. A query to the CondorView collector will verify that it is working. A query example:

```
condor_status -pool condor.view.host[:portnumber]
```

3.12.6 Running Condor Jobs within a VMware or Xen Virtual Machine Environment

Condor jobs are formed from executables that are compiled to execute on specific platforms. This in turn restricts the machines within a Condor pool where a job may be executed. A Condor job may now be executed on a virtual machine system running VMware or Xen. This allows Windows executables to run on a Linux machine, and Linux executables to run on a Windows machine. These virtual machine systems exist for the Intel x86 architecture.

In older versions of Condor, other parts of the system were also referred to as *virtual machines*, but in all cases, those are now known as *slots*. A virtual machine here describes the environment in which the outside operating system (called the host) emulates an inner operating system (called the inner virtual machine), such that an executable appears to run directly on the inner virtual machine. In other parts of Condor, a *slot* (formerly known as *virtual machine*) refers to the multiple CPUs of an SMP machine. Also, be careful not to confuse the virtual machines discussed here with the Java Virtual Machine (JVM) referenced in other parts of this manual.

Under Xen or VMware, Condor has the flexibility to run a job on either the host or the inner virtual machine, hence two platforms appear to exist on a single machine. Since two platforms are an illusion, Condor understands the illusion, allowing a Condor job to be executed on only one at a time.

Installation and Configuration

Condor must be separately installed, separately configured, and separately running on both the host and the inner virtual machine.

The configuration for the host specifies `VMP_VMLIST`. This specifies host names or IP addresses of all inner virtual machines running on this host. An example configuration on the host machine:

```
VMP_VM_LIST = vmware1.domain.com, vmware2.domain.com
```

The configuration for each separate inner virtual machine specifies `VMP_HOST_MACHINE`. This specifies the host for the inner virtual machine. An example configuration on an inner virtual machine:

```
VMP_HOST_MACHINE = host.domain.com
```

Given this configuration, as well as communication between Condor daemons running on the host and on the inner virtual machine, the policy for when jobs may execute is set by Condor. While

the host is executing a Condor job, the `START` policy on the inner virtual machine is overridden with `False`, so no Condor jobs will be started on the inner virtual machine. Conversely, while the inner virtual machine is executing a Condor job, the `START` policy on the host is overridden with `False`, so no Condor jobs will be started on the host.

The inner virtual machine is further provided with a new syntax for referring to the machine ClassAd attributes of its host. Any machine ClassAd attribute with a prefix of the string `HOST_` explicitly refers to the host's ClassAd attributes. The `START` policy on the inner virtual machine ought to use this syntax to avoid starting jobs when its host is too busy processing other items. An example configuration for `START` on an inner virtual machine:

```
START = ( (KeyboardIdle > 150 ) && ( HOST_KeyboardIdle > 150 ) \
        && ( LoadAvg <= 0.3 ) && ( HOST_TotalLoadAvg <= 0.3 ) )
```

3.12.7 Configuring The Startd for SMP Machines

This section describes how to configure the *condor_startd* for SMP (Symmetric Multi-Processor) machines. Machines with more than one CPU may be configured to run more than one job at a time. As always, owners of the resources have great flexibility in defining the policy under which multiple jobs may run, suspend, vacate, etc.

How Shared Resources are Represented to Condor

The way SMP machines are represented to the Condor system is that the shared resources are broken up into individual *slots*. Each slot can be matched and claimed by users. Each slot is represented by an individual ClassAd (see the ClassAd reference, section 4.1, for details). In this way, each SMP machine will appear to the Condor system as a collection of separate slots. As an example, an SMP machine named `vulture.cs.wisc.edu` would appear to Condor as the multiple machines, named `slot1@vulture.cs.wisc.edu`, `slot2@vulture.cs.wisc.edu`, `slot3@vulture.cs.wisc.edu`, and so on.

The way that the *condor_startd* breaks up the shared system resources into the different slots is configurable. All shared system resources (like RAM, disk space, swap space, etc.) can either be divided evenly among all the slots, with each CPU getting its own slot, or you can define your own *slot types*, so that resources can be unevenly partitioned. Regardless of the partitioning scheme used, it is important to remember the goal is to create a representative slot ClassAd, to be used for matchmaking with jobs. Condor does not directly enforce slot shared resource allocations, and jobs are free to oversubscribe to shared resources.

Consider an example where two slots are each defined with 50% of available RAM. The resultant ClassAd for each slot will advertise one half the available RAM. Users may submit jobs with RAM requirements that match these slots. However, jobs run on either slot are free to consume more than 50% of available RAM. Condor will not directly enforce a RAM utilization limit on either slot. If a shared resource enforcement capability is needed, it is possible to write a Startd policy that will evict a job that oversubscribes to shared resources, see section 3.12.7.

The following section gives details on how to configure Condor to divide the resources on an SMP machine into separate slots.

Dividing System Resources in SMP Machines

This section describes the settings that allow you to define your own slot types and to control how many slots of each type are reported to Condor.

There are two main ways to go about partitioning an SMP machine:

Define your own slot types. By defining your own types, you can specify what fraction of shared system resources (CPU, RAM, swap space and disk space) go to each slot. Once you define your own types, you can control how many of each type are reported at any given time.

Evenly divide all resources. If you do not define your own types, the *condor_startd* will automatically partition your machine into slots for you. It will do so by placing a single CPU in each slot, and evenly dividing all shared resources among the slots. With this default partitioning, you only specify how many slots are reported at a time. By default, all slots are reported to Condor.

The number of each type being reported can be changed at run-time, by issuing a reconfiguration command to the *condor_startd* daemon (sending a SIGHUP or using *condor_reconfig*). However, the definitions for the types themselves cannot be changed with reconfiguration. If you change any slot type definitions, you must use *condor_restart*

```
condor_restart -startd
```

for that change to take effect.

Defining Slot Types

To define your own slot types, add configuration file parameters that list how much of each system resource you want in the given slot type. Do this by defining configuration variables of the form `SLOT_TYPE_<N>`. The `<N>` represents an integer (for example, `SLOT_TYPE_1`), which specifies the slot type defined. Note that there may be multiple slots of each type. The number created is configured with `NUM_SLOTS_TYPE_<N>` as described later in this section.

A type describes what share of the total system resources a given slot has available to it.

The type can be defined by:

- A simple fraction, such as 1/4
- A simple percentage, such as 25%

- A comma-separated list of attributes, with a percentage, fraction, numerical value, or `auto` for each one.
- A comma-separated list including a blanket value that serves as a default for any resources not explicitly specified in the list.

A simple fraction or percentage causes an allocation of the total system resources. This includes the number of CPUs. A comma-separated list allows a fine-tuning of the amounts for specific attributes.

The attributes that specify the number of CPUs and the total amount of RAM in the SMP machine do not change. For these attributes, specify either absolute values or percentages of the total available amount (or `auto`). For example, in a machine with 128 Mbytes of RAM, all the following definitions result in the same allocation amount.

```
mem=64
mem=1 / 2
mem=50%
mem=auto
```

Other attributes are dynamic, such as disk space and swap space. For these, specify a percentage or fraction of the total value that is allocated to each slot, instead of specifying absolute values. As the total values of these resources change on your machine, each slot will take its fraction of the total and report that as its available amount.

The disk space allocated to each slot is taken from the disk partition containing the slots execute directory (configured with `EXECUTE` or `SLOTx_EXECUTE`). If every slot is in a different partition, then each one may be defined with up to 100% for its disk share. If some slots are in the same partition, then their total is not allowed to exceed 100%.

The four attribute names are case insensitive when defining slot types. The first letter of the attribute name distinguishes between the attributes. The four attributes, with several examples of acceptable names for each are

- Cpus, C, c, cpu
- ram, RAM, MEMORY, memory, Mem, R, r, M, m
- disk, Disk, D, d
- swap, SWAP, S, s, VirtualMemory, V, v

As an example, consider a host of 4 CPUs and 256 megs of RAM. Here are valid example slot type definitions. Types 1-3 are all equivalent to each other, as are types 4-6. Note that in a real configuration, you would not use all of these slot types together because they add up to more than 100% of the various system resources. Also note that in a real configuration, you would need to also define `NUM_SLOTS_TYPE_<N>` for each slot type.

```

SLOT_TYPE_1 = cpus=2, ram=128, swap=25%, disk=1/2

SLOT_TYPE_2 = cpus=1/2, memory=128, virt=25%, disk=50%

SLOT_TYPE_3 = c=1/2, m=50%, v=1/4, disk=1/2

SLOT_TYPE_4 = c=25%, m=64, v=1/4, d=25%

SLOT_TYPE_5 = 25%

SLOT_TYPE_6 = 1/4

```

The default value for each resource share is `auto`. The share may also be explicitly set to `auto`. All slots with the value `auto` for a given type of resource will evenly divide whatever remains after subtracting out whatever was explicitly allocated in other slot definitions. For example, if one slot is defined to use 10% of the memory and the rest define it as `auto` (or leave it undefined), then the rest of the slots will evenly divide 90% of the memory between themselves.

In both of the following examples, the disk share is set to `auto`, `cpus` is 1, and everything else is 50%:

```

SLOT_TYPE_1 = cpus=1, ram=1/2, swap=50%

SLOT_TYPE_1 = cpus=1, disk=auto, 50%

```

The number of slots of each type is set with the configuration variable `NUM_SLOTS_TYPE_<N>`, where `N` is the type as given in the `SLOT_TYPE_<N>` variable.

Note that it is possible to set the configuration variables such that they specify an impossible configuration. If this occurs, the *condor_startd* daemon fails after writing a message to its log attempting to indicate the configuration requirements that it could not implement.

Evenly Divided Resources

If you are not defining your own slot types, then all resources are divided equally among the slots. The number of slots within the SMP machine is the only attribute that needs to be defined. Its definition is accomplished by setting the configuration variable `NUM_SLOTS` to the integer number of slots desired. If variable `NUM_SLOTS` is not defined, it defaults to the number of CPUs within the SMP machine. You cannot use `NUM_SLOTS` to make Condor advertise more slots than there are CPUs on the machine. To do that, use `NUM_CPUS`.

Configuring Startd Policy for SMP Machines

Section 3.5 details the Startd Policy Configuration. This section continues the discussion with respect to SMP machines.

Each slot within an SMP machine is treated as an independent machine, each with its own view of its machine state. There is a single set of policy expressions for the SMP machine as a whole. This policy may consider the slot state(s) in its expressions. This makes some policies easy to set, but it makes other policies difficult or impossible to set.

An easy policy to set configures how many of the slots notice console or tty activity on the SMP as a whole. Slots that are not configured to notice any activity will report ConsoleIdle and KeyboardIdle times from when the *condor_startd* daemon was started, (plus a configurable number of seconds). With this, you can set up a multiple CPU machine with the default policy settings plus add that the keyboard and console noticed by only one slot. Assuming a reasonable load average (see section 3.12.7 below on “Load Average for SMP Machines”), only the one slot will suspend or vacate its job when the owner starts typing at their machine again. The rest of the slots could be matched with jobs and leave them running, even while the user was interactively using the machine. If the default policy is used, all slots notice tty and console activity and currently running jobs would suspend or preempt.

This example policy is controlled with the following configuration variables.

- SLOTS_CONNECTED_TO_CONSOLE
- SLOTS_CONNECTED_TO_KEYBOARD
- DISCONNECTED_KEYBOARD_IDLE_BOOST

These configuration variables are fully described in section 3.3.10 on page 179 which lists all the configuration file settings for the *condor_startd*.

The configuration of slots allows each slot to advertise its own machine ClassAd. Yet, there is only one set of policy expressions for the SMP machine as a whole. This makes the implementation of certain types of policies impossible. While evaluating the state of one slot (within the SMP machine), the state of other slots (again within the SMP machine) are not available. Decisions for one slot cannot be based on what other machines within the SMP are doing.

Specifically, the evaluation of a slot policy expression works in the following way.

1. The configuration file specifies policy expressions that are shared among all of the slots on the SMP machine.
2. Each slot reads the configuration file and sets up its own machine ClassAd.
3. Each slot is now separate from the others. It has a different state, a different machine ClassAd, and if there is a job running, a separate job ad. Each slot periodically evaluates the policy expressions, changing its own state as necessary. This occurs independently of the other slots

on the machine. So, if the *condor_startd* daemon is evaluating a policy expression on a specific slot, and the policy expression refers to `ProcID`, `Owner`, or any attribute from a job ad, it *always* refers to the `ClassAd` of the job running on the specific slot.

To set a different policy for the slots within an SMP machine, a (SUSPEND) policy will be of the form

```
SUSPEND = ( (SlotID == 1) && (PolicyForSlot1) ) || \
           ( (SlotID == 2) && (PolicyForSlot2) )
```

where `(PolicyForSlot1)` and `(PolicyForSlot2)` are the desired expressions for each slot.

Load Average for SMP Machines

Most operating systems define the load average for an SMP machine as the total load on all CPUs. For example, if you have a 4-CPU machine with 3 CPU-bound processes running at the same time, the load would be 3.0. In Condor, we maintain this view of the total load average and publish it in all resource `ClassAds` as `TotalLoadAvg`.

Condor also provides a per-CPU load average for SMP machines. This nicely represents the model that each node on an SMP is a slot, separate from the other nodes. All of the default, single-CPU policy expressions can be used directly on SMP machines, without modification, since the `LoadAvg` and `CondorLoadAvg` attributes are the per-slot versions, not the total, SMP-wide versions.

The per-CPU load average on SMP machines is a Condor invention. No system call exists to ask the operating system for this value. Condor already computes the load average generated by Condor on each slot. It does this by close monitoring of all processes spawned by any of the Condor daemons, even ones that are orphaned and then inherited by *init*. This Condor load average per slot is reported as the attribute `CondorLoadAvg` in all resource `ClassAds`, and the total Condor load average for the entire machine is reported as `TotalCondorLoadAvg`. The total, system-wide load average for the entire machine is reported as `TotalLoadAvg`. Basically, Condor walks through all the slots and assigns out portions of the total load average to each one. First, Condor assigns the known Condor load average to each node that is generating load. If there's any load average left in the total system load, it is considered an owner load. Any slots Condor believes are in the Owner state (like ones that have keyboard activity), are the first to get assigned this owner load. Condor hands out owner load in increments of at most 1.0, so generally speaking, no slot has a load average above 1.0. If Condor runs out of total load average before it runs out of virtual machines, all the remaining machines believe that they have no load average at all. If, instead, Condor runs out of slots and it still has owner load remaining, Condor starts assigning that load to Condor nodes as well, giving individual nodes with a load average higher than 1.0.

Debug logging in the SMP Startd

This section describes how the *condor_startd* daemon handles its debugging messages for SMP machines. In general, a given log message will either be something that is machine-wide (like reporting the total system load average), or it will be specific to a given slot. Any log entries specific to a slot have an extra header printed out in the entry: `slot#:`. So, for example, here's the output about system resources that are being gathered (with `D_FULLDEBUG` and `D_LOAD` turned on) on a 2-CPU machine with no Condor activity, and the keyboard connected to both slots:

```
11/25 18:15 Swap space: 131064
11/25 18:15 number of Kbytes available for (/home/condor/execute): 1345063
11/25 18:15 Looking up RESERVED_DISK parameter
11/25 18:15 Reserving 5120 Kbytes for file system
11/25 18:15 Disk space: 1339943
11/25 18:15 Load avg: 0.340000 0.800000 1.170000
11/25 18:15 Idle Time: user= 0 , console= 4 seconds
11/25 18:15 SystemLoad: 0.340 TotalCondorLoad: 0.000 TotalOwnerLoad: 0.340
11/25 18:15 slot1: Idle time: Keyboard: 0 Console: 4
11/25 18:15 slot1: SystemLoad: 0.340 CondorLoad: 0.000 OwnerLoad: 0.340
11/25 18:15 slot2: Idle time: Keyboard: 0 Console: 4
11/25 18:15 slot2: SystemLoad: 0.000 CondorLoad: 0.000 OwnerLoad: 0.000
11/25 18:15 slot1: State: Owner Activity: Idle
11/25 18:15 slot2: State: Owner Activity: Idle
```

If, on the other hand, this machine only had one slot connected to the keyboard and console, and the other slot was running a job, it might look something like this:

```
11/25 18:19 Load avg: 1.250000 0.910000 1.090000
11/25 18:19 Idle Time: user= 0 , console= 0 seconds
11/25 18:19 SystemLoad: 1.250 TotalCondorLoad: 0.996 TotalOwnerLoad: 0.254
11/25 18:19 slot1: Idle time: Keyboard: 0 Console: 0
11/25 18:19 slot1: SystemLoad: 0.254 CondorLoad: 0.000 OwnerLoad: 0.254
11/25 18:19 slot2: Idle time: Keyboard: 1496 Console: 1496
11/25 18:19 slot2: SystemLoad: 0.996 CondorLoad: 0.996 OwnerLoad: 0.000
11/25 18:19 slot1: State: Owner Activity: Idle
11/25 18:19 slot2: State: Claimed Activity: Busy
```

As you can see, shared system resources are printed without the header (like total swap space), and slot-specific messages (like the load average or state of each slot) get the special header appended.

Configuring STARTD_ATTRS on a per-slot basis

The `STARTD_ATTRS` (and legacy `STARTD_EXPRS`) settings can be configured on a per-slot basis. The *condor_startd* daemon builds the list of items to advertise by combining the lists in this order:

1. `STARTD_ATTRS`
2. `STARTD_EXPRS`
3. `SLOTx_STARTD_ATTRS`
4. `SLOTx_STARTD_EXPRS`

For example, consider the following configuration:

```
STARTD_ATTRS = favorite_color, favorite_season
SLOT1_STARTD_ATTRS = favorite_movie
SLOT2_STARTD_ATTRS = favorite_song
```

This will result in the *condor_startd* ClassAd for slot1 defining values for `favorite_color`, `favorite_season`, and `favorite_movie`. slot2 will have values for `favorite_color`, `favorite_season`, and `favorite_song`.

Attributes themselves in the `STARTD_ATTRS` list can also be defined on a per-slot basis. Here is another example:

```
favorite_color = "blue"
favorite_season = "spring"
STARTD_ATTRS = favorite_color, favorite_season
SLOT2_favorite_color = "green"
SLOT3_favorite_season = "summer"
```

For this example, the *condor_startd* ClassAds are

slot1:

```
favorite_color = "blue"
favorite_season = "spring"
```

slot2:

```
favorite_color = "green"
favorite_season = "spring"
```

slot3:

```
favorite_color = "blue"
favorite_season = "summer"
```

Dynamic *condor_startd* Provisioning: Dynamic Slots

Dynamic provisioning, also referred to as a partitionable *condor_startd* or as dynamic slots, allows users to mark slots as partitionable. This means that more than one job can occupy a single slot at any one time. Typically, slots have a fixed set of resources, including the CPUs, memory and disk space. By partitioning the slot, these resources become more flexible and able to be better utilized.

Dynamic provisioning provides powerful configuration possibilities, and so should be used with care. Specifically, while preemption occurs for each individual dynamic slot, it cannot occur directly for the partitionable slot, or for groups of dynamic slots. For example, for a large number of jobs requiring 1GB of memory, a pool might be split up into 1GB dynamic slots. In this instance a job requiring 2GB of memory will be starved and unable to run.

Here is an example that demonstrates how more than one job can be matched to a single slot using dynamic provisioning. In this example, slot1 has the following resources:

```
cpu=10
memory=10240
disk=BIG
```

Assume that JobA is allocated to this slot. JobA includes the following requirements:

```
cpu=3
memory=1024
disk=10240
```

The portion of the slot that is utilized is referred to as Slot1.1, and after allocation, the slot advertises that it has the following resources still available:

```
cpu=7
memory=9216
disk=BIG-10240
```

As each new job is allocated to Slot1, it breaks into Slot1.1, Slot1.2, etc., until the entire set of available resources have been consumed by jobs.

To enable dynamic provisioning, set the `SLOT_TYPE_<N>_PARTITIONABLE` configuration variable to `True`. The string `N` within the configuration variable name is the slot number.

In a pool using dynamic provisioning, jobs can have extra, and desired, resources specified in the submit description file:

```
request_cpus  
request_memory  
request_disk (in kilobytes)
```

This example shows a portion of the job submit description file for use when submitting a job to a pool with dynamic provisioning.

```
universe = vanilla  
executable =  
  
request_cpus = 3  
request_memory = 1024  
request_disk = 10240  
  
queue
```

For each type of slot, the original, partitionable slot and the new smaller, dynamic slots, an attribute is added to identify it. The original slot will have an attribute stating

```
PartitionableSlot = True
```

and the dynamic slots will have an attribute

```
DynamicSlot = True
```

These attributes may be used in a START expression for the purposes of creating detailed policies.

A partitionable slot will always appear as though it is not running a job. It will eventually show as having no available resources, which will prevent further matching to new jobs. Because it has been effectively broken up into smaller slots, these will show as running jobs directly. These dynamic slots can also be preempted in the same way as nonpartitioned slots.

3.12.8 Condor's Dedicated Scheduling

Applications that require multiple resources, yet must not be preempted, are handled gracefully by Condor. Condor combines *opportunistic scheduling* and *dedicated scheduling* within a single system. *Opportunistic scheduling* involves placing a job on a non-dedicated resource under the assumption that the resource may not be available for the entire duration of the job. *Dedicated scheduling* assumes the constant availability of resources; it is assumed that the job will run to completion, without interruption.

To support applications needing dedicated resources, an administrator configures resources to be dedicated. These resources are controlled by a dedicated scheduler, a single machine within the

pool that runs a *condor_schedd* daemon. There is no limit on the number of dedicated schedulers within a Condor pool. However, each dedicated resource may only be managed by a single dedicated scheduler. Running multiple dedicated schedulers within a single pool results in a fragmentation of dedicated resources. This can create a situation where jobs cannot run, because there are too few resource that may be allocated.

After a *condor_schedd* daemon has been selected as the dedicated scheduler for the pool and resources are configured to be dedicated, users submit parallel universe jobs (including MPI applications) through that *condor_schedd* daemon. When an idle parallel universe job is found in the queue, this dedicated scheduler performs its own scheduling algorithm to find and claim appropriate resources for the job. When a resource can no longer be used to serve a job that must not be preempted, the resource is allowed to run opportunistic jobs.

Selecting and Setting Up a Dedicated Scheduler

We recommend that you select a single machine within a Condor pool to act as the dedicated scheduler. This becomes the machine from upon which all users submit their parallel universe jobs. The perfect choice for the dedicated scheduler is the single, front-end machine for a dedicated cluster of compute nodes. For the pool without an obvious choice for a submit machine, choose a machine that all users can log into, as well as one that is likely to be up and running all the time. All of Condor's other resource requirements for a submit machine apply to this machine, such as having enough disk space in the spool directory to hold jobs. See section 3.2.2 on page 118 for details on these issues.

Configuration Examples for Dedicated Resources

Each machine may have its own policy for the execution of jobs. This policy is set by configuration. Each machine with aspects of its configuration that are dedicated identifies the dedicated scheduler. And, the ClassAd representing a job to be executed on one or more of these dedicated machines includes an identifying attribute. An example configuration file with the following various policy settings is `/etc/condor_config.local.dedicated.resource`.

Each dedicated machine defines the configuration variable `DedicatedScheduler`, which identifies the dedicated scheduler it is managed by. The local configuration file for any dedicated resource contains a modified form of

```
DedicatedScheduler = "DedicatedScheduler@full.host.name"
STARTD_ATTRS = $(STARTD_ATTRS), DedicatedScheduler
```

Substitute the host name of the dedicated scheduler machine for the string `"full.host.name"`.

If running personal Condor, the name of the scheduler includes the user name it was started as, so the configuration appears as:

```
DedicatedScheduler = "DedicatedScheduler@username@full.host.name"
STARTD_ATTRS = $(STARTD_ATTRS), DedicatedScheduler
```

All dedicated resources must have policy expressions which allow for jobs to always run, but not be preempted. The resource must also be configured to prefer jobs from the dedicated scheduler over all other jobs. Therefore, configuration gives the dedicated scheduler of choice the highest rank. It is worth noting that Condor puts no other requirements on a resource for it to be considered dedicated.

Job ClassAds from the dedicated scheduler contain the attribute `Scheduler`. The attribute is defined by a string of the form

```
Scheduler = "DedicatedScheduler@full.host.name"
```

The host name of the dedicated scheduler substitutes for the string `full.host.name`.

Different resources in the pool may have different dedicated policies by varying the local configuration.

Policy Scenario: Machine Runs Only Jobs That Require Dedicated Resources One possible scenario for the use of a dedicated resource is to only run jobs that require the dedicated resource. To enact this policy, the configure with the following expressions:

```
START      = Scheduler == $(DedicatedScheduler)
SUSPEND    = False
CONTINUE   = True
PREEMPT    = False
KILL       = False
WANT_SUSPEND = False
WANT_VACATE = False
RANK       = Scheduler == $(DedicatedScheduler)
```

The `START` expression specifies that a job with the `Scheduler` attribute must match the string corresponding `DedicatedScheduler` attribute in the machine ClassAd. The `RANK` expression specifies that this same job (with the `Scheduler` attribute) has the highest rank. This prevents other jobs from preempting it based on user priorities. The rest of the expressions disable all of the *condor_startd* daemon's regular policies for evicting jobs when keyboard and CPU activity is discovered on the machine.

Policy Scenario: Run Both Jobs That Do and Do Not Require Dedicated Resources While the first example works nicely for jobs requiring dedicated resources, it can lead to poor utilization of the dedicated machines. A more sophisticated strategy allows the machines to run other jobs, when no jobs that require dedicated resources exist. The machine is configured to prefer jobs that require dedicated resources, but not prevent others from running.

To implement this, configure the machine as a dedicated resource (as above) modifying only the `START` expression:

```
START = True
```

Policy Scenario: Adding Desk-Top Resources To The Mix A third policy example allows all jobs. These desk-top machines use a preexisting `START` expression that takes the machine owner's usage into account for some jobs. The machine does not preempt jobs that must run on dedicated resources, while it will preempt other jobs based on a previously set policy. So, the default pool policy is used for starting and stopping jobs, while jobs that require a dedicated resource always start and are not preempted.

The `START`, `SUSPEND`, `PREEMPT`, and `RANK` policies are set in the global configuration. Locally, the configuration is modified to this hybrid policy by adding a second case.

```
SUSPEND      = Scheduler != $(DedicatedScheduler) && ($(SUSPEND))
PREEMPT      = Scheduler != $(DedicatedScheduler) && ($(PREEMPT))
RANK_FACTOR   = 1000000
RANK         = (Scheduler == $(DedicatedScheduler) * $(RANK_FACTOR)) \
               + $(RANK)
START        = (Scheduler == $(DedicatedScheduler)) || ($(START))
```

Define `RANK_FACTOR` to be a larger value than the maximum value possible for the existing rank expression. `RANK` is just a floating point value, so there is no harm in having a value that is very large.

Policy Scenario: Parallel Scheduling Groups In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines – that is, all the nodes of a parallel job should be allocated to machines within the same group. The most common example is a pool of machines using infiniband switches. Each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the infiniband switches are not routed to each other, each job must run on machines connected to the same switch.

The dedicated scheduler's parallel scheduling groups features supports jobs that must not cross group boundaries. Define a group by having each machine within a group set the configuration variable `ParallelSchedulingGroup` with a string that is a unique name for the group. The submit description file for a parallel universe job which must not cross group boundaries contains

```
+WantParallelSchedulingGroups = True
```

The dedicated scheduler enforces the allocation to within a group.

Preemption with Dedicated Jobs

The dedicated scheduler can optionally preempt running MPI jobs in favor of higher priority MPI jobs in its queue. Note that this is different from preemption in non-parallel universes, and MPI jobs cannot be preempted either by a machine's user pressing a key or by other means.

By default, the dedicated scheduler will never preempt running MPI jobs. Two configuration file items control dedicated preemption: `SCHEDD_PREEMPTION_REQUIREMENTS` and

`SCHEDD_PREEMPTION_RANK` . These have no default value, so if either are not defined, preemption will never occur. `SCHEDD_PREEMPTION_REQUIREMENTS` must evaluate to `True` for a machine to be a candidate for this kind of preemption. If more machines are candidates for preemption than needed to satisfy a higher priority job, the machines are sorted by `SCHEDD_PREEMPTION_RANK`, and only the highest ranked machines are taken.

Note that preempting one node of a running MPI job requires killing the entire job on all of its nodes. So, when preemption happens, it may end up freeing more machines than strictly speaking are needed. Also, as Condor cannot produce checkpoints for MPI jobs, preempted jobs will be re-run, starting again from the beginning. Thus, the administrator should be careful when enabling dedicated preemption. The following example shows how to enable dedicated preemption.

```
STARTD_JOB_EXPRS = JobPrio
SCHEDD_PREEMPTION_REQUIREMENTS = (My.JobPrio < Target.JobPrio)
SCHEDD_PREEMPTION_RANK = 0.0
```

In this case, preemption is enabled by the user job priority. If a set of machines is running a job at user priority 5, and the user submits a new job at user priority 10, the running job will be preempted for the new job. The old job is put back in the queue, and will begin again from the beginning when assigned to a new set of machines.

Grouping dedicated nodes into parallel scheduling groups

In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines – that is, all the nodes of a parallel job should be allocated to machines in the same group. The most common example is a pool of machine using infiniband switches. Each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the infiniband switches are not routed to each other, each job must run on machines connected to the same switch. The dedicated scheduler's parallel scheduling groups features supports this operation.

Each `startd` must define which group it belongs to by setting the `ParallelSchedulingGroup` property in the config file, and advertising it into the machine `ClassAd`. The value of this property is simply a string, which should be the same for all `startds` in a given group. The property must be advertised in the `startd` job ad by appending `ParallelSchedulingGroup` into the `STARTD_ATTRS` configuration variable. Then, parallel jobs which want to be scheduled by group, declare this in their submit file by setting `+WantParallelSchedulingGroups=True`.

3.12.9 Configuring Condor for Running Backfill Jobs

Condor can be configured to run backfill jobs whenever the *condor_startd* has no other work to perform. These jobs are considered the lowest possible priority, but when machines would otherwise be idle, the resources can be put to good use.

Currently, Condor only supports using the Berkeley Open Infrastructure for Network Computing (BOINC) to provide the backfill jobs. More information about BOINC is available at <http://boinc.berkeley.edu>.

The rest of this section provides an overview of how backfill jobs work in Condor, details for configuring the policy for when backfill jobs are started or killed, and details on how to configure Condor to spawn the BOINC client to perform the work.

Overview of Backfill jobs in Condor

Whenever a resource controlled by Condor is in the Unclaimed/Idle state, it is totally idle; neither the interactive user nor a Condor job is performing any work. Machines in this state can be configured to enter the *Backfill* state, which allows the resource to attempt a background computation to keep itself busy until other work arrives (either a user returning to use the machine interactively, or a normal Condor job). Once a resource enters the Backfill state, the *condor_startd* will attempt to spawn another program, called a *backfill client*, to launch and manage the backfill computation. When other work arrives, the *condor_startd* will kill the backfill client and clean up any processes it has spawned, freeing the machine resources for the new, higher priority task. More details about the different states a Condor resource can enter and all of the possible transitions between them are described in section 3.5 beginning on page 252, especially sections 3.5.5, 3.5.6, and 3.5.7.

At this point, the only backfill system supported by Condor is BOINC. The *condor_startd* has the ability to start and stop the BOINC client program at the appropriate times, but otherwise provides no additional services to configure the BOINC computations themselves. Future versions of Condor might provide additional functionality to make it easier to manage BOINC computations from within Condor. For now, the BOINC client must be manually installed and configured outside of Condor on each backfill-enabled machine.

Defining the Backfill Policy

There are a small set of policy expressions that determine if a *condor_startd* will attempt to spawn a backfill client at all, and if so, to control the transitions in to and out of the Backfill state. This section briefly lists these expressions. More detail can be found in section 3.3.10 on page 179.

ENABLE_BACKFILL A boolean value to determine if any backfill functionality should be used. The default value is `False`.

BACKFILLSYSTEM A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported string is "BOINC".

START_BACKFILL A boolean expression to control if a Condor resource should start a backfill client. This expression is only evaluated when the machine is in the Unclaimed/Idle state and the `ENABLE_BACKFILL` expression is `True`.

EVICT_BACKFILL A boolean expression that is evaluated whenever a Condor resource is in the Backfill state. A value of `True` indicates the machine should immediately kill the currently running backfill client and any other spawned processes, and return to the Owner state.

The following example shows a possible configuration to enable backfill:

```
# Turn on backfill functionality, and use BOINC
ENABLE_BACKFILL = TRUE
BACKFILL_SYSTEM = BOINC

# Spawn a backfill job if we've been Unclaimed for more than 5
# minutes
START_BACKFILL = $(StateTimer) > (5 * $(MINUTE))

# Evict a backfill job if the machine is busy (based on keyboard
# activity or cpu load)
EVICT_BACKFILL = $(MachineBusy)
```

Overview of the BOINC system

The BOINC system is a distributed computing environment for solving large scale scientific problems. A detailed explanation of this system is beyond the scope of this manual. Thorough documentation about BOINC is available at their website: <http://boinc.berkeley.edu>. However, a brief overview is provided here for sites interested in using BOINC with Condor to manage backfill jobs.

BOINC grew out of the relatively famous SETI@home computation, where volunteers installed special client software, in the form of a screen saver, that contacted a centralized server to download work units. Each work unit contained a set of radio telescope data and the computation tried to find patterns in the data, a sign of intelligent life elsewhere in the universe (hence the name: “Search for Extra Terrestrial Intelligence at home”). BOINC is developed by the Space Sciences Lab at the University of California, Berkeley, by the same people who created SETI@home. However, instead of being tied to the specific radio telescope application, BOINC is a generic infrastructure by which many different kinds of scientific computations can be solved. The current generation of SETI@home now runs on top of BOINC, along with various physics, biology, climatology, and other applications.

The basic computational model for BOINC and the original SETI@home is the same: volunteers install BOINC client software which runs whenever the machine would otherwise be idle. However, the BOINC installation on any given machine must be configured so that it knows what computations to work for (each computation is referred to as a *project* using BOINC’s terminology), instead of always working on a hard coded computation. A given BOINC client can be configured to donate all of its cycles to a single project, or to split the cycles between projects so that, on average, the desired percentage of the computational power is allocated to each project. Once the client software (a program called the *boinc_client*) starts running, it attempts to contact a centralized server for each project it has been configured to work for. The BOINC software downloads the appropriate platform-specific application binary and some work units from the central server for each project. Whenever the client software completes a given work unit, it once again attempts to connect to that project’s central server to upload the results and download more work.

BOINC participants must register at the centralized server for each project they wish to donate cycles to. The process produces a unique identifier so that the work performed by a given client can be credited to a specific user. BOINC keeps track of the work units completed by each user, so that users providing the most cycles get the highest rankings (and therefore, bragging rights).

Because BOINC already handles the problems of distributing the application binaries for each scientific computation, the work units, and compiling the results, it is a perfect system for managing backfill computations in Condor. Many of the applications that run on top of BOINC produce their own application-specific checkpoints, so even if the *boinc_client* is killed (for example, when a Condor job arrives at a machine, or if the interactive user returns) an entire work unit will not necessarily be lost.

Installing the BOINC client software

If a working installation of BOINC currently exists on machines where backfill is desired, skip the remainder of this section. Continue reading with the section titled “Configuring the BOINC client under Condor”.

In Condor Version 7.2.3, the BOINC client software that actually spawns and manages the backfill computations (the *boinc_client*) must be manually downloaded, installed and configured outside of Condor. Hopefully in future versions, the Condor package will include the *boinc_client*, and there will be a way to automatically install and configure the BOINC software together with Condor.

The *boinc_client* executables can be obtained at one of the following locations:

<http://boinc.berkeley.edu/download.php> This is the official BOINC download site, which provides binaries for MacOS 10.3 or higher, Linux/x86, Solaris/SPARC and Windows/x86. From the download table, use the “Recommended version”, and use the “Core client only (command-line)” package when available.

http://boinc.berkeley.edu/download_other.php This page contains links to sites that distribute *boinc_client* binaries for other platforms beyond the officially supported ones.

Once the BOINC client software has been downloaded, the *boinc_client* binary should be placed in a location where the Condor daemons can use it. The path will be specified via a Condor configuration setting, `BOINC_Executable`, described below.

Additionally, a local directory on each machine should be created where the BOINC system can write files it needs. This directory must not be shared by multiple instances of the BOINC software, just like the `spool` or `execute` directories used by Condor. This location of this directory is defined using the `BOINC_InitialDir` macro, described below. The directory must be writable by whatever user the *boinc_client* will run as. This user is either the same as the user the Condor daemons are running as (if Condor is not running as root), or a user defined via the `BOINC_Owner` setting described below.

Finally, Condor administrators wishing to use BOINC for backfill jobs must create accounts at the various BOINC projects they want to donate cycles to. The details of this process vary

from project to project. Beware that this step must be done manually, as the BOINC software spawned by Condor (the *boinc_client*) can not automatically register a user at a given project (unlike the more fancy GUI version of the BOINC client software which many users run as a screen saver). For example, to configure machines to perform work for the Einstein@home project (a physics experiment run by the University of Wisconsin at Milwaukee) Condor administrators should go to http://einstein.phys.uwm.edu/create_account_form.php, fill in the web form, and generate a new Einstein@home identity. This identity takes the form of a project URL (such as <http://einstein.phys.uwm.edu>) followed by an *account key*, which is a long string of letters and numbers that is used as a unique identifier. This URL and account key will be needed when configuring Condor to use BOINC for backfill computations (described in the next section).

Configuring the BOINC client under Condor

This section assumes that the BOINC client software has already been installed on a given machine, that the BOINC projects to join have been selected, and that a unique project account key has been created for each project. If any of these steps has not been completed, please read the previous section titled “Installing the BOINC client software”

Whenever the *condor_startd* decides to spawn the *boinc_client* to perform backfill computations (when `ENABLE_BACKFILL` is `True`, when the resource is in `Unclaimed/Idle`, and when the `START_BACKFILL` expression evaluates to `True`), it will spawn a *condor_starter* to directly launch and monitor the *boinc_client* program. This *condor_starter* is just like the one used to spawn normal Condor jobs. In fact, the `argv[0]` of the *boinc_client* will be renamed to “*condor_exec*”, as described in section 2.16.1 on page 111.

The *condor_starter* for spawning the *boinc_client* reads values out of the Condor configuration files to define the job it should run, as opposed to getting these values from a job classified ad in the case of a normal Condor job. All of the configuration settings to control things like the path to the *boinc_client* binary to use, the command-line arguments, the initial working directory, and so on, are prefixed with the string “BOINC_”. Each possible setting is described below:

Required settings:

BOINCExecutable The full path to the *boinc_client* binary to use.

BOINCInitialDir The full path to the local directory where BOINC should run.

BOINCUniverse The Condor universe used for running the *boinc_client* program. This **must** be set to “*vanilla*” for BOINC to work under Condor.

BOINCOwner What user the *boinc_client* program should be run as. This macro is only used if the Condor daemons are running as root. In this case, the *condor_starter* must be told what user identity to switch to before spawning the *boinc_client*. This can be any valid user on the local system, but it must have write permission in whatever directory is specified in `BOINCInitialDir`.

Optional settings:

BOINCArguments Command-line arguments that should be passed to the *boinc_client* program. For example, one way to specify the BOINC project to join is to use the **–attach_project** argument to specify a project URL and account key. For example:

```
BOINCArguments = --attach_project http://einstein.phys.uwm.edu [account_key]
```

BOINCEnvironment Environment variables that should be set for the *boinc_client*.

BOINCOutput Full path to the file where STDOUT from the *boinc_client* should be written. If this macro is not defined, STDOUT will be discarded.

BOINCError Full path to the file where STDERR from the *boinc_client* should be written. If this macro is not defined, STDERR will be discarded.

The following example shows one possible usage of these settings:

```
# Define a shared macro that can be used to define other settings.
# This directory must be manually created before attempting to run
# any backfill jobs.
BOINC_HOME = $(LOCAL_DIR)/boinc

# Path to the boinc_client to use, and required universe setting
BOINC_Executable = /usr/local/bin/boinc_client
BOINC_Universe = vanilla

# What initial working directory should BOINC use?
BOINC_InitialDir = $(BOINC_HOME)

# Save STDOUT and STDERR
BOINC_Output = $(BOINC_HOME)/boinc.out
BOINC_Error = $(BOINC_HOME)/boinc.err
```

If the Condor daemons reading this configuration are running as root, an additional macro must be defined:

```
# Specify the user that the boinc_client should run as:
BOINC_Owner = nobody
```

In this case, Condor would spawn the *boinc_client* as “**nobody**”, so the directory specified in `$(BOINC_HOME)` would have to be writable by the “**nobody**” user.

A better choice would probably be to create a separate user account just for running BOINC jobs, so that the local BOINC installation is not writable by other processes running as “**nobody**”. Alternatively, the `BOINC_Owner` could be set to “**daemon**”.

Attaching to a specific BOINC project

There are a few ways to attach a Condor/BOINC installation to a given BOINC project:

- The **`--attach_project`** argument to the *boinc_client* program, defined via the `BOINC_Arguments` setting (described above). The *boinc_client* will only accept a single **`--attach_project`** argument, so this method can only be used to attach to one project.
- The *boinc_cmd* command-line tool can perform various BOINC administrative tasks, including attaching to a BOINC project. Using *boinc_cmd*, the appropriate argument to use is called **`--project_attach`**. Unfortunately, the *boinc_client* must be running for *boinc_cmd* to work, so this method can only be used once the Condor resource has entered the Backfill state and has spawned the *boinc_client*.
- Manually create account files in the local BOINC directory. Upon startup, the *boinc_client* will scan its local directory (the directory specified with `BOINC_InitialDir`) for files of the form `account_[URL].xml`, for example, `account_einstein.phys.uwm.edu.xml`. Any files with a name that matches this convention will be read and processed. The contents of the file define the project URL and the authentication key. The format is:

```
<account>
  <master_url>[URL]</master_url>
  <authenticator>[key]</authenticator>
</account>
```

For example:

```
<account>
  <master_url>http://einstein.phys.uwm.edu</master_url>
  <authenticator>aaaa1111bbbb2222cccc3333</authenticator>
</account>
```

(Of course, the `<authenticator>` tag would use the real authentication key returned when the account was created at a given project).

These account files can be copied to the local BOINC directory on all machines in a Condor pool, so administrators can either distribute them manually, or use symbolic links to point to a shared file system.

In the first two cases (using command-line arguments for *boinc_client* or running the *boinc_cmd* tool), BOINC will write out the resulting account file to the local BOINC directory on the machine, and then future invocations of the *boinc_client* will already be attached to the appropriate project(s). More information about participating in multiple BOINC projects can be found at http://boinc.berkeley.edu/multiple_projects.php.

BOINC on Windows

The Windows version of BOINC has multiple installation methods. The preferred method of installation for use with Condor is the “Shared Installation” method. Using this method gives all users access to the executables. During the installation process

1. Deselect the option which makes BOINC the default screen saver
2. Deselect the option which runs BOINC on start-up.
3. Do not launch BOINC at the conclusion of the installation.

There are three major differences from the Unix version to keep in mind when dealing with the Windows installation:

1. The Windows executables have different names from the Unix versions. The Windows client is called *boinc.exe*. Therefore, the configuration variable `BOINC_Executable` is written:

```
BOINC_Executable = C:\PROGRA~1\BOINC\boinc.exe
```

The Unix administrative tool *boinc_cmd* is called *boinccmd.exe* on Windows.

2. When using BOINC on Windows, the configuration variable `BOINC_InitialDir` will not be respected fully. To work around this difficulty, pass the BOINC home directory directly to the BOINC application via the `BOINC_Arguments` configuration variable. For Windows, rewrite the argument line as:

```
BOINC_Arguments = --dir $(BOINC_HOME) \
  --attach_project http://einstein.phys.uwm.edu [account_key]
```

As a consequence of setting the BOINC home directory, some projects may fail with the authentication error:

```
Scheduler request failed: Peer
certificate cannot be authenticated
with known CA certificates.
```

To resolve this issue, copy the *ca-bundle.crt* file from the BOINC installation directory to `$(BOINC_HOME)`. This file appears to be project and machine independent, and it can therefore be distributed as part of an automated Condor installation.

3. The `BOINC_Owner` configuration variable behaves differently on Windows than it does on Unix. Its value may take one of two forms:

- `domain\user`
- `user`

This form assumes that the user exists in the local domain (that is, on the computer itself).

Setting this option causes the addition of the job attribute

```
RunAsUser = True
```

to the backfill client. This further implies that the configuration variable `STARTER_ALLOW_RUNAS_OWNER` be set to `True` to insure that the local *condor_starter* be able to run jobs in this manner. For more information on the `RunAsUser` attribute, see section 6.2.4. For more information on the `STARTER_ALLOW_RUNAS_OWNER` configuration variable, see section 3.3.7.

3.12.10 Group ID-Based Process Tracking

One function that Condor often must perform is keeping track of all processes created by a job. This is done so that Condor can provide resource usage statistics about jobs, and also so that Condor can properly clean up any processes that jobs leave behind when they exit.

In general, tracking process families is difficult to do reliably. By default Condor uses a combination of process parent-child relationships, process groups, and information that Condor places in a job's environment to track process families on a best-effort basis. This usually works well, but it can falter for certain applications or for jobs that try to evade detection.

Jobs that run with a user account dedicated for Condor's use can be reliably tracked, since all Condor needs to do is look for all processes running using the given account. Administrators must specify in Condor's configuration what accounts can be considered dedicated via the `DEDICATED_EXECUTE_ACCOUNT_REGEX` setting. See Section 3.6.11 for further details.

Ideally, jobs can be reliably tracked regardless of the user account they execute under. This can be accomplished with group ID-based tracking. This method of tracking requires that a range of dedicated *group* IDs (GID) be set aside for Condor's use. The number of GIDs that must be set aside for an execute machine is equal to its number of execution slots. GID-based tracking is only available on Linux, and it requires that Condor either runs as `root` or uses privilege separation (see Section 3.6.12).

GID-based tracking works by placing a dedicated GID in the supplementary group list of a job's initial process. Since modifying the supplementary group ID list requires `root` privilege, the job will not be able to create processes that go unnoticed by Condor.

Once a suitable GID range has been set aside for process tracking, GID-based tracking can be enabled via the `USE_GID_PROCESS_TRACKING` parameter. The minimum and maximum GIDs included in the range are specified with the `MIN_TRACKING_GID` and `MAX_TRACKING_GID` settings. For example, the following would enable GID-based tracking for an execute machine with 8 slots.

```
USE_GID_PROCESS_TRACKING = True
MIN_TRACKING_GID = 750
MAX_TRACKING_GID = 757
```

GID-based process tracking requires use of the *condor.procd*. If `USE_GID_PROCESS_TRACKING` is true, the *condor.procd* will be used regardless of the `USE_PROCD` setting.

3.12.11 Concurrency Limits

Condor's implementation of the mechanism called *concurrency limits* allows an administrator to define and set integer limits on consumable resources. These limits are utilized during matchmaking, preventing matches when the resources are allocated. Typical uses of this mechanism will include

the management of software licenses, database connections, and any other consumable resource external to Condor.

Use of the concurrency limits mechanism requires configuration variables to set distinct limits, while jobs must identify the need for a specific resource.

In the configuration, a string must be chosen as a name for the particular resource. This name is used in the configuration of a *condor_negotiator* daemon variable that defines the concurrency limit, or integer quantity available of this resource. For example, assume that there are 3 licenses for the X software. The configuration variable concurrency limit may be:

```
XSW_LIMIT = 3
```

where "XSW" is the invented name of this resource, which is appended with the string `_LIMIT`. With this limit, a maximum of 3 jobs declaring that they need this resource may be executed concurrently.

In addition to named limits, such as in the example named limit XSW, configuration may specify a concurrency limit for all resources that are not covered by specifically-named limits. The configuration variable `CONCURRENCY_LIMIT_DEFAULT` sets this value. For example,

```
CONCURRENCY_LIMIT_DEFAULT = 1
```

sets a limit of 1 job in execution for any job that declares its requirement for a resource that is not named in the configuration. If `CONCURRENCY_LIMIT_DEFAULT` is omitted from the configuration, then no limits are placed on the number of concurrently executing jobs of resources for which there is no specifically named concurrency limit.

The job must declare its need for a resource by placing a command in its submit description file or adding an attribute to the job ClassAd. In the submit description file, an example job that requires the X software adds:

```
concurrency_limits = XSW
```

This results in the job ClassAd attribute

```
ConcurrencyLimits = "XSW"
```

The implementation of the job ClassAd attribute `ConcurrencyLimits` has a more general implementation. It is either a string or a string list. A list contains items delimited by space characters and comma characters. Therefore, a job that requires the 3 separate resources named as "XSW", "Y", and "Z", will contain in its submit description file:

```
concurrency_limits = Y,XSW,Z
```

Additionally, a numerical value identifying the number of resources required may be specified in the definition of a resource, following the resource name by a colon character and the integer

number of resources. Modifying the given example to specify that 3 of the "XSW" resource are needed results in:

```
concurrency_limits = y,XSW:3,Z
```

Note that the maximum for any given limit, as specified with the configuration variable `<*>_LIMIT`, is as strictly enforced **as possible**. In the presence of preemption and dropped updates from the *condor_startd* daemon to the *condor_collector* daemon, it is possible for the limit to be exceeded. Condor will never kill a job to free up a limit, including the case where a limit maximum is exceeded.

3.13 Java Support Installation

Compiled Java programs may be executed (under Condor) on any execution site with a Java Virtual Machine (JVM). To do this, Condor must be informed of some details of the JVM installation.

Begin by installing a Java distribution according to the vendor's instructions. We have successfully used the Sun Java Developer's Kit, but any distribution should suffice. Your machine may have been delivered with a JVM already installed – installed code is frequently found in `/usr/bin/java`.

Condor's configuration includes the location of the installed JVM. Edit the configuration file. Modify the `JAVA` entry to point to the JVM binary, typically `/usr/bin/java`. Restart the *condor_startd* daemon on that host. For example,

```
% condor_restart -startd bluejay
```

The *condor_startd* daemon takes a few moments to exercise the Java capabilities of the *condor_starter*, query its properties, and then advertise the machine to the pool as Java-capable. If the set up succeeded, then *condor_status* will tell you the host is now Java-capable by printing the Java vendor and the version number:

```
% condor_status -java bluejay
```

After a suitable amount of time, if this command does not give any output, then the *condor_starter* is having difficulty executing the JVM. The exact cause of the problem depends on the details of the JVM, the local installation, and a variety of other factors. We can offer only limited advice on these matters, but here is an approach to solving the problem.

To reproduce the test that the *condor_starter* is attempting, try running the Java *condor_starter* directly. To find where the *condor_starter* is installed, run this command:

```
% condor_config_val STARTER
```

This command prints out the path to the *condor_starter*, perhaps something like this:

```
/usr/condor/sbin/condor_starter
```

Use this path to execute the *condor_starter* directly with the *-classad* argument. This tells the starter to run its tests and display its properties.

```
/usr/condor/sbin/condor_starter -classad
```

This command will display a short list of cryptic properties, such as:

```
IsDaemonCore = True
HasFileTransfer = True
HasMPI = True
CondorVersion = "$CondorVersion: 7.1.0 Mar 26 2008 BuildID: 80210 $"
```

If the Java configuration is correct, there will also be a short list of Java properties, such as:

```
JavaVendor = "Sun Microsystems Inc."
JavaVersion = "1.2.2"
JavaMFlops = 9.279696
HasJava = True
```

If the Java installation is incorrect, then any error messages from the shell or Java will be printed on the error stream instead.

One identified difficulty occurs when the machine has a large quantity of physical RAM, and this quantity exceeds the Java limitations. This is a known problem for the Sun JVM. Condor appends the maximum amount of system RAM to the Java Maxheap Argument, and sometimes this value is larger than the JVM allows. The end result is that Condor believes that the JVM on the machine is faulty, resulting in nothing showing up as a result of executing the command `condor_status -java`.

The way to work around this particular problem is to modify the configuration file for those machines that may execute Java universe jobs. The `JAVA_MAXHEAP_ARGUMENT` macro is explicitly set to null in the configuration, to prevent Condor from appending the machine-specific, but too-big value. Then the Java Maxheap Argument is set (again, in the configuration) to the maximum value allowed for the JVM on that platform, using the `JAVA_EXTRA_ARGUMENTS` configuration variable. Note that the name of the switch that regulates the Java Maxheap Argument is different for different vendors' JVM.

The following is an example of the configuration fix for the Sun JVM:

```
# First set JAVA_MAXHEAP_ARGUMENT to null, to disable the default of max RAM
JAVA_MAXHEAP_ARGUMENT =
# Now set the argument with the Sun-specific maximum allowable value
JAVA_EXTRA_ARGUMENTS = -Xmx1906m
```

3.14 Virtual Machines

Virtual Machines can be executed on any execution site with VMware or Xen (via *libvirt*). To do this, Condor must be informed of some details of the VM installation.

What follows is not a comprehensive list of the VM Universe options; rather, it is intended to serve as a starting point for those users interested in getting VM Universe up and running quickly. For further, more comprehensive coverage of the configuration options please refer to section 3.3.28.

Begin by installing the virtualization package according to the vendor's instructions. We have successfully used both VMware Server and Xen. If you are considering running on a Windows system, you will also need to install a Perl distribution; for this we have used ActivePerl successfully.

If you are considering Xen, then there are four things that must exist on a system to fully support it. First, a Xen kernel must be running on the execute machine. This running Xen kernel acts as Dom0, in Xen terminology, under which all VMs are started, called DomUs in Xen terminology. Second, the *virsh* utility must be available, and its companion *libvirtd* and *Xend* services must be running. Third, a reasonably recent version of the *mkisofs* utility must be available, for creation of CD-ROM disk images. Fourth, the *pygrub* program must be available, for execution of VMs whose disks contain the kernel they will run.

3.14.1 Configuration Parameters

There are a number of configuration parameters related to virtual machines. Some options are required, while others are optional. Here we only discuss those that are required.

First, you are required to specify the type of VM that is installed. For instance, the following tells Condor we are using VMware:

```
VM_TYPE = vmware
```

You are also required to specify the location of *condor_vm-gahp* and its log file. On a Windows installation, these options would look like this:

```
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp.exe
VM_GAHP_LOG = $(LOG)/VMGahpLog
```

You must also provide a version string for the Virtual Machine software you are using:

```
VM_VERSION = server1.0.4
```

While required, this option does not alter the behavior of Condor. Instead, it is added to the ClassAd for the machine, so it can be matched against. This way, if future releases of VMware/Xen support new features that are desirable for your job, you can match on this string.

VMware-Specific Configuration

If you are using VMware you also need to set the location of the Perl executable. In most cases, however, the default value should suffice:

```
VMWARE_PERL = perl
```

This, of course, assumes the Perl executable is in the path. If this is not the case, then a full path to the Perl executable will be required.

The final required option is the location of the VMware control script. It's located in Condor's `sbin` directory:

```
VMWARE_PERL = $(SBIN)/condor_vm_vmware.pl
```

Finally, note that an execute machine's `EXECUTE` setting should not contain any symlinks in its path if the machine is configured to run VMware jobs. See the FAQ entry in section 7.3 for details.

Xen-Specific Configuration

Xen configurations must set the location of the control script Condor uses to interact with Xen:

```
XEN_SCRIPT = $(SBIN)/condor_vm_xen.sh
```

The other required option not included in the default Xen configuration is `XEN_DEFAULT_KERNEL`: this is the kernel image that will be used in cases where the user does not specify one explicitly in their job submission. In most cases, this can be the default kernel from which the system was booted. For instance, the following was used on a Fedora Core installation:

```
XEN_DEFAULT_KERNEL = /boot/vmlinuz-2.6.18-1.2798.fc6xen
```

There is one final option worth mentioning: `XEN_DEFAULT_INITRD`. It's not a required option, but if you do decide to use it, there are a few things that you should be careful with. Unlike the kernel image above, this image cannot be the stock one used to boot the system. The reason for this is that Xen requires several device drivers in DomUs: *xennet* and *xenblk*. This can be easily fixed by creating a new *initrd* using *mkinitrd* and loading the drivers into it.

Once the configuration options have been set, restart the *condor_startd* daemon on that host. For example:

```
> condor_restart -startd leovinus
```

The *condor_startd* daemon takes a few moments to exercise the VM capabilities of the *condor_vm-gahp*, query its properties, and then advertise the machine to the pool as VM-capable. If the set up succeeded, then *condor_status* will tell you the host is now VM-capable by printing the VM type and the version number:

```
> condor_status -vm leovinus
```

After a suitable amount of time, if this command does not give any output, then the *condor_vm-gahp* is having difficulty executing the VM software. The exact cause of the problem depends on the details of the VM, the local installation, and a variety of other factors. We can offer only limited advice on these matters:

For Xen, the VM Universe is only available when root starts Condor. This is a restriction currently imposed because root privileges are required to create a VM on top of a Xen kernel. Specifically, root is needed to properly use the *virsh* utility that controls creation and management of Xen guest virtual machines. This restriction may be lifted in future versions depending on features provided by the underlying tools, *virsh*, or upon Condor's direct support of Qemu VMs that do not require network access.

3.15 Power Management

Condor supports placing machines in low power states. Power setting decisions are based upon Condor configuration.

Power conservation is relevant when machines are not in heavy use, or when there are known periods of low activity within the pool.

3.15.1 Entering a Low Power State

By default, Condor does not do power management. When desired, the ability to place a machine into a low power state is accomplished through configuration. This occurs when all slots on a machine agree that a low power state is desired.

A slot's readiness to hibernate is determined by the evaluating the `HIBERNATE` configuration variable within the context of the slot. Readiness is evaluated at fixed intervals, as determined by the `HIBERNATE_CHECK_INTERVAL` configuration variable. A non-zero value of this variable enables the power management facility. It is an integer value representing seconds, and it need not be a small value. There is a trade off between the accuracy of measurement and the unnecessary computation of readiness.

To put the machine in a low power state rapidly after it has become idle, consider checking each slot's state frequently, as in the example configuration:

```
HIBERNATE_CHECK_INTERVAL = 20
```

This checks each slot's readiness every 20 seconds. A more common value for frequency of checks is 300 (5 minutes). A value of 300 loses some degree of granularity, but it is more reasonable as machines are likely to be put in to a low power state after a few hours, rather than minutes.

A slot's readiness or willingness to enter a low power state is determined by the HIBERNATE expression. Because this expression is evaluated in the context of each slot, and not on the machine as a whole, any one slot can veto a change of power state. The HIBERNATE expression may include a wide array of variables. Possibilities include the change in power state if none of the slots are claimed, or if the slots are not in the Owner state. See the description of HIBERNATE in section 3.3.10 on page 193 for definitions of the values representing the different power states.

Here is a concrete example. Assume that the START expression is not set to always be True. This permits an easy determination whether or not the machine is in an Unclaimed state through the use of an auxiliary macro called ShouldHibernate.

```
ShouldHibernate = ((KeyboardIdle > $(StartIdleTime)) \
&& $(CPUIidle) \
&& ($(StateTimer) > (2 * $(HOUR)))
```

This macro evaluates to True if

- The keyboard has been idle long enough.
- The CPU is idle
- The slot has been Unclaimed for more than 2 hours.

The HIBERNATE expression that enters the power state called "RAM" if ShouldHibernate evaluates to True, and remains in its current state otherwise is

```
HIBERNATE = ifThenElse($(ShouldHibernate), "RAM", "NONE" )
```

If any slot returns "NONE", it vetoes the decision to enter a low power state. Only when values returned by all slots are all non-zero is there a decision to enter a low power state. If all agree to enter the low power state, but differ in which state to enter, then the largest magnitude value is chosen.

3.15.2 Returning From a Low Power State

The Condor command line tool *condor-power* may wake a machine from a low power state by sending a UDP Wake On LAN (WOL) packet.

See the *condor-power* manual page on page 744.

3.15.3 Keeping a ClassAd for a Hibernating Machine

A pool's *condor_collector* daemon can be configured to keep a persistent ClassAd entry for each machine, once it has entered hibernation.

To do this, define a log file using the `OFFLINE_LOG` configuration variable. An optional expiration time for each ClassAd can be specified with `OFFLINE_EXPIRE_ADS_AFTER`. The timing begins from the time the hibernating machine's ClassAd enters the *condor_collector* daemon. A description of `OFFLINE_LOG` is at section 3.3.10 on page 194, and a description of `OFFLINE_EXPIRE_ADS_AFTER` is at section 3.3.10 on page 194.

3.15.4 Linux Platform Details

Depending on the Linux distribution and version, there are three methods for controlling a machine's power state. The methods:

1. *pm-utils* is a set of command line tools which can be used to detect and switch power states. In Condor, this is defined by the string "pm-utils".
2. The directory in the virtual file system `/sys/power` contains virtual files that can be used to detect and set the power states. In Condor, this is defined by the string `/sys`.
3. The directory in the virtual file system `/proc/acpi` contains virtual files that can be used to detect and set the power states. In Condor, this is defined by the string `/proc`.

By default, the Condor attempts to detect the method to use in the order shown. The first method detected as usable on the system is chosen.

This ordered detection may be bypassed, to use a specified method instead by setting the configuration variable `LINUX_HIBERNATION_METHOD` with one of the defined strings. This variable is defined in section 3.3.10 on page 194. If no usable methods are detected or the method specified by `LINUX_HIBERNATION_METHOD` is either not detected or invalid, hibernation is disabled.

The details of this selection process, and the final method selected can be logged via enabling `D_FULLDEBUG` in the relevant subsystem's log configuration.

3.15.5 Windows Platform Details

If after a suitable amount of time, a Windows machine has not entered the expected power state, then Condor is having difficulty exercising the operating system's low power capabilities. While the cause will be specific to the machine's hardware, it may also be due to improperly configured software. For hardware difficulties, the likely culprit is the configuration within the machine's BIOS, for which Condor can offer little guidance. For operating system difficulties, the Vista the *powercfg* tool can be used to discover the available power states on the machine. The following command demonstrates how to list all of the supported power states of the machine:

```
> powercfg -A
The following sleep states are available on this system:
Standby (S3) Hibernate Hybrid Sleep
The following sleep states are not available on this system:
Standby (S1)
    The system firmware does not support this standby state.
Standby (S2)
    The system firmware does not support this standby state.
```

Note that the HIBERNATE expression is written in terms of the S_n state, where n is the value evaluated from the expression.

This tool can also be used to enable and disable other sleep states. This example turns hibernation on.

```
> powercfg -h on
```

If this tool is insufficient for configuring the machine in the manner required, the *Power Options* control panel application offers the full extent of the machine's power management abilities. Windows 2000 and XP lack the *powercfg* program, so all configuration must be done via the *Power Options* control panel application.

Miscellaneous Concepts

This chapter contains sections describing a variety of key Condor concepts that do not belong in other chapters.

ClassAds and the ClassAd language are presented.

Details of checkpoints are presented.

Description and usage of COD (Computing on Demand) extensions to Condor are presented.

The various APIs that Condor implements are described.

4.1 Condor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the Condor system. ClassAds are used extensively in the Condor system to represent jobs, resources, submitters and other Condor daemons. An understanding of this mechanism is required to harness the full flexibility of the Condor system.

A ClassAd is a set of uniquely named expressions. Each named expression is called an *attribute*. Figure 4.1 shows an example of a ClassAd with ten attributes.

ClassAd expressions look very much like expressions in C, and are composed of literals and attribute references composed with operators and functions. The difference between ClassAd expressions and C expressions arise from the fact that ClassAd expressions operate in a much more dynamic environment. For example, an expression from a machine's ClassAd may refer to an attribute in a job's ClassAd, such as `TARGET.Owner` in the above example. The value and type of the attribute is not known until the expression is evaluated in an environment which pairs a specific

```

MyType      = "Machine"
TargetType  = "Job"
Machine     = "froth.cs.wisc.edu"
Arch        = "INTEL"
OpSys       = "SOLARIS251"
Disk        = 35882
Memory      = 128
KeyboardIdle = 173
LoadAvg     = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60

```

Figure 4.1: An example ClassAd

job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be *total* operators, which means that they have well defined behavior regardless of supplied operands. This functionality is provided through two distinguished values, `UNDEFINED` and `ERROR`, and defining all operators so that they can operate on all possible values in the ClassAd system. For example, the multiplication operator which usually only operates on numbers, has a well defined behavior if supplied with values which are not meaningful to multiply. Thus, the expression `10 * "A string"` evaluates to the value `ERROR`. Most operators are *strict* with respect to `ERROR`, which means that they evaluate to `ERROR` if any of their operands are `ERROR`. Similarly, most operators are strict with respect to `UNDEFINED`.

4.1.1 Syntax

ClassAd expressions are formed by composing literals, attribute references and other sub-expressions with operators and functions.

Literals

Literals in the ClassAd language may be of integer, real, string, undefined or error types. The syntax of these literals is as follows:

Integer A sequence of continuous digits (i.e., `[0-9]`). Additionally, the keywords `TRUE` and `FALSE` (case insensitive) are syntactic representations of the integers 1 and 0 respectively.

Real Two sequences of continuous digits separated by a period (i.e., `[0-9]+.[0-9]+`).

String A double quote character, followed by an list of characters terminated by a double quote character. A backslash character inside the string causes the following character to be considered as part of the string, irrespective of what that character is.

Undefined The keyword `UNDEFINED` (case insensitive) represents the `UNDEFINED` value.

Error The keyword `ERROR` (case insensitive) represents the `ERROR` value.

Attributes

Every expression in a ClassAd is named by an *attribute name*. Together, the (name,expression) pair is called an *attribute*. An attributes may be referred to in other expressions through its attribute name.

Attribute names are sequences of alphabetic characters, digits and underscores, and may not begin with a digit. All characters in the name are significant, but case is *not* significant. Thus, Memory, memory and MeMoRY all refer to the same attribute.

An *attribute reference* consists of the name of the attribute being referenced, and an optional *scope resolution prefix*. The prefixes that may be used are MY. and TARGET.. The case used for these prefixes is *not* significant. The semantics of supplying a prefix are discussed in Section 4.1.2.

Operators

The operators that may be used in ClassAd expressions are similar to those available in C. The available operators and their relative precedence is shown in figure 4.2. The operator with the highest

```

- (unary negation)    (high precedence)
*      /
+      - (addition, subtraction)
<      <=      >=      >
==      !=      ==?      !=?
&&
||
                                (low precedence)

```

Figure 4.2: Relative precedence of ClassAd expression operators

precedence is the unary minus operator. The only operators which are unfamiliar are the ==? and !=? operators, which are discussed in Section 4.1.2.

Predefined Functions

Any ClassAd expression may utilize predefined functions. Function names are case insensitive. Parameters to functions and a return value from a function may be typed (as given) or not. Nested or recursive function calls are allowed.

Here are descriptions of each of these predefined functions. The possible types are the same as itemized in in Section 4.1.1. Where the type may be any of these literal types, it is called out as AnyType. Where the type is Integer, but only returns the value 1 or 0 (implying True or False), it is called out as Boolean. The format of each function is given as

```
ReturnType FunctionName(ParameterType parameter1, ParameterType parameter2, ...)
```

Optional parameters are given within square brackets.

AnyType ifThenElse(AnyType IfExpr, AnyType ThenExpr, AnyType ElseExpr)

A conditional expression is described by `IfExpr`. The following defines return values, when `IfExpr` evaluates to

- `True`. Evaluate and return the value as given by `ThenExpr`.
- `False`. Evaluate and return the value as given by `ElseExpr`.
- `UNDEFINED`. Return the value `UNDEFINED`.
- `ERROR`. Return the value `ERROR`.
- `0.0`. Evaluate, and return the value as given by `ElseExpr`.
- non-`0.0` Real values. Evaluate, and return the value as given by `ThenExpr`.

Where `IfExpr` evaluates to give a value of type `String`, the function returns the value `ERROR`. The implementation uses lazy evaluation, so expressions are only evaluated as defined.

This function returns `ERROR` if other than exactly 3 arguments are given.

Boolean isUndefined(AnyType Expr) Returns `True`, if `Expr` evaluates to `UNDEFINED`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

Boolean isError(AnyType Expr) Returns `True`, if `Expr` evaluates to `ERROR`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

Boolean isString(AnyType Expr) Returns `True`, if the evaluation of `Expr` gives a value of type `String`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

Boolean isInteger(AnyType Expr) Returns `True`, if the evaluation of `Expr` gives a value of type `Integer`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

Boolean isReal(AnyType Expr) Returns `True`, if the evaluation of `Expr` gives a value of type `Real`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

Boolean isBoolean(AnyType Expr) Returns `True`, if the evaluation of `Expr` gives the integer value 0 or 1. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer int(AnyType Expr) Returns the integer value as defined by Expr. Where the type of the evaluated Expr is Real, the value is truncated (round towards zero) to an integer. Where the type of the evaluated Expr is String, the string is converted to an integer using a C-like atoi() function. When this result is not an integer, ERROR is returned. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Real real(AnyType Expr) Returns the real value as defined by Expr. Where the type of the evaluated Expr is Integer, the return value is the converted integer. Where the type of the evaluated Expr is String, the string is converted to a real value using a C-like atof() function. When this result is not a real, ERROR is returned. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

String string(AnyType Expr) Returns the string that results from the evaluation of Expr. Converts a non-string value to a string. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer floor(AnyType Expr) Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the largest magnitude integer that is not larger than the returned value. Where real(Expr) returns ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer ceiling(AnyType Expr) Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the smallest magnitude integer that is not less than the returned value. Where real(Expr) returns ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer round(AnyType Expr) Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the integer that results from a round-to-nearest rounding method. The nearest integer value to the return value is returned, except in the case of the value at the exact midpoint between two integer values. In this case, the even valued integer is returned. Where real(Expr) returns ERROR or UNDEFINED, or the integer value does not fit into 32 bits, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

Integer random([AnyType Expr]) Where the optional argument Expr evaluates to type Integer or type Real (and called x), the return value is the integer or real r randomly

chosen from the interval $0 \leq r < x$. With no argument, the return value is chosen with `random(1.0)`. Returns `ERROR` in all other cases.

This function returns `ERROR` if greater than 1 argument is given.

String strcat(AnyType Expr1 [, AnyType Expr2..]) Returns the string which is the concatenation of all arguments, where all arguments are converted to type `String` by function `string(Expr)`. Returns `ERROR` if any argument evaluates to `UNDEFINED` or `ERROR`.

String substr(String s, Integer offset [, Integer length]) Returns the substring of `s`, from the position indicated by `offset`, with (optional) `length` characters. The first character within `s` is at offset 0. If the optional `length` argument is not present, the substring extends to the end of the string. If `offset` is negative, the value $(length - offset)$ is used for the offset. If `length` is negative, an initial substring is computed, from the offset to the end of the string. Then, the absolute value of `length` characters are deleted from the right end of the initial substring. Further, where characters of this resulting substring lie outside the original string, the part that lies within the original string is returned. If the substring lies completely outside of the original string, the null string is returned.

This function returns `ERROR` if greater than 3 or less than 2 arguments are given.

Integer strcmp(AnyType Expr1, AnyType Expr2) Both arguments are converted to type `String` by function `string(Expr)`. The return value is an integer that will be

- less than 0, if `Expr1` is lexicographically less than `Expr2`
- equal to 0, if `Expr1` is lexicographically equal to `Expr2`
- greater than 0, if `Expr1` is lexicographically greater than `Expr2`

Case is significant in the comparison. Where either argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than 2 arguments are given.

Integer stricmp(AnyType Expr1, AnyType Expr2) This function is the same as `strcmp`, except that letter case is *not* significant.

String toUpper(AnyType Expr) The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all lower case letters converted to upper case. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if greater than 1 argument is given.

String toLower(AnyType Expr) The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all upper case letters converted to lower case. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer size(AnyType Expr) Returns the number of characters in the string, after calling function `string(Expr)`. If the argument evaluates to `ERROR` or `UNDEFINED`, `ERROR` is returned.

This function returns `ERROR` if other than exactly 1 argument is given.

Integer time() Returns the current coordinated universal time, which is the same as the ClassAd attribute `CurrentTime`. This is the time, in seconds, since midnight of January 1, 1970.

String formatTime([Integer time] [, String format]) Returns a formatted string that is a representation of time. The argument `time` is interpreted as coordinated universe time in seconds, since midnight of January 1, 1970. If not specified, `time` will default to the value of attribute `CurrentTime`.

The argument `format` is interpreted similarly to the `format` argument of the ANSI C `strftime` function. It consists of arbitrary text plus placeholders for elements of the time. These placeholders are percent signs (%) followed by a single letter. To have a percent sign in the output, use a double percent sign (%%). If `format` is not specified, it defaults to `%c`.

Because the implementation uses `strftime()` to implement this, and some versions implement extra, non-ANSI C options, the exact options available to an implementation may vary. An implementation is only required to implement the ANSI C options, which are:

- %a** abbreviated weekday name
- %A** full weekday name
- %b** abbreviated month name
- %B** full month name
- %c** local date and time representation
- %d** day of the month (01-31)
- %H** hour in the 24-hour clock (0-23)
- %I** hour in the 12-hour clock (01-12)
- %j** day of the year (001-366)
- %m** month (01-12)
- %M** minute (00-59)
- %p** local equivalent of AM or PM
- %S** second (00-59)
- %U** week number of the year (Sunday as first day of week) (00-53)
- %w** weekday (0-6, Sunday is 0)
- %W** week number of the year (Monday as first day of week) (00-53)
- %x** local date representation
- %X** local time representation
- %y** year without century (00-99)

%Y year with century

%Z time zone name, if any

String interval(Integer seconds) Uses seconds to return a string of the form days+hh:mm:ss. This represents an interval of time. Leading values that are zero are omitted from the string. For example, seconds of 67 becomes "1:07". A second example, seconds of $1472523 = 17*24*60*60 + 1*60*60 + 2*60 + 3$, results in the string "17+1:02:03".

For the following functions, a delimiter is represented by a string. Each character within the delimiter string delimits individual strings within a list of strings that is given by a single string. The default delimiter contains the comma and space characters. A string within the list is ended (delimited) by one or more characters within the delimiter string.

Integer stringListSize(String list [, String delimiter]) Returns the number of elements in the string list, as delimited by the optional delimiter string. Returns ERROR if either argument is not a string.

This function returns ERROR if other than 1 or 2 arguments are given.

Integer stringListSum(String list [, String delimiter])

OR Real stringListSum(String list [, String delimiter]) Sums and returns the sum of all items in the string list, as delimited by the optional delimiter string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR.

Real stringListAve(String list [, String delimiter]) Sums and returns the real-valued average of all items in the string list, as delimited by the optional delimiter string. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value 0.0.

Integer stringListMin(String list [, String delimiter])

OR Real stringListMin(String list [, String delimiter]) Finds and returns the minimum value from all items in the string list, as delimited by the optional delimiter string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value UNDEFINED.

Integer stringListMax(String list [, String delimiter])

OR Real stringListMax(String list [, String delimiter]) Finds and returns the maximum value from all items in the string list, as delimited by the optional delimiter string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not

represent an integer or real value, the return value is `ERROR`. A list with 0 items (the empty list) returns the value `UNDEFINED`.

Boolean stringListMember(String x, String list [, String delimiter])

Returns `TRUE` if item `x` is in the string list, as delimited by the optional delimiter string. Returns `FALSE` if item `x` is not in the string list. Comparison is done with `strcmp()`. The return value is `ERROR`, if any of the arguments are not strings.

Boolean stringListIMember(String x, String list [, String delimiter])

Same as `stringListMember()`, but comparison is done with `stricmp()`, so letter case is not relevant.

The following three functions utilize regular expressions as defined and supported by the PCRE library. See <http://www.pcre.org> for complete documentation of regular expressions.

The `options` argument to these functions is a string of special characters that modify the use of the regular expressions. Inclusion of characters other than these as options are ignored.

I or i Ignore letter case.

M or m Modifies the interpretation of the carat (^) and dollar sign (\$) characters. The carat character matches the start of a string, as well as after each newline character. The dollar sign character matches before a newline character.

S or s The period matches any character, including the newline character.

X or x Ignore both white space and comments within the pattern. A comment is defined by starting with the pound sign (#) character, and continuing until the newline character.

Boolean regexp(String pattern, String target [, String options])

Returns `TRUE` if the string `target` is a regular expression as described by `pattern`. Returns `FALSE` otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns `ERROR`.

String regexps(String pattern, String target, String substitute,

[String options]) The regular expression `pattern` is applied to `target`. If the string `target` is a regular expression as described by `pattern`, the string `substitute` is returned, with backslash expansion performed. The return value is `ERROR`, if any of the arguments are not strings.

Boolean stringListRegexpMember(String pattern, String list [, String delimiter]

[, String options]) Returns `TRUE` if any of the strings within the list is a regular expression as described by `pattern`. Returns `FALSE` otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns `ERROR`. To include the fourth (optional) argument `options`, a third argument of `delimiter` is required. A default value for a delimiter is `","`.

4.1.2 Evaluation Semantics

The ClassAd mechanism's primary purpose is for matching entities that supply constraints on candidate matches. The mechanism is therefore defined to carry out expression evaluations in the context of two ClassAds that are testing each other for a potential match. For example, the *condor_negotiator* evaluates the `Requirements` expressions of machine and job ClassAds to test if they can be matched. The semantics of evaluating such constraints is defined below.

Literals

Literals are self-evaluating. Thus, integer, string, real, undefined and error values evaluate to themselves.

Attribute References

Since the expression evaluation is being carried out in the context of two ClassAds, there is a potential for name space ambiguities. The following rules define the semantics of attribute references made by ad *A* that is being evaluated in a context with another ad *B*:

1. If the reference is prefixed by a scope resolution prefix,
 - If the prefix is `MY .`, the attribute is looked up in ClassAd *A*. If the named attribute does not exist in *A*, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
 - Similarly, if the prefix is `TARGET .`, the attribute is looked up in ClassAd *B*. If the named attribute does not exist in *B*, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
2. If the reference is not prefixed by a scope resolution prefix,
 - If the attribute is defined in *A*, the value of the reference is the value of the expression bound to the attribute name in *A*.
 - Otherwise, if the attribute is defined in *B*, the value of the reference is the value of the expression bound to the attribute name in *B*.
 - Otherwise, if the attribute is defined in the ClassAd environment, the value from the environment is returned. This is a special environment, to be distinguished from the Unix environment. Currently, the only attribute of the environment is `CurrentTime`, which evaluates to the integer value returned by the system call `time(2)`.
 - Otherwise, the value of the reference is `UNDEFINED`.
3. Finally, if the reference refers to an expression that is itself in the process of being evaluated, there is a circular dependency in the evaluation. The value of the reference is `ERROR`.

Operators

All operators in the ClassAd language are *total*, and thus have well defined behavior regardless of the supplied operands. Furthermore, most operators are *strict* with respect to `ERROR` and `UNDEFINED`, and thus evaluate to `ERROR` (or `UNDEFINED`) if either of their operands have these exceptional values.

- **Arithmetic operators:**

1. The operators `*`, `/`, `+` and `-` operate arithmetically only on integers and reals.
2. Arithmetic is carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is an integer and the other real.
3. The operators are strict with respect to both `UNDEFINED` and `ERROR`.
4. If either operand is not a numerical type, the value of the operation is `ERROR`.

- **Comparison operators:**

1. The comparison operators `==`, `!=`, `<=`, `<`, `>=` and `>` operate on integers, reals and strings.
2. String comparisons are case insensitive for most operators. The only exceptions are the operators `==?` and `!=?`, which do case sensitive comparisons assuming both sides are strings.
3. Comparisons are carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is a real, and the other an integer. Strings may not be converted to any other type, so comparing a string and an integer or a string and a real results in `ERROR`.
4. The operators `==`, `!=`, `<=`, `<` and `>=` are strict with respect to both `UNDEFINED` and `ERROR`.
5. In addition, the operators `==?` and `!=?` behave similar to `==` and `!=`, but are not strict. Semantically, the `==?` tests if its operands are “identical,” i.e., have the same type and the same value. For example, `10 == UNDEFINED` and `UNDEFINED == UNDEFINED` both evaluate to `UNDEFINED`, but `10 ==? UNDEFINED` and `UNDEFINED ==? UNDEFINED` evaluate to `FALSE` and `TRUE` respectively. The `!=?` operator test for the “is not identical to” condition.

- **Logical operators:**

1. The logical operators `&&` and `||` operate on integers and reals. The zero value of these types are considered `FALSE` and non-zero values `TRUE`.
2. The operators are *not* strict, and exploit the “don’t care” properties of the operators to squash `UNDEFINED` and `ERROR` values when possible. For example, `UNDEFINED && FALSE` evaluates to `FALSE`, but `UNDEFINED || FALSE` evaluates to `UNDEFINED`.
3. Any string operand is equivalent to an `ERROR` operand for a logical operator. In other words, `TRUE && "foobar"` evaluates to `ERROR`.

4.1.3 ClassAds in the Condor System

The simplicity and flexibility of ClassAds is heavily exploited in the Condor system. ClassAds are not only used to represent machines and jobs in the Condor pool, but also other entities that exist in the pool such as checkpoint servers, submitters of jobs and master daemons. Since arbitrary expressions may be supplied and evaluated over these ads, users have a uniform and powerful mechanism to specify constraints over these ads. These constraints can take the form of `Requirements` expressions in resource and job ads, or queries over other ads.

Constraints and Preferences

The `requirements` and `rank` expressions within the submit description file are the mechanism by which users specify the constraints and preferences of jobs. For machines, the configuration determines both constraints and preferences of the machines.

For both machine and job, the `rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, a job ad may contain the following expressions:

```
Requirements = Arch=="SUN4u" && OpSys == "SOLARIS251"
Rank          = TARGET.Memory + TARGET.Mips
```

In this case, the job requires an UltraSparc computer running the Solaris 2.5.1 operating system. Among all such computers, the customer prefers those with large physical memories and high MIPS ratings. Since the `Rank` is a user-specified metric, *any* expression may be used to specify the perceived desirability of the match. The `condor_negotiator` daemon runs algorithms to deliver the best resource (as defined by the `rank` expression) while satisfying other required criteria.

Similarly, the machine may place constraints and preferences on the jobs that it will run by setting the machine's configuration. For example,

```
Friend          = Owner == "tannenba" || Owner == "wright"
ResearchGroup   = Owner == "jbasney" || Owner == "raman"
Trusted         = Owner != "rival" && Owner != "riffraff"
START          = Trusted && ( ResearchGroup || LoadAvg < 0.3 &&
                          KeyboardIdle > 15*60 )
RANK            = Friend + ResearchGroup*10
```

The above policy states that the computer will never run jobs owned by users `rival` and `riffraff`, while the computer will always run a job submitted by members of the research group. Furthermore, jobs submitted by friends are preferred to other foreign jobs, and jobs submitted by the research group are preferred to jobs submitted by friends.

Note: Because of the dynamic nature of ClassAd expressions, there is no *a priori* notion of an integer-valued expression, a real-valued expression, etc. However, it is intuitive to think of the `Requirements` and `Rank` expressions as integer-valued and real-valued expressions, respectively. If the actual type of the expression is not of the expected type, the value is assumed to be zero.

Querying with ClassAd Expressions

The flexibility of this system may also be used when querying ClassAds through the *condor_status* and *condor_q* tools which allow users to supply ClassAd constraint expressions from the command line.

For example, to find all computers which have had their keyboards idle for more than 20 minutes and have more than 100 MB of memory:

```
% condor_status -const 'KeyboardIdle > 20*60 && Memory > 100'
```

Name	Arch	OpSys	State	Activity	LoadAv	Mem	ActvtyTime
amul.cs.wi	SUN4u	SOLARIS251	Claimed	Busy	1.000	128	0+03:45:01
aura.cs.wi	SUN4u	SOLARIS251	Claimed	Busy	1.000	128	0+00:15:01
balder.cs.	INTEL	SOLARIS251	Claimed	Busy	1.000	1024	0+01:05:00
beatrice.c	INTEL	SOLARIS251	Claimed	Busy	1.000	128	0+01:30:02
...							
...							

	Machines	Owner	Claimed	Unclaimed	Matched	Preempting
SUN4u/SOLARIS251	3	0	3	0	0	0
INTEL/SOLARIS251	21	0	21	0	0	0
SUN4x/SOLARIS251	3	0	3	0	0	0
INTEL/WINNT51	1	0	0	1	0	0
INTEL/LINUX	1	0	1	0	0	0
Total	29	0	28	1	0	0

Here is an example that utilizes a regular expression ClassAd function to list specific information. A file contains ClassAd information. *condor_advertise* is used to inject this information, and *condor_status* constrains the search with an expression that contains a ClassAd function.

```
% cat ad
MyType = "Generic"
FauxType = "DBMS"
Name = "random-test"
Machine = "f05.cs.wisc.edu"
MyAddress = "<128.105.149.105:34000>"
DaemonStartTime = 1153192799
UpdateSequenceNumber = 1

% condor_advertise UPDATE_AD_GENERIC ad

% condor_status -any -constraint 'FauxType=="DBMS" && regexp("random.*", Name, "i")'
```

MyType	TargetType	Name
Generic	None	random-test

Similar flexibility exists in querying job queues in the Condor system.

4.2 Condor's Checkpoint Mechanism

Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to no longer allocate a machine to a job (for example, when the owner of that machine returns), it can checkpoint the job and preempt it without losing the work the job has already accomplished. The job can be resumed later when the scheduler allocates it a new machine. Additionally, periodic checkpointing provides fault tolerance in Condor. Snapshots are taken periodically, and after an interruption in service the program can continue from the most recent snapshot.

Condor provides checkpointing services to single process jobs on a number of Unix platforms. To enable checkpointing, the user must link the program with the Condor system call library (`libcondorsyscall.a`), using the `condor_compile` command. This means that the user must have the object files or source code of the program to use Condor checkpointing. However, the checkpointing services provided by Condor are strictly optional. So, while there are some classes of jobs for which Condor does not provide checkpointing services, these jobs may still be submitted to Condor to take advantage of Condor's resource management functionality. (See section 2.4.1 on page 16 for a description of the classes of jobs for which Condor does not provide checkpointing services.)

Process checkpointing is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal handler writes the state of the process out to a file or a network socket. This state includes the contents of the process stack and data segments, all shared library code and data mapped into the process's address space, the state of all open files, and any signal handlers and pending signals. On restart, the process reads this state from the file, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then returns to user code, which continues from where it left off when the checkpoint signal arrived.

Condor processes for which checkpointing is enabled perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off. Jobs that can not be checkpointed are preempted and restarted from the beginning.

Condor's periodic checkpointing provides fault tolerance. Condor pools are each configured with the `PERIODIC_CHECKPOINT` expression which controls when and how often jobs which can be checkpointed do periodic checkpoints (examples: never, every three hours, etc.). When the time for a periodic checkpoint occurs, the job suspends processing, performs the checkpoint, and immediately continues from where it left off. There is also a `condor_ckpt` command which allows the user to request that a Condor job immediately perform a periodic checkpoint.

In all cases, Condor jobs continue execution from the most recent complete checkpoint. If service is interrupted while a checkpoint is being performed, causing that checkpoint to fail, the process will restart from the previous checkpoint. Condor uses a commit style algorithm for writing checkpoints: a previous checkpoint is deleted only after a new complete checkpoint has been written

successfully.

In certain cases, checkpointing may be delayed until a more appropriate time. For example, a Condor job will defer a checkpoint request if it is communicating with another process over the network. When the network connection is closed, the checkpoint will occur.

The Condor checkpointing facility can also be used for any Unix process outside of the Condor batch environment. Standalone checkpointing is described in section 4.2.1.

Condor can produce and use compressed checkpoints. Configuration variables (detailed in section 3.3.12) control whether compression is used. The default is to not compress.

By default, a checkpoint is written to a file on the local disk of the machine where the job was submitted. A Condor pool can also be configured with a checkpoint server or servers that serve as a repository for checkpoints. (See section 3.8 on page 346.) When a host is configured to use a checkpoint server, jobs submitted on that machine write and read checkpoints to and from the server rather than the local disk of the submitting machine, taking the burden of storing checkpoint files off of the submitting machines and placing it instead on server machines (with disk space dedicated to the purpose of storing checkpoints).

4.2.1 Standalone Checkpointing

Using the Condor checkpoint library without the remote system call functionality and outside of the Condor system is known as standalone mode checkpointing.

To prepare a program for standalone checkpointing, simply use the *condor_compile* utility as for a standard Condor job, but do not use *condor_submit* – run the program normally from the command line. The checkpointing library will print a message to let you know that checkpointing is enabled and to inform you of the default name for the checkpoint image. The message is of the form:

```
Condor: Notice: Will checkpoint to program_name.ckpt
Condor: Notice: Remote system calls disabled.
```

To force the program to write a checkpoint image and stop, send it the SIGTSTP signal or press control-Z. To force the program to write a checkpoint image and continue executing, send it the SIGUSR2 signal.

To restart a program using a checkpoint, run the program with the argument *-_condor_restart* followed by the name of the checkpoint image file. As an example, if the program is called *P1* and the checkpoint is called *P1.ckpt*, use

```
P1 -_condor_restart P1.ckpt
```

4.2.2 Checkpoint Safety

Some programs have fundamental limitations that make them unsafe for checkpointing. For example, a program that both reads and writes a single file may enter an unexpected state. Here is an example of how this might happen.

1. Record a checkpoint image.
2. Read data from a file.
3. Write data to the same file.
4. Execution failure, so roll back to step 2.

In this example, the program would re-read data from the file, but instead of finding the original data, would see data created in the future, and yield unexpected results.

To prevent this sort of accident, Condor displays a warning if a file is used for both reading and writing. You can ignore or disable these warnings if you choose (see section 4.2.3,) but please understand that your program may compute incorrect results.

4.2.3 Checkpoint Warnings

Condor has warning messages in the case unexpected behaviors in your program. For example, if file `x` is opened for reading and writing, you will see:

```
Condor: Warning: READWRITE: File '/tmp/x' used for both reading and writing.
```

You may control how these messages are displayed with the `-_condor_warning` command-line argument. This argument accepts a warning category and a mode. The category describes a certain class of messages, such as `READWRITE` or `ALL`. The mode describes what to do with the category. It may be `ON`, `OFF`, or `ONCE`. If a category is `ON`, it is always displayed. If a category is `OFF`, it is never displayed. If a category is `ONCE`, it is displayed only once. To show all the available categories and modes, just use `-_condor_warning` with no arguments.

For example, to limit read/write warnings to one instance:

```
-_condor_warning READWRITE ONCE
```

To turn all ordinary notices off:

```
-_condor_warning NOTICE OFF
```

The same effect can be accomplished within a program by using the function `_condor_warning_config`, described in section 4.2.4.

4.2.4 Checkpoint Library Interface

A program need not be rewritten to take advantage of checkpointing. However, the checkpointing library provides several C entry points that allow for a program to control its own checkpointing behavior if needed.

- `void init_image_with_file_name(char *ckpt_file_name)`
This function explicitly sets a file name to use when producing or using a checkpoint. `ckpt()` or `ckpt_and_exit()` must be called to produce the checkpoint, and `restart()` must be called to perform the actual restart.
- `void init_image_with_file_descriptor(int fd)`
This function explicitly sets a file descriptor to use when producing or using a checkpoint. `ckpt()` or `ckpt_and_exit()` must be called to produce the checkpoint, and `restart()` must be called to perform the actual restart.
- `void ckpt()`
This function causes a checkpoint image to be written to disk. The program will continue to execute. This is identical to sending the program a SIGUSR2 signal.
- `void ckpt_and_exit()`
This function causes a checkpoint image to be written to disk. The program will then exit. This is identical to sending the program a SIGTSTP signal.
- `void restart()`
This function causes the program to read the checkpoint image and to resume execution of the program from the point where the checkpoint was taken. This function does not return.
- `void _condor_ckpt_disable()`
This function temporarily disables checkpointing. This can be handy if your program does something that is not checkpoint-safe. For example, if a program must not be interrupted while accessing a special file, call `_condor_ckpt_disable()`, access the file, and then call `_condor_ckpt_enable()`. Some program actions, such as opening a socket or a pipe, implicitly cause checkpointing to be disabled.
- `void _condor_ckpt_enable()`
This function re-enables checkpointing after a call to `_condor_ckpt_disable()`. If a checkpointing signal arrived while checkpointing was disabled, the checkpoint will occur when this function is called. Disabling and enabling of checkpointing must occur in matched pairs. `_condor_ckpt_enable()` must be called once for every time that `_condor_ckpt_disable()` is called.
- `int _condor_warning_config(const char *kind, const char *mode)`
This function controls what warnings are displayed by Condor. The `kind` and `mode` arguments are the same as for the `-condor_warning` option described in section 4.2.3. This function returns true if the arguments are understood and accepted. Otherwise, it returns false.

- `extern int condor_compress_ckpt`
Setting this variable to one causes checkpoint images to be compressed. Setting it to zero disables compression.

4.3 Computing On Demand (COD)

Computing On Demand (COD) extends Condor's high throughput computing abilities to include a method for running short-term jobs on instantly-available resources.

The motivation for COD extends Condor's job management to include interactive, compute-intensive jobs, giving these jobs immediate access to the compute power they need over a relatively short period of time. COD provides computing power *on demand*, switching predefined resources from working on Condor jobs to working on the COD jobs. These COD jobs (applications) cannot use the batch scheduling functionality of Condor, since the COD jobs require interactive response-time. Many of the applications that are well-suited to Condor's COD capabilities involve a cycle: application blocked on user input, computation burst to compute results, block again on user input, computation burst, etc. When the resources are not being used for the bursts of computation to service the application, they should continue to execute long-running batch jobs.

Here are examples of applications that may benefit from COD capability:

- A giant spreadsheet with a large number of highly complex formulas which take a lot of compute power to recalculate. The spreadsheet application (as a COD application) predefines a claim on resources within the Condor pool. When the user presses a `recalculate` button, the predefined Condor resources (nodes) work on the computation and send the results back to the master application providing the user interface and displaying the data. Ideally, while the user is entering new data or modifying formulas, these nodes work on non-COD jobs.
- A graphics rendering application that waits for user input to select an image to render. The rendering requires a huge burst of computation to produce the image. Examples are various Computer-Aided Design (CAD) tools, fractal rendering programs, and ray-tracing tools.
- Visualization tools for data mining.

The way Condor helps these kinds of applications is to provide an infrastructure to use Condor batch resources for the types of compute nodes described above. Condor does *NOT* provide tools to parallelize existing GUI applications. The COD functionality is an interface to allow these compute nodes to interact with long-running Condor batch jobs. The user provides both the compute node applications and the interactive master application that controls them. Condor only provides a mechanism to allow these interactive (and often parallelized) applications to seamlessly interact with the Condor batch system.

4.3.1 Overview of How COD Works

The resources of a Condor pool (nodes) run jobs. When a high-priority COD job appears at a node, the lower-priority (currently running) batch job is suspended. The COD job runs immediately, while the batch job remains suspended. When the COD job completes, the batch job instantly resumes execution.

Administratively, an interactive COD application puts claims on nodes. While the COD application does not need the nodes (to run the COD jobs), the claims are suspended, allowing batch jobs to run.

4.3.2 Authorizing Users to Create and Manage COD Claims

Claims on nodes are assigned to users. A user with a claim on a resource can then suspend and resume a COD job at will. This gives the user a great deal of power on the claimed resource, even if it is owned by another user. Because of this, it is essential that users allowed to claim COD resources can be trusted not to abuse this power. Users are authorized to have access to the privilege of creating and using a COD claim on a machine. This privilege is granted when the Condor administrator places a given user name in the `VALID_COD_USERS` list in the Condor configuration for the machine (usually in a local configuration file).

In addition, the tools to request and manage COD claims require that the user issuing the commands be authenticated. Use one of the strong authentication methods described in section 3.6.1 “Security Configuration” on page 282. If one of these methods cannot be used, then file system authentication may be used when directly logging in to that machine (to be claimed) and issuing the command locally.

4.3.3 Defining a COD Application

To run an application on a claimed COD resource, an authorized user defines characteristics of the application. Examples of characteristics are the executable or script to use, the directory to run the application in, command-line arguments, and files to use for standard input and output. COD users specify a ClassAd that describes these characteristics for their application. There are two ways for a user to define a COD application’s ClassAd:

1. in the Condor configuration files of the COD resources
2. when they use the *condor_cod* command-line tool to launch the application itself

These two methods for defining the ClassAd can be used together. For example, the user can define some attributes in the configuration file, and only provide a few dynamically defined attributes with the *condor_cod* tool.

Regardless of how the COD application's ClassAd is defined, the application's executable and input data must be pre-staged at the node. This is a current limitation of Condor's support for COD that will eventually go away. For now, there is no mechanism to transfer files for a COD application, and all I/O must be performed locally or onto a network file system that is accessible by a node.

The following three sections detail defining the attributes. The first lists the attributes that can be used to define a COD application. The second describes how to define these attributes in a Condor configuration file. The third explains how to define these attributes using the *condor_cod* tool.

COD Application Attributes

Attributes for a COD application are either required or optional. The following attributes are *required*:

Cmd This attribute defines the full path to the executable program to be run as a COD application. Since Condor does not currently provide any mechanism to transfer files on behalf of COD applications, this path should be a valid path on the machine where the application will be run. It is a string attribute, and must therefore be enclosed in quotation marks ("). There is no default.

Owner If the *condor_startd* daemon is executing as root on the resource where a COD application will run, the user must also define *Owner* to specify what user name the application will run as. (On Windows, the *condor_startd* daemon always runs as an Administrator service, which is equivalent to running as root on UNIX platforms). If the user specifies any COD application attributes with the *condor_cod_activate* command-line tool, the *Owner* attribute will be defined as the user name that ran *condor_cod_activate*. However, if the user defines all attributes of their COD application in the Condor configuration files, and does not define any attributes with the *condor_cod_activate* command-line tool (both methods are described below in more detail), there is no default and *Owner* must be specified in the configuration file. *Owner* must contain a valid user name on the given COD resource. It is a string attribute, and must therefore be enclosed in quotation marks (").

The following list of attributes are *optional*:

IWD IWD is an acronym for Initial Working Directory. It defines the full path to the directory where a given COD application are to be run. Unless the application changes its current working directory, any relative path names used by the application will be relative to the IWD. If any other attributes that define file names (for example, *In*, *Out*, and so on) do not contain a full path, the IWD will automatically be pre-pended to those filenames. It is a string attribute, and must therefore be enclosed in quotation marks ("). If the IWD is not specified, the temporary execution sandbox created by the *condor_starter* will be used as the initial working directory.

In This string defines the path to the file on the COD resource that should be used as standard input (*stdin*) for the COD application. This file (and all parent directories) must be readable by whatever user the COD application will run as. If not specified, the default is */dev/null*.

- Out** This string defines the path to the file on the COD resource that should be used as standard output (`stdout`) for the COD application. This file must be writable (and all parent directories readable) by whatever user the COD application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).
- Err** This string defines the path to the file on the COD resource that should be used as standard error (`stderr`) for the COD application. This file must be writable (and all parent directories readable) by whatever user the COD application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).
- Env** This string defines environment variables to set for a given COD application. Each environment variable has the form `NAME=value`. Multiple variables are delimited with a semi-colon. An example: `Env = "PATH=/usr/local/bin:/usr/bin;TERM=vt100"` It is a string attribute, and must therefore be enclosed in quotation marks (`"`).
- Args** This string attribute defines the list of arguments to be supplied to the program on the command-line. The arguments are delimited (separated) by space characters. There is no default. If the `JobUniverse` corresponds to the Java universe, the first argument must be the name of the class containing `main`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).
- JobUniverse** This attribute defines what Condor job universe to use for the given COD application. At this point, the only supported universes are vanilla and Java. This attribute must be an integer, with vanilla using the value 5, and Java the value 10. If `JobUniverse` is not specified, the vanilla universe is used by default. For more information about the Condor job universes, see section 2.4.1 on page 15.
- JarFiles** This string attribute is only used if `JobUniverse` is 10 (the Java universe). If a given COD application is a Java program, specify the JAR files that the program requires with this attribute. There is no default. It is a string attribute, and must therefore be enclosed in quotation marks (`"`). Multiple file names may be delimited with either commas or whitespace characters, and therefore, file names can not contain spaces.
- KillSig** This attribute specifies what signal should be sent whenever the Condor system needs to gracefully shutdown the COD application. It can either be specified as a string containing the signal name (for example `KillSig = "SIGQUIT"`), or as an integer (`KillSig = 3`) The default is to use `SIGTERM`.
- StarterUserLog** This string specifies a file name for a log file that the `condor_starter` daemon can write with entries for relevant events in the life of a given COD application. It is similar to the `UserLog` file specified for regular Condor jobs with the `Log` setting in a submit description file. However, certain attributes that are placed in the regular `UserLog` file do not make sense in the COD environment, and are therefore omitted. The default is not to write this log file. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).
- StarterUserLogUseXML** If the `StarterUserLog` attribute is defined, the default format is a human-readable format. However, Condor can write out this log in an XML representation, instead. To enable the XML format for this `UserLog`, the `StarterUserLogUseXML` boolean is set to `TRUE`. The default if not specified is `FALSE`.

NOTE: If any path attribute (*Cmd*, *In*, *Out*, *Err*, *StarterUserLog*) is not a full path name, Condor automatically prepends the value of *IWD*.

The final set of attributes define an identification for a COD application. The job ID is made up of both the *ClusterId* and *ProcId* attributes (as described below). This job ID is similar to the job ID that is created whenever a regular Condor batch job is submitted. For regular Condor batch jobs, the job ID is assigned automatically by the *condor_schedd* whenever a new job is submitted into the persistent job queue. However, since there is no persistent job queue for COD, the usual mechanism to identify the jobs does not exist. Moreover, commands that require the job ID for batch jobs such as *condor_q* and *condor_rm* do not exist for COD. Instead, the claim ID is the unique identifier for COD jobs and COD-related commands.

When using COD, the job ID is only used to identify the job in various log messages and in the COD-specific output of *condor_status*. The COD job ID is part of the information included in all events written to the *StarterUserLog* regarding a given job. The COD job ID is also used in the Condor debugging logs described in section 3.3.4 on page 157. For example, in the *condor_starter* daemon's log file for COD jobs (called *StarterLog.cod* by default) or in the *condor_startd* daemon's log file (called *StartLog* by default).

These COD IDs are optional. The job ID is useful to define where it helps a user with accounting or debugging of their own application. In this case, it is the user's responsibility to ensure uniqueness, if so desired.

ClusterId This integer defines the cluster identifier for a COD job. The default value is 1. The *ClusterId* can also be defined with the *condor_cod_activate* command-line tool using the **-cluster** option.

ProcId This integer defines the process identifier (within a cluster) for a COD job. The default value is 0. The *ProcId* can also be defined with the *condor_cod_activate* command-line tool using the **-cluster** option.

NOTE: The cluster and proc identifiers can also be specified as command-line arguments to the *condor_cod_activate* tool when spawning a given COD application. See section 4.3.4 below for details on using *condor_cod_activate*.

Defining Attributes in the Condor Configuration Files

To define COD attributes in the Condor configuration file for a given application, the user selects a keyword to uniquely name ClassAd attributes of the application. This case-insensitive keyword is used as a prefix for the various configuration file attribute names. When a user wishes to spawn a given application, the keyword is given as an argument to the *condor_cod* tool and the keyword is used at the remote COD resource to find attributes which define the application.

Any of the ClassAd attributes described in the previous section can be specified in the configuration file with the keyword prefix followed by an underscore character ("_").

For example, if the user's keyword for a given fractal generation application is "FractGen", the resulting entries in the Condor configuration file may appear as:

```
FractGen_Cmd = "/usr/local/bin/fractgen"
FractGen_Iwd = "/tmp/cod-fractgen"
FractGen_Out = "/tmp/cod-fractgen/output"
FractGen_Err = "/tmp/cod-fractgen/error"
FractGen_Args = "mandelbrot -0.65865,-0.56254 -0.45865,-0.71254"
```

In this example, the executable may create other files. The `Out` and `Err` attributes specified in the configuration file are only for standard output and standard error redirection.

When the user wishes to spawn an instance of this application, they use the **-keyword** option of **FractGen** in the command-line of the *condor_cod_activate* command.

NOTE: If a user is defining all attributes of their COD application in the Condor configuration files, and the *condor_startd* daemon on the COD resource they are using is running as root, the user must also define `Owner` to be the user that the COD application should run as (see section 4.3.3 above).

Defining Attributes with the *condor_cod* Tool

COD users may define attributes dynamically (at the time they spawn a COD application). In this case, the user writes the ClassAd attributes into a file, and the file name is passed to the *condor_cod_activate* tool using the **-jobad** command-line option. These attributes are read by the *condor_cod* tool and passed through the system onto the *condor_starter* daemon which spawns the COD application. If the file name given is `-`, the *condor_cod* tool will read from standard input (`stdin`).

Users should not add a keyword prefix when defining attributes with the *condor_cod_activate* tool. The attribute names can be used in the file directly.

WARNING: The current syntax for this file is not the same as the syntax in the file used with *condor_submit*.

NOTE: Users should not define the `Owner` attribute when using *condor_cod_activate* on the command line, since Condor will automatically insert the correct value based on what user runs the *condor_cod_activate* command and how that user authenticates to the COD resource. If a user defines an attribute that does not match the authenticated identity, Condor treats this case as an error, and it will fail to launch the application.

4.3.4 Managing COD Resource Claims

Separate commands are provided by Condor to manage COD claims on batch resources. Once created, each COD claim has a unique identifying string, called the claim ID. Most commands

require a claim ID to specify which claim you wish to act on. These commands are the means by which COD applications interact with the rest of the Condor system. They should be issued by the controller application to manage its compute nodes. Here is a list of the commands:

Request Create a new COD claim on a given resource.

Activate Spawn a specific application on a specific COD claim.

Suspend Suspend a running application within a specific COD claim.

Renew Renew the lease to a COD claim.

Resume Resume a suspended application on a specific COD claim.

Deactivate Shut down an application, but hold onto the COD claim for future use.

Release Destroy a specific COD claim, and shut down any job that is currently running on it.

Delegate proxy Send an x509 proxy credential to the specific COD claim (optional, only required in rare cases like using glxexec to spawn the *condor_starter* at the execute machine where the COD job is running).

To issue these commands, a user or application invokes the *condor_cod* tool. A command may be specified as the first argument to this tool, as

```
condor_cod request -name c02.cs.wisc.edu
```

or the *condor_cod* tool can be installed in such a way that the same binary is used for a set of names, as

```
condor_cod_request -name c02.cs.wisc.edu
```

Other than the command name itself (which must be included in full) additional options supported by each tool can be abbreviated to the shortest unambiguous value. For example, **-name** can also be specified as **-n**. However, for a command like *condor_cod_activate* that supports both **-classad** and **-cluster**, the user must use at least **-cla** or **-clu**. If the user specifies an ambiguous option, the *condor_cod* tool will exit with an error message.

In addition, there is now a **-cod** option to *condor_status*.

The following sections describe each option in greater detail.

Request

A user must be granted authorization to create COD claims on a specific machine. In addition, when the user uses these COD claims, the application binary or script they wish to run (and any input

data) must be pre-staged on the machine. Therefore, a user cannot simply request a COD claim at random.

The user specifies the resource on which to make a COD claim. This is accomplished by specifying the name of the *condor_startd* daemon desired by invoking *condor_cod_request* with the **-name** option and the resource name (usually the host name). For example:

```
condor_cod_request -name c02.cs.wisc.edu
```

If the *condor_startd* daemon desired belongs to a different Condor pool than the one where executing the COD commands, use the **-pool** option to provide the name of the central manager machine of the other pool. For example:

```
condor_cod_request -name c02.cs.wisc.edu -pool condor.cs.wisc.edu
```

An alternative is to provide the IP address and port number where the *condor_startd* daemon is listening with the **-addr** option. This information can be found in the *condor_startd* ClassAd as the attribute *StartdIpAddr* or by reading the log file when the *condor_startd* first starts up. For example:

```
condor_cod_request -addr "<128.105.146.102:40967>"
```

If neither **-name** or **-addr** are specified, *condor_cod_request* attempts to connect to the *condor_startd* daemon running on the local machine (where the request command was issued).

If the *condor_startd* daemon to be used for the COD claim is an SMP machine and has multiple slots, specify which resource on the machine to use for COD by providing the full name of the resource, not just the host name. For example:

```
condor_cod_request -name slot2@c02.cs.wisc.edu
```

A constraint on what slot is desired may be provided, instead of specifying it by name. For example, to run on machine c02.cs.wisc.edu, not caring which slot is used, so long as it the machine is not currently running a job, use something like:

```
condor_cod_request -name c02.cs.wisc.edu -requirements 'State!="Claimed"'
```

In general, be careful with shell quoting issues, so that your shell is not confused by the ClassAd expression syntax (in particular if the expression includes a string). The safest method is to enclose any requirement expression within single quote marks (as shown above).

Once a given *condor_startd* daemon has been contacted to request a new COD claim, the *condor_startd* daemon checks for proper authorization of the user issuing the command. If the user has the authority, and the *condor_startd* daemon finds a resource that matches any given requirements,

the *condor_startd* daemon creates a new COD claim and gives it a unique identifier, the claim ID. This ID is used to identify COD claims when using other commands. If *condor_cod_request* succeeds, the claim ID for the new claim is printed out to the screen. All other commands to manage this claim require the claim ID to be provided as a command-line option.

When the *condor_startd* daemon assigns a COD claim, the ClassAd describing the resource is returned to the user that requested the claim. This ClassAd is a snap-shot of the output of *condor_status -long* for the given machine. If *condor_cod_request* is invoked with the **-classad** option (which takes a file name as an argument), this ClassAd will be written out to the given file. Otherwise, the ClassAd is printed to the screen. The only essential piece of information in this ClassAd is the Claim ID, so that is printed to the screen, even if the whole ClassAd is also being written to a file.

The claim ID as given after listing the machine ClassAd appears as this example:

```
ID of new claim is: "<128.105.121.21:49973>#1073352104#4"
```

When using this claim ID in further commands, include the quote marks as well as all the characters in between the quote marks.

NOTE: Once a COD claim is created, there is no persistent record of it kept by the *condor_startd* daemon. So, if the *condor_startd* daemon is restarted for any reason, all existing COD claims will be destroyed and the new *condor_startd* daemon will not recognize any attempts to use the previous claims.

Also note that it is your responsibility to ensure that the claim is eventually removed (see section 4.3.4). Failure to remove the COD claim will result in the *condor_startd* continuing to hold a record of the claim for as long as *condor_startd* continues running. If a very large number of such claims are accumulated by the *condor_startd*, this can impact its performance. Even worse: if a COD claim is unintentionally left in an activated state, this results in the suspension of any batch job running on the same resource for as long as the claim remains activated. For this reason, an optional **-lease** argument is supported by *condor_cod_request*. This tells the *condor_startd* to automatically release the COD claim after the specified number of seconds unless the lease is renewed with *condor_cod_renew*. The default lease is infinitely long.

Activate

Once a user has created a valid COD claim and has the claim ID, the next step is to spawn a COD job using the claim. The way to do this is to activate the claim, using the *condor_cod_activate* command. Once a COD application is active on a COD claim, the COD claim will move into the Running state, and any batch Condor job on the same resource will be suspended. Whenever the COD application is inactive (either suspended, removed from the machine, or if it exits on its own), the state of the COD claim changes. The new state depends on why the application became inactive. The batch Condor job then resumes.

To activate a COD claim, first define attributes about the job to be run in either the local configuration of the COD resource, or in a separate file as described in this manual section. Invoke

the *condor_cod_activate* command to launch a specific instance of the job on a given COD claim ID. The options given to *condor_cod_activate* vary depending on if the job attributes are defined in the configuration file or are passed via a file to the *condor_cod_activate* tool itself. However, the **-id** option is always required by *condor_cod_activate*, and this option should be followed by a COD claim ID that the user acquired via *condor_cod_request*.

If the application is defined in the configuration files for the COD resource, the user provides the keyword (described in section 4.3.3) that uniquely identifies the application's configuration attributes. To continue the example from that section, the user would spawn their job by specifying **-keyword FractGen**, for example:

```
condor_cod_activate -id "<claim_id>" -keyword FractGen
```

Substitute the *<claim_id>* with the valid Cod Claim Id. Using the same example as given above, this example would be:

```
condor_cod_activate -id "<128.105.121.21:49973>#1073352104#4" -keyword FractGen
```

If the job attributes are placed into a file to be passed to the *condor_cod_activate* tool, the user must provide the name of the file using the **-jobad** option. For example, if the job attributes were defined in a file named *cod-fractgen.txt*, the user spawns the job using the command:

```
condor_cod_activate -id "<claim_id>" -jobad cod-fractgen.txt
```

Alternatively, if the filename specified with **-jobad** is **-**, the *condor_cod_activate* tool reads the job ClassAd from standard input (stdin).

Regardless of how the job attributes are defined, there are other options that *condor_cod_activate* accepts. These options specify the job ID for the application to be run. The job ID can either be specified in the job's ClassAd, or it can be specified on the command line to *condor_cod_activate*. These options are **-cluster** and **-proc**. For example, to launch a COD job with keyword *foo* as cluster 23, proc 5, or 23.5, the user invokes:

```
condor_cod_activate -id "<claim_id>" -key foo -cluster 23 -proc 5
```

The **-cluster** and **-proc** arguments are optional, since the job ID is not required for COD. If not specified, the job ID defaults to 1 . 0.

Suspend

Once a COD application has been activated with *condor_cod_activate* and is running on a COD resource, it may be temporarily suspended using *condor_cod_suspend*. In this case, the claim state becomes Suspended. Once a given COD job is suspended, if there are no other running COD jobs on the resource, a Condor batch job can use the resource. By suspending the COD application, the

batch job is allowed to run. If a resource is idle when a COD application is first spawned, suspension of the COD job makes the batch resource available for use in the Condor system. Therefore, whenever a COD application has no work to perform, it should be suspended to prevent the resource from being wasted.

The interface of *condor_cod_suspend* supports the single option **-id**, to specify the COD claim ID to be suspended. For example:

```
condor_cod_suspend -id "<claim_id>"
```

If the user attempts to suspend a COD job that is not running, *condor_cod_suspend* exits with an error message. The COD job may not be running because it is already suspended or because the job was never spawned on the given COD claim in the first place.

Renew

This command tells the *condor_startd* to renew the lease on the COD claim for the amount of lease time specified when the claim was created. See section 4.3.4 for more information on using leases.

The *condor_cod_renew* tool supports only the **-id** option to specify the COD claim ID the user wishes to renew. For example:

```
condor_cod_renew -id "<claim_id>"
```

If the user attempts to renew a COD job that no longer exists, *condor_cod_renew* exits with an error message.

Resume

Once a COD application has been suspended with *condor_cod_suspend*, it can be resumed using *condor_cod_resume*. In this case, the claim state returns to Running. If there is a regular batch job running on the same resource, it will automatically be suspended if a COD application is resumed.

The *condor_cod_resume* tool supports only the **-id** option to specify the COD claim ID the user wishes to resume. For example:

```
condor_cod_resume -id "<claim_id>"
```

If the user attempts to resume a COD job that is not suspended, *condor_cod_resume* exits with an error message.

Deactivate

If a given COD application does not exit on its own and needs to be removed manually, invoke the *condor_cod_deactivate* command to kill the job, but leave the COD claim ID valid for future COD jobs. The user must specify the claim ID they wish to deactivate using the **-id** option. For example:

```
condor_cod_deactivate -id "<claim_id>"
```

By default, *condor_cod_deactivate* attempts to gracefully cleanup the COD application and give it time to exit. In this case the COD claim goes into the *Vacating* state and the *condor_starter* process controlling the job will send it the *KillSig* defined for the job (*SIGTERM* by default). This allows the COD job to catch the signal and do whatever final work is required to exit cleanly.

However, if the program is stuck or if the user does not want to give the application time to clean itself up, the user may use the **-fast** option to tell the *condor_starter* to quickly kill the job and all its descendants using *SIGKILL*. In this case the COD claim goes into the *Killing* state. For example:

```
condor_cod_deactivate -id "<claim_id>" -fast
```

In either case, once the COD job has finally exited, the COD claim will go into the *Idle* state and will be available for future COD applications. If there are no other active COD jobs on the same resource, the resource would become available for batch Condor jobs. Whenever the user wishes to spawn another COD application, they can reuse this idle COD claim by using the same claim ID, without having to go through the process of running *condor_cod_request*.

If the user attempts a *condor_cod_deactivate* request on a COD claim that is neither *Running* nor *Suspended*, the *condor_cod* tool exits with an error message.

Release

If users no longer wish to use a given COD claim, they can release the claim with the *condor_cod_release* command. If there is a COD job running on the claim, the job will first be shut down (as if *condor_cod_deactivate* was used), and then the claim itself is removed from the resource and the claim ID is destroyed. Further attempts to use the claim ID for any COD commands will fail.

The *condor_cod_release* command always prints out the state the COD claim was in when the request was received. This way, users can know what state a given COD application was in when the claim was destroyed.

Like most COD commands, *condor_cod_release* requires the claim ID to be specified using **-id**. In addition, *condor_cod_release* supports the **-fast** option (described above in the section about *condor_cod_deactivate*). If there is a job running or suspended on the claim when it is released with *condor_cod_release -fast*, the job will be immediately killed. If **-fast** is not specified, the

default behavior is to use a graceful shutdown, sending whatever signal is specified in the `KillSig` attribute for the job (SIGTERM by default).

Delegate proxy

In some cases, a user will want to delegate a copy of their user credentials (in the form of an x509 proxy) to the machine where one of their COD jobs will run. For example, sites wishing to spawn the *condor_starter* using *glxexec* will need a copy of this credential before the claim can be activated. Therefore, beginning with Condor version 6.9.2, COD users have access to the command *delegate_proxy*. If users do not specifically require this proxy delegation, this command should not be used and the rest of this section can be skipped.

The *delegate_proxy* command optionally takes a **-x509proxy** argument to specify the path to the proxy file to use. Otherwise, it uses the same discovery logic that *condor_submit* uses to find the user's currently active proxy.

Just like every other COD command (except *request*), this command requires a valid COD claim id (specified with **-id**) to indicate what COD claim you wish to delegate the credentials to.

This command can only be sent to idle COD claims, so it should be done before *activate* is run for the first time. However, once a proxy has been delegated, it can be reused by successive claim activations, so normally this step only has to happen once, not before every *activate*. If a proxy is going to expire, and a new one should be sent, this should only happen after the existing COD claim has been deactivated.

4.3.5 Limitations of COD Support in Condor

Condor's support for COD has a few limitations.

The following items are all limitations we plan to remove in future releases of Condor:

- Applications and data must be pre-staged at a given machine.
- There is no way to define limits for how long a given COD claim can be active, how often it is run, and so on.
- There is no accounting done for applications run under COD claims. Therefore, use of a lot of COD resources in a given Condor pool does not adversely affect user priority.

None of the above items are fundamentally difficult to add and we hope to address them relatively quickly. If you run into one of these limitations, and it is a barrier to using COD, please contact condor-admin@cs.wisc.edu with the subject "COD limitation" to gain quick help.

The following list are more fundamental limitations that we do not plan to address:

- COD claims are not persistent on a given *condor_startd* daemon.
- Condor does not provide a mechanism to parallelize a graphic application to take advantage of COD. The Condor Team is not in the business of developing applications, we only provide mechanisms to execute them.

4.4 Job Hooks

A *hook* is an external program or script invoked by Condor at various points during the life cycle of a job. Instead of putting all the code and logic directly into the Condor daemons to handle the variety of external systems from which it might fetch work, sites can write their own programs or scripts and allow Condor to invoke these hooks at the right moments to accomplish the desired outcome. This eliminates the expense of the matchmaking and scheduling provided by the *condor_schedd* and the *condor_negotiator*, although at the price of the flexibility they offer. Therefore, the hooks allow Condor to more easily and directly interface with external scheduling systems.

The following sections describe the system of hooks, both for work that is fetched from an external system, and for hooks that behave as a Job Router.

4.4.1 Hooks that Fetch Work

In the past, Condor has always sent work to the execute machines by pushing jobs to the *condor_startd* daemon, either from the *condor_schedd* daemon or via *condor_cod*. Beginning with the Condor 7.1.0, the *condor_startd* daemon now has the ability to pull work by fetching jobs via a system of plug-ins or hooks. Any site can configure a set of hooks to fetch work completely outside of the usual Condor matchmaking system.

A projected use of the hook mechanism implements what might be termed a *glide-in factory*, especially where the factory is behind a firewall. Without using the hook mechanism to fetch work, a glide-in *condor_startd* daemon behind a firewall depends on GCB to help it listen and eventually receive work pushed from elsewhere. With the hook mechanism, a glide-in *condor_startd* daemon behind a firewall uses the hook to pull work. The hook needs only an outbound network connection to complete its task, thereby being able to operate from behind the firewall, without the intervention of GCB.

Periodically, each execution slot managed by a *condor_startd* will invoke a hook to see if there is any work that can be fetched. Whenever this hook returns a valid job, the *condor_startd* will evaluate the current state of the slot and decide if it should start executing the fetched work. If the slot is unclaimed and the *Start* expression evaluates to TRUE, a new claim will be created for the fetched job. If the slot is claimed, the *condor_startd* will evaluate the *Rank* expression relative to the fetched job and compare it to the value of the *Rank* for the currently running job and decide if the existing job should be preempted due to the fetched job having a higher rank. If the slot is unavailable for whatever reason, the *condor_startd* will refuse the fetched job and ignore it. Either

way, once the *condor_startd* decides what it should do with the fetched job, it will invoke another hook to reply to the attempt to fetch work, so that the external system knows what happened to that work unit.

If the job is accepted, a claim is created for it and the slot moves into the Claimed state. As soon as this happens, the *condor_startd* will spawn a *condor_starter* to manage the execution of the job. At this point, from the perspective of the *condor_startd*, this claim is just like any other. The usual policy expressions are evaluated, and if the job needs to be suspended or evicted, it will be. If a higher-ranked job being managed by a *condor_schedd* is matched with the slot, that job will preempt the fetched work.

The *condor_starter* itself can optionally invoke additional hooks to help manage the execution of the specific job. There are hooks to prepare the execution environment for the job, periodically update information about the job as it runs, notify when the job exits, and to take special actions when the job is being evicted.

Assuming there are no interruptions, the job completes, and the *condor_starter* exits, the *condor_startd* will invoke the hook to fetch work again. If another job is available, the existing claim will be reused and a new *condor_starter* is spawned. If the hook returns that there is no more work to perform, the claim will be evicted, and the slot will return to the Owner state.

Work Fetching Hooks Invoked by Condor

There are a handful of hooks invoked by Condor related to fetching work, some of which are called by the *condor_startd* and others by the *condor_starter*. Each hook will be described below, including when it is invoked, what task it is supposed to accomplish, what data is passed to the hook, and what output (and, when relevant) exit status is expected.

Hook: Fetch Work The hook defined by the configuration variable `HOOK_FETCH_WORK` is invoked whenever the *condor_startd* wants to see if there is any work to fetch. There is a related configuration expression called `FetchWorkDelay` which determines how long the *condor_startd* will wait between attempts to fetch work, which is described in detail in within section 4.4.1 on page 465. `HOOK_FETCH_WORK` is the most important hook in the whole system, and is the only hook that must be defined for any of the other *condor_startd* hooks to operate.

The job ClassAd returned by the hook needs to contain enough information for the *condor_starter* to eventually spawn the work. The required and optional attributes in this ClassAd are identical to the ones described for Computing on Demand (COD) jobs in section 4.3.3 called COD Application Attributes on page 449.

Command-line arguments passed to the hook None.

Standard input given to the hook ClassAd of the slot that is looking for work.

Expected standard output from the hook ClassAd of a job that can be run. If there is no work, the hook should return no output.

Exit status of the hook Ignored.

Hook: Reply Fetch The hook defined by the configuration variable `HOOK_REPLY_FETCH` is invoked whenever `HOOK_FETCH_WORK` returns data and the `condor_startd` decides if it is going to accept the fetched job or not.

The `condor_startd` will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the `condor_startd`.

Command-line arguments passed to the hook Either the string `accept` or `reject`.

Standard input given to the hook A copy of the job ClassAd and the slot ClassAd (separated by the string `-----` and a new line).

Expected standard output from the hook None.

Exit status of the hook Ignored.

Hook: Evict Claim The hook defined by the configuration variable `HOOK_EVICT_CLAIM` is invoked whenever the `condor_startd` needs to evict a claim representing fetched work.

The `condor_startd` will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the `condor_startd`.

Command-line arguments passed to the hook None.

Standard input given to the hook A copy of the job ClassAd and the slot ClassAd (separated by the string `-----` and a new line).

Expected standard output from the hook None.

Exit status of the hook Ignored.

Hook: Prepare Job The hook defined by the configuration variable `HOOK_PREPARE_JOB` is invoked by the `condor_starter` before a job is going to be run. This hook provides a chance to execute commands to setup the job environment, for example to transfer input files.

The `condor_starter` waits until this hook returns before attempting to execute the job. If the hook returns a non-zero exit status, the `condor_starter` will assume an error was reached while attempting to set up the job environment and abort the job.

Command-line arguments passed to the hook None.

Standard input given to the hook A copy of the job ClassAd.

Expected standard output from the hook A set of attributes to insert or update into the job ad. For example, changing the `Cmd` attribute to a quoted string changes the executable to be run.

Exit status of the hook 0 for success preparing the job, any non-zero value on failure.

Hook: Update Job Info The hook defined by the configuration variable `HOOK_UPDATE_JOB_INFO` is invoked periodically during the life of the job to update information about the status of the job. When the job is first spawned, the `condor_starter` will invoke this hook after

`STARTER_INITIAL_UPDATE_INTERVAL` seconds (defaults to 8). Thereafter, the *condor_starter* will invoke the hook every `STARTER_UPDATE_INTERVAL` seconds (defaults to 300, in other words, every 5 minutes).

The *condor_starter* will not wait for this hook to return before taking other actions, and ignores all output. The hook is simply advisory, and has no impact on the behavior of the *condor_starter*.

Command-line arguments passed to the hook None.

Standard input given to the hook A copy of the job ClassAd that has been augmented with additional attributes describing the current status and execution behavior of the job.

Expected standard output from the hook None.

Exit status of the hook Ignored.

The additional attributes included inside the job ClassAd are:

JobState The current state of the job. Can be either "Running" or "Suspended".

JobPid The process identifier for the initial job directly spawned by the *condor_starter*.

NumPids The number of processes that the job has currently spawned.

JobStartDate The epoch time when the job was first spawned by the *condor_starter*.

RemoteSysCpu The total number of seconds of system CPU time (the time spent at system calls) the job has used.

RemoteUserCpu The total number of seconds of user CPU time the job has used.

ImageSize The memory image size of the job in Kbytes.

Hook: Job Exit The hook defined by the configuration variable `HOOK_JOB_EXIT` is invoked whenever a job exits, either on its own or when being evicted from an execution slot.

The *condor_starter* will wait for this hook to return before taking any other actions. In the case of jobs that are being managed by a *condor_shadow*, this hook is invoked before the *condor_starter* does its own optional file transfer back to the submission machine, writes to the local user log file, or notifies the *condor_shadow* that the job has exited.

Command-line arguments passed to the hook A string describing how the job exited:

- `exit` The job exited or died with a signal on its own.
- `remove` The job was removed with *condor_rm* or the user job policy expressions (for example, `PeriodicRemove`).
- `hold` The job was held with *condor_hold* or the user job policy expressions (for example, `PeriodicHold`).
- `evict` The job was evicted from the execution slot for any other reason (`PREEMPT` evaluated to `TRUE` in the *condor_startd*, *condor_vacate*, *condor_off*, etc).

Standard input given to the hook A copy of the job ClassAd that has been augmented with additional attributes describing the execution behavior of the job and its final results.

Expected standard output from the hook None.

Exit status of the hook Ignored.

The job ClassAd passed to this hook contains all of the extra attributes described above for `HOOK_UPDATE_JOB_INFO`, and the following additional attributes that are only present once a job exits:

ExitReason A human-readable string describing why the job exited.

ExitBySignal A boolean indicating if the job exited due to being killed by a signal, or if it exited with an exit status.

ExitSignal If `ExitBySignal` is true, the signal number that killed the job.

ExitCode If `ExitBySignal` is false, the integer exit code of the job.

JobDuration The number of seconds that the job ran during this invocation.

Keywords to Define Job Fetch Hooks in the Condor Configuration files

Hooks are defined in the Condor configuration files by prefixing the name of the hook with a keyword. This way, a given machine can have multiple sets of hooks, each set identified by a specific keyword.

Each slot on the machine can define a separate keyword for the set of hooks that should be used (`SLOTN_JOB_HOOK_KEYWORD`). Note that the `N` in `SLOTN` should be replaced with the slot identification number. For example, on slot1, the setting would be called `SLOT1_JOB_HOOK_KEYWORD`. If the slot-specific keyword is not defined, the *condor_startd* will use a global keyword (`STARTD_JOB_HOOK_KEYWORD`).

Once a job is fetched via `HOOK_FETCH_WORK`, the *condor_startd* will insert the keyword used to fetch that job into the job ClassAd as `HookKeyword`. This way, the same keyword will be used to select the hooks invoked by the *condor_starter* during the actual execution of the job. However, the `STARTER_JOB_HOOK_KEYWORD` can be defined to force the *condor_starter* to always use a given keyword for its own hooks, instead of looking the job ClassAd for a `HookKeyword` attribute.

For example, the following configuration defines two sets of hooks, and on a machine with 4 slots, 3 of the slots use the global keyword for running work from a database-driven system, and one of the slots uses a custom keyword to handle work fetched from a web service.

```
# Most slots fetch and run work from the database system.
STARTD_JOB_HOOK_KEYWORD = DATABASE

# Slot4 fetches and runs work from a web service.
SLOT4_JOB_HOOK_KEYWORD = WEB

# The database system needs to both provide work and know the reply
# for each attempted claim.
DATABASE_HOOK_DIR = /usr/local/condor/fetch/database
DATABASE_HOOK_FETCH_WORK = $(DATABASE_HOOK_DIR)/fetch_work.php
```

```

DATABASE_HOOK_REPLY_FETCH = $(DATABASE_HOOK_DIR)/reply_fetch.php

# The web system only needs to fetch work.
WEB_HOOK_DIR = /usr/local/condor/fetch/web
WEB_HOOK_FETCH_WORK = $(WEB_HOOK_DIR)/fetch_work.php

```

The keywords "DATABASE" and "WEB" are completely arbitrary, so each site is encouraged to use different (more specific) names as appropriate for their own needs.

Defining the FetchWorkDelay Expression

There are two events that trigger the *condor_startd* to attempt to fetch new work:

- Whenever the *condor_startd* evaluates its own state.
- Whenever the *condor_starter* exits after completing some fetched work.

Even if a given compute slot is already busy running other work, it is possible that if it fetched new work, the *condor_startd* would prefer this newly fetched work (via the Rank expression) over the work it is currently running. However, the *condor_startd* frequently evaluates its own state, especially when a slot is claimed. Therefore, administrators can define a configuration variable which controls how long the *condor_startd* will wait between attempts to fetch new work. This variable is called `FetchWorkDelay`.

The `FetchWorkDelay` expression must evaluate to an integer, which defines the number of seconds since the last fetch attempt completed before the *condor_startd* will attempt to fetch more work. However, as a ClassAd expression (evaluated in the context of the ClassAd of the slot considering if it should fetch more work, and the ClassAd of the currently running job, if any), the length of the delay can be based on the current state the slot and even the currently running job.

For example, a very common configuration would be to always wait 5 minutes (300 seconds) between attempts to fetch work, unless the slot is Claimed/Idle, in which case the *condor_startd* should fetch immediately:

```
FetchWorkDelay = ifThenElse(State == "Claimed" && Activity == "Idle", 0, 300)
```

If the *condor_startd* wants to fetch work, but the time since the last attempted fetch is shorter than the current value of the delay expression, the *condor_startd* will set a timer to fetch as soon as the delay expires.

If this expression is not defined, the *condor_startd* will default to a five minute (300 second) delay between all attempts to fetch work.

Example Job Hook: Making multiple java run times available

A common problem the Java Universe does not solve is when there are multiple Java virtual machines available to run user jobs. If the user knows which jvm their job needs, the administrator can use the Job hooks feature to enable the user to select one java vm. The main idea is to use the prepare job hook to overwrite the Cmd attribute of the job ad. This attribute is the name of the executable the starter will use to run the job. When the job hook overwrites Cmd, it effectively picks which executable will run. So, it is easy to put the execute machine in charge of selecting the specific java install. To do this, define a job hook on the execute machine by adding the following line to the config file.

```
JAVA5_HOOK_PREPARE_JOB=$(LIBEXEC)/java5_prepare_hook
```

With this line, if a user job asks for the JAVA5 job hook, by setting

```
+HookKeyword="JAVA"
```

in the condor_submit file, the starter will run the program java5_prepare_hook before running this job. Note that the double quote characters are required. Any output from this program will be considered updates to the job ad. So, a simple java5_prepare_hook script might look like this:

```
#!/bin/sh

# Read and discard the job classad
cat > /dev/null
echo 'Cmd = "/usr/java/java5/bin/java"'
```

A complete job submit file in this case might look like the following:

```
universe = vanilla
executable = /usr/bin/java
arguments = Hello

should_transfer_files = always
when_to_transfer_output = on_exit
transfer_input_files = Hello.class

output = o
error = e
log = log

+HookKeyword="JAVA5"
queue
```

4.4.2 Hooks for a Job Router

Job Router Hooks allow for an alternate transformation and/or monitoring engine for the Job Router. Routing is still controlled by the Job Router, but if the Job Router Hooks are configured, then these hooks will be used to transform and monitor the job instead of Condor's internal engine.

Job Router Hooks are similar in concept to Job Fetch Hooks, but they are limited in their scope to just the Job Router. As with Job Fetch Hooks, a hook is an external program or script invoked by Condor's Job Router at various points during the life cycle of a routed job.

The following sections describe how these hooks are used by the Job Router, what hooks are invoked by Condor at various stages of the job's life, and how to configure Condor to use Job Router Hooks.

Hooks Invoked for Job Routing

The Job Router Hooks allow for replacement of the transformation engine used by Condor for routing a job. Since the external transformation engine is not controlled by Condor, additional hooks provide a means to update the job's status in Condor and clean up upon exit or failure cases. This allows a job to be transformed to just about any other type of job that Condor supports, as well as to use execution nodes not normally available to Condor.

It is important to note that if the Job Router Hooks are configured, then Condor will not ignore a failure in any hook execution. If a hook is configured, then Condor assumes it is required to execute and will not continue by falling back to a part of its internal engine. For example, if there is a problem transforming the job using the hooks, Condor will not fall back on its internal transformation engine to process the job.

There are 4 hooks that the Job Router can be configured to use. Each hook will be described below along with data passed to the hook and expected output. Again, all hooks must exit successfully.

Hook: Translate The hook defined by the configuration variable `HOOK_TRANSLATE` is invoked when the Job Router has determined that a job meets the definition for a route. This hook is responsible for doing the transformation of the job and configuring any resources that are external to Condor if applicable.

Command-line arguments passed to the hook None.

Standard input given to the hook The first line will be the route that the job matched as defined in Condor's configuration files followed by the job ClassAd, separated by the string "-----" and a new line.

Expected standard output from the hook The transformed job.

Exit status of the hook 0 for success, any non-zero value on failure.

Hook: Update Job Info The hook defined by the configuration variable `HOOK_UPDATE_JOB_INFO` is invoked to provide status on the specified routed job when the Job Router polls the status of routed jobs at intervals set by `JOB_ROUTER_POLLING_PERIOD`.

Command-line arguments passed to the hook None.

Standard input given to the hook The routed job ClassAd that is to be updated.

Expected standard output from the hook The job attributes to be updated in the routed job, or nothing, if there was no update. To prevent clashing with Condor's management of job attributes, only attributes that are not managed by Condor should be output from this hook.

Exit status of the hook 0 for success, any non-zero value on failure.

Hook: Job Finalize The hook defined by the configuration variable `HOOK_JOB_FINALIZE` is invoked when the Job Router has found that the job has completed. Any output from the hook is treated as an update to the source job.

Command-line arguments passed to the hook None.

Standard input given to the hook The source job ClassAd, followed by the routed copy ClassAd that completed, separated by the string "-----" and a new line.

Expected standard output from the hook An updated source job ClassAd, or nothing if there was no update.

Exit status of the hook 0 for success, any non-zero value on failure.

Hook: Job Cleanup The hook defined by the configuration variable `HOOK_JOB_CLEANUP` is invoked when the Job Router finishes managing the job. This hook will be invoked regardless of whether the job completes successfully or not, and must exit successfully.

Command-line arguments passed to the hook None.

Standard input given to the hook The job ClassAd that the Job Router is done managing.

Expected standard output from the hook None.

Exit status of the hook 0 for success, any non-zero value on failure.

4.5 Application Program Interfaces

4.5.1 Web Service

Condor's Web Service (WS) API provides a way for application developers to interact with Condor, without needing to utilize Condor's command-line tools. In keeping with the Condor philosophy of reliability and fault-tolerance, this API is designed to provide a simple and powerful way to interact with Condor. Condor daemons understand and implement the SOAP (Simple Object Access Protocol) XML API to provide a web service interface for Condor job submission and management.

To deal with the issues of reliability and fault-tolerance, a two-phase commit mechanism provides a transaction-based protocol. The following API description describes interaction between a client using the API and both the *condor_schedd* and *condor_collector* daemons to illustrate transactions for use in job submission, queue management and ClassAd management functions.

Transactions

All applications using the API to interact with the *condor_schedd* will need to use transactions. A transaction is an ACID unit of work (atomic, consistent, isolated, and durable). The API limits the lifetime of a transaction, and both the client (application) and the server (the *condor_schedd* daemon) may place a limit on the lifetime. The server reserves the right to specify a maximum duration for a transaction.

The client initiates a transaction using the `beginTransaction()` method. It ends the transaction with either a commit (using `commitTransaction()`) or an abort (using `abortTransaction()`).

Not all operations in the API need to be performed within a transaction. Some accept a null transaction. A null transaction is a SOAP message with

```
<transaction xsi:type="ns1:Transaction" xsi:nil="true"/>
```

Often this is achieved by passing the programming language's equivalent of null in place of a transaction identifier. It is possible that some operations will have access to more information when they are used inside a transaction. For instance, a `getJobAds()` query would have access to the jobs that are pending in a transaction, which are not committed and therefore not visible outside of the transaction. Transactions are as ACID compliant as possible. Therefore, do not query for information outside of a transaction on which to make a decision inside a transaction based on the query's results.

Job Submission

A ClassAd is required to describe a job. The job ClassAd will be submitted to the *condor_schedd* within a transaction using the `submit()` method. The complexity of job ClassAd creation may be simplified by the `createJobTemplate()` method. It returns an instance of a ClassAd structure that may be further modified. A necessary part of the job ClassAd are the job attributes `ClusterId` and `ProcId`, which uniquely identify the cluster and the job within a cluster. Allocation and assignment of (monotonically increasing) `ClusterId` values utilize the `newCluster()` method. Jobs may be submitted within the assigned cluster only until the `newCluster()` method is invoked a subsequent time. Each job is allocated and assigned a (monotonically increasing) `ProcId` within the current cluster using the `newJob()` method. Therefore, the sequence of method calls to submit a set of jobs initially calls `newCluster()`. This is followed by calls to `newJob()` and then `submit()` for each job within the cluster.

As an example, here are sample cluster and job numbers that result from the ordered calls to submission methods:

1. A call to `newCluster()`, assigns a `ClusterId` of 6.
2. A call to `newJob()`, assigns a `ProcId` of 0, as this is the first job within the cluster.
3. A call to `submit()` results in a job submission numbered 6.0.
4. A call to `newJob()`, assigns a `ProcId` of 1.
5. A call to `submit()` results in a job submission numbered 6.1.
6. A call to `newJob()`, assigns a `ProcId` of 2.
7. A call to `submit()` results in a job submission numbered 6.2.
8. A call to `newCluster()`, assigns a `ClusterId` of 7.
9. A call to `newJob()`, assigns a `ProcId` of 0, as this is the first job within the cluster.
10. A call to `submit()` results in a job submission numbered 7.0.
11. A call to `newJob()`, assigns a `ProcId` of 1.
12. A call to `submit()` results in a job submission numbered 7.1.

There is the potential that a call to `submit()` will fail. Failure means that the job is in the queue, and it typically indicates that something needed by the job has not been sent. As a result the job has no hope in successfully running. It is possible to recover from such a failure by trying to resend information that the job will need. It is also completely acceptable to abort and make another attempt. To simplify the client's effort in figuring out what the job requires, a `discoverJobRequirements()` method accepting a job `ClassAd` and returning a list of things that should be sent along with the job is provided.

File Transfer

A common job submission case requires the job's executable and input files to be transferred from the machine where the application is running to the machine where the *condor_schedd* daemon is running. This is the analogous situation to running *condor_submit* using the **-spool** or **-remote** option. The executable and input files must be sent directly to the *condor_schedd* daemon, which places all files in a spool location.

The two methods `declareFile()` and `sendFile()` work in tandem to transfer files to the *condor_schedd* daemon. The `declareFile()` method causes the *condor_schedd* daemon to create the file in its spool location, or indicate in its return value that the file already exists. This increases efficiency, as resending an existing file is a waste of resources. The `sendFile()` method sends base64 encoded data. `sendFile()` may be used to send an entire file, or chunks of files as desired.

The `declareFile()` method has both required and optional arguments. `declareFile()` requires the name of the file and its size in bytes. The optional arguments relate hash information. A hash type of `NOHASH` disables file verification; the *condor_schedd* daemon will not have a reliable way to determine the existence of the file being declared.

Methods for retrieving files are most useful when a job is completed. Consider the categorization of the typical life-cycle for a job:

Birth: The birth of a job begins with `submit()`.

Childhood: The job executes.

Middle Age: A completed job waits to be removed. As the job enters Middle Age, its `JobStatus` `ClassAd` attribute becomes `Completed` (the value 4).

Old Age: The job's information goes into the history log.

Once the job enters Middle Age, the `getFile()` method retrieves a file. The `listSpool()` method assists by providing a list of all the job's files in the spool location.

The job enters Old Age by the application's use of the `closeSpool()` method. It causes the *condor_schedd* daemon to remove the job from the queue, and the job's spool files are no longer available. As there is no requirement for the application to invoke the `closeSpool()` method, jobs can potentially remain in the queue forever. The configuration variable `SOAP_LEAVE_IN_QUEUE` may mitigate this problem. When this boolean variable evaluates to `False`, a job enters Old Age. A reasonable example for this configuration variable is

```
SOAP_LEAVE_IN_QUEUE = ((JobStatus==4) && ((ServerTime - CompletionDate) < (60 * 60 * 24)))
```

This expression results in Old age for a job (removed from the queue), once the job has been Middle Aged (been completed) for 24 hours.

Implementation Details

Condor daemons understand and communicate using the SOAP XML protocol. An application seeking to use this protocol will require code that handles the communication. The XML WSDL (Web Services Description Language) that Condor implements is included with the Condor distribution. It is in `$(RELEASE_DIR)/lib/webservice`. The WSDL must be run through a toolkit to produce language-specific routines that do communication. The application is compiled with these routines.

Condor must be configured to enable responses to SOAP calls. Please see section 3.3.32 for definitions of the configuration variables related to the web services API. The WS interface is listening on the *condor_schedd* daemon's command port. To obtain a list of all the *condor_schedd* daemons in the pool with a WS interface, issue the command:

```
% condor_status -schedd -constraint "HasSOAPInterface=?=TRUE"
```

With this information, a further command locates the port number to use:

```
% condor_status -schedd -constraint "HasSOAPInterface=?=TRUE" -l | grep MyAddress
```

Condor's security configuration must be set up such that access is authorized for the SOAP client. See Section 3.6.7 for information on how to set the `ALLOW_SOAP` and `DENY_SOAP` configuration variables.

The API's routines can be roughly categorized into ones that deal with

- Transactions
- Job Submission
- File Transfer
- Job Management
- ClassAd Management
- Version Information

The routines for each of these categories is detailed. Note that the signature provided will accurately reflect a routine's name, but that return values and parameter specification will vary according to the target programming language.

Get These Items Correct

- For jobs that are to be executed on Windows platforms, explicitly set the job ClassAd attribute `NTDomain`. This attribute defines the NT domain within which the job's owner authenticates. The attribute is necessary, and it is not set for the job by the `createJobTemplate()` function.

Methods for Transaction Management

beginTransaction Begin a transaction. A prototype is

```
StatusAndTransaction beginTransaction(int duration);
```

Parameters • `duration` The expected duration of the transaction.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the new transaction.

commitTransaction Commits a transaction. A prototype is

```
Status commitTransaction(Transaction transaction);
```

Parameters • `transaction` The transaction to be committed.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

abortTransaction Abort a transaction. A prototype is

```
Status abortTransaction(Transaction transaction);
```

Parameters • `transaction` The transaction to be aborted.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

extendTransaction Request an extension in duration for a specific transaction. A prototype is

```
StatusAndTransaction extendTransaction( Transaction
transaction, int duration);
```

Parameters • `transaction` The transaction to be extended.
• `duration` The duration of the extension.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the transaction with the extended duration.

Methods for Job Submission

submit Submit a job. A prototype is

```
StatusAndRequirements submit(Transaction transaction, int
clusterId, int jobId, ClassAd jobAd);
```

Parameters • `transaction` The transaction in which the submission takes place.
• `clusterId` The cluster identifier.
• `jobId` The job identifier.
• `jobAd` The ClassAd describing the job. Creation of this ClassAd can be simplified with `createJobTemplate()`.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, the return value contains the job's requirements.

createJobTemplate Request a job Class Ad, given some of the job requirements. This job Class Ad will be suitable for use when submitting the job. Note that the job attribute `NTDomain` is not set by this function, but must be set for jobs that will execute on Windows platforms. A prototype is

```
StatusAndClassAd createJobTemplate(int clusterId, int jobId,
String owner, UniverseType type, String command, String
arguments, String requirements);
```

- Parameters**
- `clusterId` The cluster identifier.
 - `jobId` The job identifier.
 - `owner` The name to be associated with the job.
 - `type` The universe under which the job will run, where `type` can be one of the following:

```
enum UniverseType { STANDARD = 1, VANILLA = 5,
  SCHEDULER = 7, MPI = 8, GRID = 9, JAVA = 10, PARALLEL = 11, LOCALUNIVERSE = 12, VM = 13 };
```
 - `command` The command to execute once the job has started.
 - `arguments` The command-line arguments for `command`.
 - `requirements` The requirements expression for the job. For further details and examples of the expression syntax, please refer to section 4.1.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

discoverJobRequirements Discover the requirements of a job, given a Class Ad. May be helpful in determining what should be sent along with the job. A prototype is

```
StatusAndRequirements discoverJobRequirements( ClassAd jobAd );
```

- Parameters**
- `jobAd` The ClassAd of the job.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the job's requirements.

Methods for File Transfer

declareFile Declare a file that may be used by a job. A prototype is

```
Status declareFile( Transaction transaction, int clusterId, int
  jobId, String name, int size, HashType hashType, String hash );
```

- Parameters**
- `transaction` The transaction in which this file is declared.
 - `clusterId` The cluster identifier.
 - `jobId` An identifier of the job that will use the file.
 - `name` The name of the file.
 - `size` The size of the file.
 - `hashType` The type of hash mechanism used to verify file integrity, where `hashType` can be one of the following:

```
enum HashType { NOHASH, MD5HASH };
```
 - `hash` An optionally zero-length string encoding of the file hash.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

sendFile Send a file that a job may use. A prototype is

```
Status sendFile(Transaction transaction, int clusterId, int
jobId, String name, int offset, Base64 data);
```

- Parameters**
- **transaction** The transaction in which this file is send.
 - **clusterId** The cluster identifier.
 - **jobId** An identifier of the job that will use the file.
 - **name** The name of the file being sent.
 - **offset** The starting offset within the file being sent.
 - **length** The length from the offset to send.
 - **data** The data block being sent. This could be the entire file or a sub-section of the file as defined by **offset** and **length**.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see **StatusCode** for valid return values.

getFile Get a file from a job's spool. A prototype is

```
StatusAndBase64 getFile(Transaction transaction, int
clusterId, int jobId, String name, int offset, int length);
```

- Parameters**
- **transaction** An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - **clusterId** The cluster in which to search.
 - **jobId** The job identifier the file is associated with.
 - **name** The name of the file to retrieve.
 - **offset** The starting offset withing the file being retrieved.
 - **length** The length from the offset to retrieve.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see **StatusCode** for valid return values. Additionally, on success, the return value contains the file or a sub-section of the file as defined by **offset** and **length**.

closeSpool Close a job's spool. All the files in the job's spool can be deleted. A prototype is

```
Status closeSpool(Transaction transaction, int clusterId, int
jobId);
```

- Parameters**
- **transaction** An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - **clusterId** The cluster identifier which the job is associated with.
 - **jobId** The job identifier for which the spool is to be removed.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see **StatusCode** for valid return values.

listSpool List the files in a job's spool. A prototype is

```
StatusAndFileInfoArray listSpool(Transaction transaction, int
clusterId, int jobId);
```

- Parameters**
- **transaction** An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - **clusterId** The cluster in which to search.
 - **jobId** The job identifier to search for.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see StatusCode for valid return values. Additionally, on success, the return value contains a list of files and their respective sizes.

Methods for Job Management

newCluster Create a new job cluster. A prototype is

```
StatusAndInt newCluster(Transaction transaction);
```

- Parameters**
- **transaction** The transaction in which this cluster is created.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see StatusCode for valid return values. Additionally, on success, the return value contains the cluster id.

removeCluster Remove a job cluster, and all the jobs within it. A prototype is

```
Status removeCluster(Transaction transaction, int clusterId,  
String reason);
```

- Parameters**
- **transaction** An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - **clusterId** The cluster to remove.
 - **reason** The reason for the removal.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see StatusCode for valid return values.

newJob Creates a new job within the most recently created job cluster. A prototype is

```
StatusAndInt newJob(Transaction transaction, int clusterId);
```

- Parameters**
- **transaction** The transaction in which this job is created.
 - **clusterId** The cluster identifier of the most recently created cluster.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see StatusCode for valid return values. Additionally, on success, the return value contains the job id.

removeJob Remove a job, regardless of the job's state. A prototype is

```
Status removeJob(Transaction transaction, int clusterId, int  
jobId, String reason, boolean forceRemoval);
```

- Parameters**
- **transaction** An optionally nullable transaction, meaning this call does not need to occur in a transaction.

- `clusterId` The cluster identifier to search in.
- `jobId` The job identifier to search for.
- `reason` The reason for the release.
- `forceRemoval` Set if the job should be forcibly removed.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

holdJob Put a job into the Hold state, regardless of the job's current state. A prototype is

```
Status holdJob(Transaction transaction, int clusterId, int
jobId, string reason, boolean emailUser, boolean emailAdmin,
boolean systemHold);
```

- Parameters**
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - `clusterId` The cluster in which to search.
 - `jobId` The job identifier to search for.
 - `reason` The reason for the release.
 - `emailUser` Set if the submitting user should be notified.
 - `emailAdmin` Set if the administrator should be notified.
 - `systemHold` Set if the job should be put on hold.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

releaseJob Release a job that has been in the Hold state. A prototype is

```
Status releaseJob(Transaction transaction, int clusterId, int
jobId, String reason, boolean emailUser, boolean emailAdmin);
```

- Parameters**
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - `clusterId` The cluster in which to search.
 - `jobId` The job identifier to search for.
 - `reason` The reason for the release.
 - `emailUser` Set if the submitting user should be notified.
 - `emailAdmin` Set if the administrator should be notified.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

getJobAds A prototype is

```
StatusAndClassAdArray getJobAds(Transaction transaction,
String constraint);
```

- Parameters**
- `transaction` An optionally nullable transaction, meaning this call does not need to occur in a transaction.

- **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains all job ClassAds matching the given constraint.

getJobAd Finds a specific job ClassAd.

This method does much the same as the first element from the array returned by

```
getJobAds(transaction, "(ClusterId==clusterId && JobId==jobId)")
```

A prototype is

```
StatusAndClassAd getJobAd(Transaction transaction, int
clusterId, int jobId);
```

- Parameters**
- **transaction** An optionally nullable transaction, meaning this call does not need to occur in a transaction.
 - **clusterId** The cluster in which to search.
 - **jobId** The job identifier to search for.

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values. Additionally, on success, the return value contains the requested ClassAd.

requestReschedule Request a *condor_reschedule* from the *condor_schedd* daemon. A prototype is

```
Status requestReschedule();
```

Return Value If the function succeeds, the return value is `SUCCESS`; otherwise, see `StatusCode` for valid return values.

Methods for ClassAd Management

insertAd A prototype is

```
Status insertAd(ClassAdType type, ClassAdStruct ad);
```

- Parameters**
- **type** The type of ClassAd to insert, where type can be one of the following:

```
enum ClassAdType { STARTD_AD_TYPE, QUILL_AD_TYPE,
SCHEDD_AD_TYPE, SUBMITTOR_AD_TYPE, LICENSE_AD_TYPE,
MASTER_AD_TYPE, CKPTSRVR_AD_TYPE, COLLECTOR_AD_TYPE,
STORAGE_AD_TYPE, NEGOTIATOR_AD_TYPE, HAD_AD_TYPE,
GENERIC_AD_TYPE };
```
 - **ad** The ClassAd to insert.

Return Value If the function succeeds, the return value is SUCCESS; otherwise, see StatusCode for valid return values.

queryStartdAds A prototype is

```
ClassAdArray queryStartdAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the *condor_startd* ClassAds matching the given constraint.

queryScheddAds A prototype is

```
ClassAdArray queryScheddAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the *condor_schedd* ClassAds matching the given constraint.

queryMasterAds A prototype is

```
ClassAdArray queryMasterAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the *condor_master* ClassAds matching the given constraint.

querySubmittorAds A prototype is

```
ClassAdArray querySubmittorAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the submitters ClassAds matching the given constraint.

queryLicenseAds A prototype is

```
ClassAdArray queryLicenseAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the license ClassAds matching the given constraint.

queryStorageAds A prototype is

```
ClassAdArray queryStorageAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the storage ClassAds matching the given constraint.

queryAnyAds A prototype is

```
ClassAdArray queryAnyAds(String constraint);
```

Parameters • **constraint** A string constraining the number ClassAds to return. For further details and examples of the constraint syntax, please refer to section 4.1.

Return Value A list of all the ClassAds matching the given constraint. to return.

Methods for Version Information

getVersionString A prototype is

```
StatusAndString getVersionString();
```

Return Value Returns the Condor version as a string.

getPlatformString A prototype is

```
StatusAndString getPlatformString();
```

Return Value Returns the platform information Condor is running on as string.

Common Data Structures

Many methods return a status. Table 4.1 lists and defines the StatusCode return values.

Value	Identifier	Definition
0	SUCCESS	All OK
1	FAIL	An error occurred that is not specific to another error code
2	INVALIDTRANSACTION	No such transaction exists
3	UNKNOWNCLUSTER	The specified cluster is not the currently active one
4	UNKNOWNJOB	The specified job does not exist within the specified cluster
5	UNKNOWNFILE	
6	INCOMPLETE	
7	INVALIDOFFSET	
8	ALREADYEXISTS	For this job, the specified file already exists

Table 4.1: StatusCode definitions

4.5.2 The DRMAA API

The following quote from the DRMAA Specification 1.0 abstract nicely describes the purpose of the API:

The Distributed Resource Management Application API (DRMAA), developed by a working group of the Global Grid Forum (GGF),

provides a generalized API to distributed resource management systems (DRMSs) in order to facilitate integration of application programs. The scope of DRMAA is limited to job submission, job monitoring and control, and the retrieval of the finished job status. DRMAA provides application developers and distributed resource management builders with a programming model that enables the development of distributed applications tightly coupled to an underlying DRMS. For deployers of such distributed applications, DRMAA preserves flexibility and choice in system design.

The API allows users who write programs using DRMAA functions and link to a DRMAA library to submit, control, and retrieve information about jobs to a Grid system. The Condor implementation of a portion of the API allows programs (applications) to use the library functions provided to submit, monitor and control Condor jobs.

See the DRMAA site (<http://www.drmaa.org>) to find the API specification for DRMA 1.0 for further details on the API.

Implementation Details

The library was developed from the DRMA API Specification 1.0 of January 2004 and the DRMAA C Bindings v0.9 of September 2003. It is a static C library that expects a POSIX thread model on Unix systems and a Windows thread model on Windows systems. Unix systems that do not support POSIX threads are not guaranteed thread safety when calling the library's functions.

The object library file is called `libcondordrmaa.a`, and it is located within the `<release>/lib` directory in the Condor download. Its header file is called `lib_condor_drmaa.h`, and it is located within the `<release>/include` directory in the Condor download. Also within `<release>/include` is the file `lib_condor_drmaa.README`, which gives further details on the implementation.

Use of the library requires that a local *condor_schedd* daemon must be running, and the program linked to the library must have sufficient spool space. This space should be in `/tmp` or specified by the environment variables `TEMP`, `TMP`, or `SPOOL`. The program linked to the library and the local *condor_schedd* daemon must have read, write, and traverse rights to the spool space.

The library currently supports the following specification-defined job attributes:

`DRMAA_REMOTE_COMMAND`

`DRMAA_JS_STATE`

`DRMAA_NATIVE_SPECIFICATION`

`DRMAA_BLOCK_EMAIL`

DRMAA_INPUT_PATH
DRMAA_OUTPUT_PATH
DRMAA_ERROR_PATH
DRMAA_V_ARGV
DRMAA_V_ENV
DRMAA_V_EMAIL

The attribute `DRMAA_NATIVE_SPECIFICATION` can be used to direct all commands supported within submit description files. See the *condor_submit* manual page at section 9 for a complete list. Multiple commands can be specified if separated by newlines.

As in the normal submit file, arbitrary attributes can be added to the job's ClassAd by prefixing the attribute with `+`. In this case, you will need to put string values in quotation marks, the same as in a submit file.

Thus to tell Condor that the job will likely use 64 megabytes of memory (65536 kilobytes), to more highly rank machines with more memory, and to add the arbitrary attribute of department set to chemistry, you would set `AttrDRMAA_NATIVE_SPECIFICATION` to the C string:

```
drmaa_set_attribute(jobtemplate, DRMAA_NATIVE_SPECIFICATION,  
    "image_size=65536\nrank=Memory\n+department=\"chemistry\"",  
    err_buf, sizeof(err_buf)-1);
```

4.5.3 The Command Line Interface

This section has not yet been written

4.5.4 The Condor GAHP

This section has not yet been written

4.5.5 The Condor Perl Module

The Condor Perl module facilitates automatic submitting and monitoring of Condor jobs, along with automated administration of Condor. The most common use of this module is the monitoring of Condor jobs. The Condor Perl module can be used as a meta scheduler for the submission of Condor jobs.

The Condor Perl module provides several subroutines. Some of the subroutines are used as callbacks; an event triggers the execution of a specific subroutine. Other of the subroutines denote actions to be taken by Perl. Some of these subroutines take other subroutines as arguments.

Subroutines

Submit(submitdescriptionfile) This subroutine takes the action of submitting a job to Condor. The argument is the name of a submit description file. The *condor_submit* program should be in the path of the user. If the user wishes to monitor the job with condor they must specify a log file in the command file. The cluster submitted is returned. For more information see the *condor_submit* man page.

Vacate(machine) This subroutine takes the action of sending a *condor_vacate* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_vacate* man page.

Reschedule(machine) This subroutine takes the action of sending a *condor_reschedule* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_reschedule* man page.

Monitor(cluster) Takes the action of monitoring this cluster. It returns when all jobs in cluster terminate.

Wait() Takes the action of waiting until all monitor subroutines finish, and then exits the Perl script.

DebugOn() Takes the action of turning debug messages on. This may be useful when attempting to debug the Perl script.

DebugOff() Takes the action of turning debug messages off.

RegisterEvicted(sub) Register a subroutine (called *sub*) to be used as a callback when a job from a specified cluster is evicted. The subroutine will be called with two arguments: cluster and job. The cluster and job are the cluster number and process number of the job that was evicted.

RegisterEvictedWithCheckpoint(sub) Same as RegisterEvicted except that the handler is called when the evicted job was checkpointed.

RegisterEvictedWithoutCheckpoint(sub) Same as RegisterEvicted except that the handler is called when the evicted job was not checkpointed.

RegisterExit(sub) Register a termination handler that is called when a job exits. The termination handler will be called with two arguments: cluster and job. The cluster and job are the cluster and process numbers of the existing job.

RegisterExitSuccess(sub) Register a termination handler that is called when a job exits without errors. The termination handler will be called with two arguments: cluster and job. The cluster and job are the cluster and process numbers of the existing job.

RegisterExitFailure(sub) Register a termination handler that is called when a job exits with errors. The termination handler will be called with three arguments: cluster, job and retval. The cluster and job are the cluster and process numbers of the existing job and the retval is the exit code of the job.

RegisterExitAbnormal(sub) Register an termination handler that is called when a job abnormally exits (segmentation fault, bus error, ...). The termination handler will be called with four arguments: cluster, job signal and core. The cluster and job are the cluster and process numbers of the existing job. The signal indicates the signal that the job died with and core indicates whether a core file was created and if so, what the full path to the core file is.

RegisterAbort(sub) Register a handler that is called when a job is aborted by a user.

RegisterJobErr(sub) Register a handler that is called when a job is not executable.

RegisterExecute(sub) Register an execution handler that is called whenever a job starts running on a given host. The handler is called with four arguments: cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_starter* supervising the job.

RegisterSubmit(sub) Register a submit handler that is called whenever a job is submitted with the given cluster. The handler is called with cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_schedd* responsible for the job.

Monitor(cluster) Begin monitoring this cluster. Returns when all jobs in cluster terminate.

Wait() Wait until all monitors finish and exit.

DebugOn() Turn debug messages on. This may be useful if you don't understand what your script is doing.

DebugOff() Turn debug messages off.

TestSubmit(commandfile) This subroutine submits a job to Condor for testing, and places all variables from the command file into the Perl hash %submit_info. Does not reset the state of variables, so that testing preserves callbacks.

SubmitDagman(DAGfile, DAGManargs) Takes the action of submitting a DAG using *condor_dagman*. The first argument is the name of the DAG input file, and the second argument is the command line arguments for *condor_dagman*. Information from the submit description file generated by *condor_dagman* is placed into the Perl hash %submit_info for access during callbacks.

TestSubmitDagman(DAGfile, DAGManargs) This subroutine submits a *condor_dagman* to Condor for testing, and places information from the submit description file generated by *condor_dagman* into the Perl hash %submit_info for access during

callbacks. The first argument is the name of the DAG input file, and the second argument is the command line arguments for *condor_dagman*. Does not reset the state of variables, so that testing preserves callbacks.

RegisterEvictedWithRequeue(sub) Register a subroutine (called sub) to be used as a callback when a job from a specified cluster is requeued. The subroutine will be called with two arguments: cluster and job. The cluster and job are the cluster number and process number of the job that was requeued.

RegisterShadow(sub) Register a subroutine (called sub) to be used as a callback when a shadow exception occurs.

RegisterHold(sub) Register a subroutine (called sub) to be used as a callback when a job enters the hold state.

RegisterRelease(sub) Register a subroutine (called sub) to be used as a callback when a job is released.

RegisterWantError(sub) Register a subroutine (called sub) to be used as a callback when a system call invoked using runCommand experiences an error.

runCommand(string) string identifies a syscall that is invoked. If the syscall exits abnormally or exits with an error, the callback registered with RegisterWantError() is called, and an error message is issued.

RegisterTimed(sub, seconds) Register a subroutine (called sub) to be called back at a delay of seconds time from this registration time. Only one callback may be registered, as subsequent calls modify the timer only.

RemoveTimed() Remove the single, timed callback registered with RegisterTimed().

Examples

The following is an example that uses the Condor Perl module. The example uses the submit description file mycmdfile.cmd to specify the submission of a job. As the job is matched with a machine and begins to execute, a callback subroutine (called execute) sends a *condor_vacate* signal to the job, and it increments a counter which keeps track of the number of times this callback executes. A second callback keeps a count of the number of times that the job was evicted before the job completes. After the job completes, the termination callback (called normal) prints out a summary of what happened.

```
#!/usr/bin/perl
use Condor;

$CMD_FILE = 'mycmdfile.cmd';
$evicts = 0;
$vacates = 0;

# A subroutine that will be used as the normal execution callback
```

```

$normal = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'} ;
    $job = $parameters{'job'} ;

    print "Job $cluster.$job exited normally without errors.\n" ;
    print "Job was vacated $vacates times and evicted $evicts times\n" ;
    exit(0) ;
};

$evicted = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'} ;
    $job = $parameters{'job'} ;

    print "Job $cluster, $job was evicted.\n" ;
    $evicts++ ;
    &Condor::Reschedule() ;
};

$execute = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'} ;
    $job = $parameters{'job'} ;
    $host = $parameters{'host'} ;
    $sinful = $parameters{'sinful'} ;

    print "Job running on $sinful, vacating...\n" ;
    &Condor::Vacate($sinful) ;
    $vacates++ ;
};

$cluster = Condor::Submit($CMD_FILE) ;
printf("Could not open. Access Denied\n") ;
break ;
&Condor::RegisterExitSuccess($normal) ;
&Condor::RegisterEvicted($evicted) ;
&Condor::RegisterExecute($execute) ;
&Condor::Monitor($cluster) ;
&Condor::Wait() ;

```

This example program will submit the command file 'mycmdfile.cmd' and attempt to vacate any machine that the job runs on. The termination handler then prints out a summary of what has happened.

A second example Perl script facilitates the meta-scheduling of two of Condor jobs. It submits a second job if the first job successfully completes.

```

#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib' ;
# tell Perl to use what it finds in the Condor library

```

```
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    $cluster = Condor::Submit($SUBMIT_FILE2);
    if (($cluster) == 0)
    {
        printf("Could not open $SUBMIT_FILE2.\n");
    }

    &Condor::RegisterExitSuccess($secondOK);
    &Condor::RegisterExitFailure($secondfails);
    &Condor::Monitor($cluster);
};

$firstfails = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exiting with an error. \n";
    exit(0);
};

# Callback used when second job exits without errors.
$secondOK = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully completed. \n";
    exit(0);
};

# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};

$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
```

```

{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);

&Condor::Monitor($cluster);
&Condor::Wait();

```

Some notes are in order about this example. The same task could be accomplished using the Condor DAGMan metascheduler. The first job is the parent, and the second job is the child. The input file to DAGMan is significantly simpler than this Perl script.

A third example using the Condor Perl module expands upon the second example. Whereas the second example could have been more easily implemented using DAGMan, this third example shows the versatility of using Perl as a metascheduler.

In this example, the result generated from the successful completion of the first job are used to decide which subsequent job should be submitted. This is a very simple example of a branch and bound technique, to focus the search for a problem solution.

```

#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';
$SUBMIT_FILE3 = 'Csubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    # open output file from first job, and read the result
    if ( -f "A.output" )
    {
        open(RESULTFILE, "A.output") or die "Could not open result file.";
        $result = <RESULTFILE>;
        close(RESULTFILE);
        # next job to submit is based on output from first job
        if ($result < 100)
        {
            $cluster = Condor::Submit($SUBMIT_FILE2);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE2.\n");
            }
        }
    }
}

```

```

        &Condor::RegisterExitSuccess($secondOK);
        &Condor::RegisterExitFailure($secondfails);
        &Condor::Monitor($cluster);
    }
    else
    {
        $cluster = Condor::Submit($SUBMIT_FILE3);
        if (($cluster) == 0)
        {
            printf("Could not open $SUBMIT_FILE3.\n");
        }

        &Condor::RegisterExitSuccess($thirdOK);
        &Condor::RegisterExitFailure($thirdfails);
        &Condor::Monitor($cluster);
    }
}
else
{
    printf("Results file does not exist.\n");
}
};

$firstfails = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exiting with an error. \n";
    exit(0);
};

# Callback used when second job exits without errors.
$secondOK = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully completed. \n";
    exit(0);
};

# Callback used when third job exits without errors.
$thirdOK = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job, $cluster.$job successfully completed. \n";
    exit(0);
};

```

```
# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'} ;
    $job = $parameters{'job'} ;

    print "The second job ($cluster.$job) failed. \n" ;
    exit(0) ;
};

# Callback used when third job exits WITH an error.
$thirdfails = sub
{
    %parameters = @_ ;
    $cluster = $parameters{'cluster'} ;
    $job = $parameters{'job'} ;

    print "The third job ($cluster.$job) failed. \n" ;
    exit(0) ;
};

$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);

&Condor::Monitor($cluster);
&Condor::Wait();
```

Grid Computing

5.1 Introduction

A goal of grid computing is to allow the utilization of resources that span many administrative domains. A Condor pool often includes resources owned and controlled by many different people. Yet collaborating researchers from different organizations may not find it feasible to combine all of their computers into a single, large Condor pool. Condor shines in grid computing, continuing to evolve with the field.

Due to the field's rapid evolution, Condor has its own native mechanisms for grid computing as well as developing interactions with other grid systems.

Flocking is a native mechanism that allows Condor jobs submitted from within one pool to execute on another, separate Condor pool. Flocking is enabled by configuration within each of the pools. An advantage to flocking is that jobs migrate from one pool to another based on the availability of machines to execute jobs. When the local Condor pool is not able to run the job (due to a lack of currently available machines), the job flocks to another pool. A second advantage to using flocking is that the user (who submits the job) does not need to be concerned with any aspects of the job. The user's submit description file (and the job's **universe**) are independent of the flocking mechanism.

Other forms of grid computing are enabled by using the **grid universe** and further specified with the **grid_type**. For any Condor job, the job is submitted on a machine in the local Condor pool. The location where it is executed is identified as the remote machine or remote resource. These various grid computing mechanisms offered by Condor are distinguished by the software running on the remote resource.

When Condor is running on the remote resource, and the desired grid computing mechanism is

to move the job from the local pool's job queue to the remote pool's job queue, it is called Condor-C. The job is submitted using the **grid universe**, and the **grid_type** is **condor**. Condor-C jobs have the advantage that once the job has moved to the remote pool's job queue, a network partition does not affect the execution of the job. A further advantage of Condor-C jobs is that the **universe** of the job at the remote resource is not restricted.

When other middleware is running on the remote resource, such as Globus, Condor can still submit and manage jobs to be executed on remote resources. A **grid universe** job, with a **grid_type** of **gt2** or **gt4** calls on Globus software to execute the job on a remote resource. Like Condor-C jobs, a network partition does not affect the execution of the job. The remote resource must have Globus software running.

Condor also facilitates the temporary addition of a Globus-controlled resource to a local pool. This is called *glidein*. Globus software is utilized to execute Condor daemons on the remote resource. The remote resource appears to have joined the local Condor pool. A user submitting a job may then explicitly specify the remote resource as the execution site of a job.

Starting with Condor Version 6.7.0, the **grid** universe replaces the **globus** universe. Further specification of a **grid** universe job is done within the **grid_resource** command in a submit description file.

5.2 Connecting Condor Pools with Flocking

Flocking is Condor's way of allowing jobs that cannot immediately run (within the pool of machines where the job was submitted) to instead run on a different Condor pool. If a machine within Condor pool A can send jobs to be run on Condor pool B, then we say that jobs from machine A flock to pool B. Flocking can occur in a one way manner, such as jobs from machine A flocking to pool B, or it can be set up to flock in both directions. Configuration variables allow the *condor_schedd* daemon (which runs on each machine that may submit jobs) to implement flocking.

NOTE: Flocking to pools which use Condor's high availability mechanisms is not advised in current versions of Condor. See section 3.10.2 "High Availability of the Central Manager" of the Condor manual for a discussion of these problems.

5.2.1 Flocking Configuration

The simplest flocking configuration sets a few configuration variables. If jobs from machine A are to flock to pool B, then in machine A's configuration, set the following configuration variables:

FLOCK_TO is a comma separated list of the central manager machines of the pools that jobs from machine A may flock to.

FLOCK_COLLECTOR_HOSTS is the list of *condor_collector* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as **FLOCK_TO**, and it would be

defined with

```
FLOCK_COLLECTOR_HOSTS = $(FLOCK_TO)
```

FLOCK_NEGOTIATOR_HOSTS is the list of *condor_negotiator* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be defined with

```
FLOCK_NEGOTIATOR_HOSTS = $(FLOCK_TO)
```

HOSTALLOWNEGOTIATOR_SCHEDD provides a host-based access level and authorization list for the *condor_schedd* daemon to allow negotiation (for security reasons) with the machines within the pools that jobs from machine A may flock to. This configuration variable will not likely need to change from its default value as given in the sample configuration:

```
## Now, with flocking we need to let the SCHEDD trust the other
## negotiators we are flocking with as well. You should normally
## not have to change this either.
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine. See section 3.6.7 on page 302 for a discussion of security macros and their use.

The configuration macros that must be set in pool B are ones that authorize jobs from machine A to flock to pool B.

The host-based configuration macros are more easily set by introducing a list of machines where the jobs may flock from. FLOCK_FROM is a comma separated list of machines, and it is used in the default configuration setting of the security macros that do host-based authorization:

```
HOSTALLOW_WRITE_COLLECTOR = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_WRITE_STARTD     = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_READ_COLLECTOR   = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_READ_STARTD      = $(HOSTALLOW_READ), $(FLOCK_FROM)
```

Wild cards may be used when setting the FLOCK_FROM configuration variable. For example, *.cs.wisc.edu specifies all hosts from the cs.wisc.edu domain.

If the user-based configuration macros for security are used, then the default will be:

```
ALLOW_NEGOTIATOR = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
```

Further, if using Kerberos or GSI authentication, then the setting becomes:

```
ALLOW_NEGOTIATOR = condor@$(UID_DOMAIN)/$(COLLECTOR_HOST)
```

To enable flocking in both directions, consider each direction separately, following the guidelines given.

5.2.2 Job Considerations

A particular job will only flock to another pool when it cannot currently run in the current pool.

At one point, all jobs that utilized flocking were standard universe jobs. This is no longer the case. The submission of jobs under other universes must consider the location of input, output and error files. The common case will be that machines within separate pools do not have a shared file system. Therefore, when submitting jobs, the user will need to consider file transfer mechanisms. These mechanisms are discussed in section 2.5.4 on page 26.

5.3 The Grid Universe

5.3.1 Condor-C, The condor Grid Type

Condor-C allows jobs in one machine's job queue to be moved to another machine's job queue. These machines may be far removed from each other, providing powerful grid computation mechanisms, while requiring only Condor software and its configuration.

Condor-C is highly resistant to network disconnections and machine failures on both the submission and remote sides. An expected usage sets up Personal Condor on a laptop, submits some jobs that are sent to a Condor pool, waits until the jobs are staged on the pool, then turns off the laptop. When the laptop reconnects at a later time, any results can be pulled back.

Condor-C scales gracefully when compared with Condor's flocking mechanism. The machine upon which jobs are submitted maintains a single process and network connection to a remote machine, without regard to the number of jobs queued or running.

Condor-C Configuration

There are two aspects to configuration to enable the submission and execution of Condor-C jobs. These two aspects correspond to the endpoints of the communication: there is the machine from which jobs are submitted, and there is the remote machine upon which the jobs are placed in the queue (executed).

Configuration of a machine from which jobs are submitted requires a few extra configuration variables:

```
CONDOR_GAHP=$(SBIN)/condor_c-gahp
C_GAHP_LOG=/tmp/CGAHPLog.$(USERNAME)
C_GAHP_WORKER_THREAD_LOG=/tmp/CGAHPWorkerLog.$(USERNAME)
```

The acronym GAHP stands for Grid ASCII Helper Protocol. A GAHP server provides grid-related services for a variety of underlying middle-ware systems. The configuration variable

CONDOR_GAHP gives a full path to the GAHP server utilized by Condor-C. The configuration variable C_GAHP_LOG defines the location of the log that the Condor GAHP server writes. The log for the Condor GAHP is written as the user on whose behalf it is running; thus the C_GAHP_LOG configuration variable must point to a location the end user can write to.

A submit machine must also have a *condor_collector* daemon to which the *condor_schedd* daemon can submit a query. The query is for the location (IP address and port) of the intended remote machine's *condor_schedd* daemon. This facilitates communication between the two machines. This *condor_collector* does not need to be the same collector that the local *condor_schedd* daemon reports to.

The machine upon which jobs are executed must also be configured correctly. This machine must be running a *condor_schedd* daemon. Unless specified explicitly in a submit file, CONDOR_HOST must point to a *condor_collector* daemon that it can write to, and the machine upon which jobs are submitted can read from. This facilitates communication between the two machines.

An important aspect of configuration is the security configuration relating to authentication. Condor-C on the remote machine relies on an authentication protocol to know the identity of the user under which to run a job. The following is a working example of the security configuration for authentication. This authentication method, CLAIMTOBE, trusts the identity claimed by a host or IP address.

```
SEC_DEFAULT_NEGOTIATION = OPTIONAL
SEC_DEFAULT_AUTHENTICATION_METHODS = CLAIMTOBE
```

Condor-C Job Submission

Job submission of Condor-C jobs is the same as for any Condor job. The **universe** is **grid**. **grid_resource** specifies the remote *condor_schedd* daemon to which the job should be submitted, and its value consists of three fields. The first field is the grid type, which is **condor**. The second field is the name of the remote *condor_schedd* daemon. Its value is the same as the *condor_schedd* ClassAd attribute Name on the remote machine. The third field is the name of the remote pool's *condor_collector*.

The following represents a minimal submit description file for a job.

```
# minimal submit description file for a Condor-C job
universe = grid
executable = myjob
output = myoutput
error = myerror
log = mylog

grid_resource = condor joe@remotemachine.example.com remotecentralmanager.example.com
+remote_jobuniverse = 5
+remote_requirements = True
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
queue
```

The remote machine needs to understand the attributes of the job. These are specified in the submit description file using the '+' syntax, followed by the string **remote_**. At a minimum, this will be the job's **universe** and the job's **requirements**. It is likely that other attributes specific to the job's **universe** (on the remote pool) will also be necessary. Note that attributes set with '+' are inserted directly into the job's ClassAd. Specify attributes as they must appear in the job's ClassAd, not the submit description file. For example, the **universe** is specified using an integer assigned for a job ClassAd JobUniverse. Similarly, place quotation marks around string expressions. As an example, a submit description file would ordinarily contain

```
when_to_transfer_output = ON_EXIT
```

This must appear in the Condor-C job submit description file as

```
+remote_WhenToTransferOutput = "ON_EXIT"
```

For convenience, the specific entries of **universe**, **remote_grid_resource**, **globus_rsl**, and **globus_xml** may be specified as **remote_** commands without the leading '+'. Instead of

```
+remote_universe = 5
```

the submit description file command may appear as

```
remote_universe = vanilla
```

Similarly, the command

```
+remote_gridresource = "condor schedd.example.com cm.example.com"
```

may be given as

```
remote_grid_resource = condor schedd.example.com cm.example.com
```

For the given example, the job is to be run as a **vanilla universe** job at the remote pool. The (remote pool's) *condor_schedd* daemon is likely to place its job queue data on a local disk and execute the job on another machine within the pool of machines. This implies that the file systems for the resulting submit machine (the machine specified by **remote_schedd**) and the execute machine (the machine that runs the job) will *not* be shared. Thus, the two inserted ClassAds

```
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
```

are used to invoke Condor's file transfer mechanism.

As Condor-C is a recent addition to Condor, the universes, associated integer assignments, and notes about the existence of functionality are given in Table 5.1. The note "untested" implies that submissions under the given universe have not yet been thoroughly tested. They may already work.

Universe Name	Value	Notes
standard	1	untested
vanilla	5	works well
scheduler	7	works well
MPI	8	untested
grid	9	works well
	grid_resource is condor	works well
	grid_resource is gt2	works well
	grid_resource is gt4	untested
	grid_resource is nordugrid	untested
	grid_resource is uncore	untested
	grid_resource is lsf	works well
	grid_resource is pbs	works well
java	10	untested
parallel	11	untested
local	12	works well

Table 5.1: Functionality of remote job universes with Condor-C

For communication between *condor_schedd* daemons on the submit and remote machines, the location of the remote *condor_schedd* daemon is needed. This information resides in the *condor_collector* of the remote machine's pool. The third field of the **grid_resource** command in the submit description file says which *condor_collector* should be queried for the remote *condor_schedd* daemon's location. An example of this submit command is

```
grid_resource = condor_schedd.example.com machine1.example.com
```

If the remote *condor_collector* is not listening on the standard port (9618), then the port it *is* listening on needs to be specified:

```
grid_resource = condor_schedd.example.comd machine1.example.com:12345
```

File transfer of a job's executable, `stdin`, `stdout`, and `stderr` are automatic. When other files need to be transferred using Condor's file transfer mechanism (see section 2.5.4 on page 26), the mechanism is applied based on the resulting job universe on the remote machine.

Condor-C Jobs Between Differing Platforms

Condor-C jobs given to a remote machine running Windows must specify the Windows domain of the remote machine. This is accomplished by defining a ClassAd attribute for the job. Where the Windows domain is different at the submit machine from the remote machine, the submit description file defines the Windows domain of the remote machine with

```
+remote_NTDomain = "DomainAtRemoteMachine"
```

A Windows machine not part of a domain defines the Windows domain as the machine name.

Current Limitations in Condor-C

Submitting jobs to run under the grid universe has not yet been perfected. The following is a list of known limitations with Condor-C:

1. Authentication methods other than CLAIMTOBE, such as GSI and KERBEROS, are untested, and may not yet work.

5.3.2 Condor-G, the gt2 and gt4 Grid Types

Condor-G is the name given to Condor when **grid universe** jobs are sent to grid resources utilizing Globus software for job execution. The Globus Toolkit provides a framework for building grid systems and applications. See the Globus Alliance web page at <http://www.globus.org> for descriptions and details of the Globus software.

Condor provides the same job management capabilities for Condor-G jobs as for other jobs. From Condor, a user may effectively submit jobs, manage jobs, and have jobs execute on widely distributed machines.

It may appear that Condor-G is a simple replacement for the Globus Toolkit's *globusrun* command. However, Condor-G does much more. It allows the submission of many jobs at once, along with the monitoring of those jobs with a convenient interface. There is notification when jobs complete or fail and maintenance of Globus credentials that may expire while a job is running. On top of this, Condor-G is a fault-tolerant system; if a machine crashes, all of these functions are again available as the machine returns.

Globus Protocols and Terminology

The Globus software provides a well-defined set of protocols that allow authentication, data transfer, and remote job execution. Authentication is a mechanism by which an identity is verified. Given proper authentication, authorization to use a resource is required. Authorization is a policy that determines who is allowed to do what.

Condor (and Globus) utilize the following protocols and terminology. The protocols allow Condor to interact with grid machines toward the end result of executing jobs.

GSI The Globus Toolkit's Grid Security Infrastructure (GSI) provides essential building blocks for other grid protocols and Condor-G. This authentication and authorization system makes it possible to authenticate a user just once, using public key infrastructure (PKI) mechanisms to verify a user-supplied grid credential. GSI then handles the mapping of the grid credential to

the diverse local credentials and authentication/authorization mechanisms that apply at each site.

GRAM The Grid Resource Allocation and Management (GRAM) protocol supports remote submission of a computational request (for example, to run a program) to a remote computational resource, and it supports subsequent monitoring and control of the computation. GRAM is the Globus protocol that Condor-G uses to talk to remote Globus jobmanagers.

GASS The Globus Toolkit's Global Access to Secondary Storage (GASS) service provides mechanisms for transferring data to and from a remote HTTP, FTP, or GASS server. GASS is used by Condor for the **gt2** grid type to transfer a job's files to and from the machine where the job is submitted and the remote resource.

GridFTP GridFTP is an extension of FTP that provides strong security and high-performance options for large data transfers. It is used with the **gt4** grid type to transfer the job's files between the machine where the job is submitted and the remote resource.

RSL RSL (Resource Specification Language) is the language GRAM accepts to specify job information.

gatekeeper A gatekeeper is a software daemon executing on a remote machine on the grid. It is relevant only to the **gt2** grid type, and this daemon handles the initial communication between Condor and a remote resource.

jobmanager A jobmanager is the Globus service that is initiated at a remote resource to submit, keep track of, and manage grid I/O for jobs running on an underlying batch system. There is a specific jobmanager for each type of batch system supported by Globus (examples are Condor, LSF, and PBS).

Figure 5.1 shows how Condor interacts with Globus software towards running jobs. The diagram is specific to the **gt2** type of grid. Condor contains a GASS server, used to transfer the executable, `stdin`, `stdout`, and `stderr` to and from the remote job execution site. Condor uses the GRAM protocol to contact the remote gatekeeper and request that a new jobmanager be started. The GRAM protocol is also used to when monitoring the job's progress. Condor detects and intelligently handles cases such as if the remote resource crashes.

There are now three different versions of the GRAM protocol. Condor supports both the **gt2** and **gt4** protocols. It does not support **gt3**.

gt2 This initial GRAM protocol is used in Globus Toolkit versions 1 and 2. It is still used by many production systems. Where available in the other, more recent versions of the protocol, **gt2** is referred to as the pre-web services GRAM (or pre-WS GRAM).

gt3 **gt3** corresponds to Globus Toolkit version 3 as part of Globus' shift to web services-based protocols. It is replaced by the Globus Toolkit version 4. An installation of the Globus Toolkit version 3 (or OSGA GRAM) may also include the the pre-web services GRAM.

gt4 This GRAM protocol was introduced in Globus Toolkit version 4.0 as a more standards-compliant version of the GT3 web services-based GRAM. It is also called WS GRAM. A

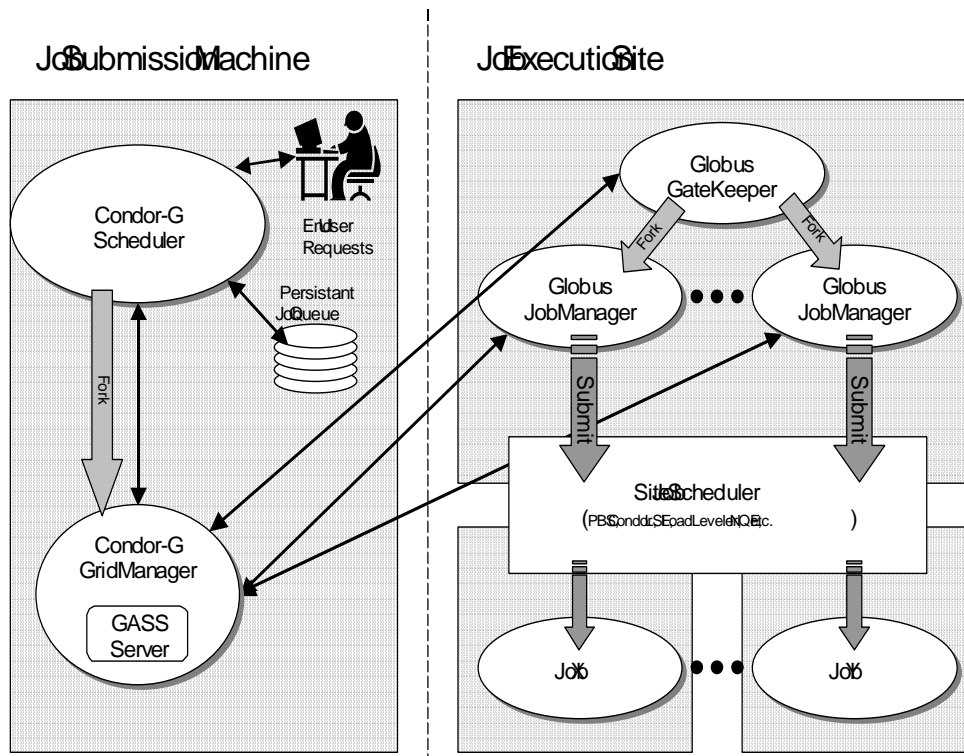


Figure 5.1: Condor-G interaction with Globus-managed resources

slightly different version of this GRAM protocol was introduced in Globus Toolkit 4.2 due to a change in the underlying Web Services standards. An installation of the Globus Toolkit version 4 may also include the the pre-web services GRAM.

The gt2 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit versions 1 and 2, also called the pre-web services GRAM (or pre-WS GRAM). These Condor-G jobs are submitted the same as any other Condor job. The **universe** is **grid**, and the pre-web services GRAM protocol is specified by setting the type of grid as **gt2** in the **grid_resource** command.

Under Condor, successful job submission to the **grid universe** with **gt2** requires credentials. An X.509 certificate is used to create a proxy, and an account, authorization, or allocation to use a grid resource is required. For general information on proxies and certificates, please consult the Globus page at

<http://www-unix.globus.org/toolkit/docs/4.0/security/key-index.html>

Before submitting a job to Condor under the **grid** universe, use *grid-proxy-init* to create a proxy.

Here is a simple submit description file. The example specifies a **gt2** job to be run on an NCSA machine.

```
executable = test
universe = grid
grid_resource = gt2 modi4.ncsa.uiuc.edu/jobmanager
output = test.out
log = test.log
queue
```

The **executable** for this example is transferred from the local machine to the remote machine. By default, Condor transfers the executable, as well as any files specified by an **input** command. Note that the executable must be compiled for its intended platform.

The command **grid_resource** is a required command for grid universe jobs. The second field specifies the scheduling software to be used on the remote resource. There is a specific jobmanager for each type of batch system supported by Globus. The full syntax for this command line appears as

```
grid_resource = gt2 machinename[:port]/jobmanagername[:X.509 distinguished name]
```

The portions of this syntax specification enclosed within square brackets ([and]) are optional. On a machine where the jobmanager is listening on a nonstandard port, include the port number. The jobmanagername is a site-specific string. The most common one is `jobmanager-fork`, but others are

```
jobmanager
jobmanager-condor
jobmanager-pbs
jobmanager-lsf
jobmanager-sge
```

The Globus software running on the remote resource uses this string to identify and select the correct service to perform. Other jobmanagername strings are used, where additional services are defined and implemented.

No input file is specified for this example job. Any output (file specified by an **output** command) or error (file specified by an **error** command) is transferred from the remote machine to the local machine as it is generated. This implies that these files may be incomplete in the case where the executable does not finish running on the remote resource. The ability to transfer standard output and standard error as they are produced may be disabled by adding to the submit description file:

```
stream_output = False
stream_error  = False
```

As a result, standard output and standard error will be transferred only after the job completes.

The job log file is maintained on the submit machine.

Example output from *condor_q* for this submission looks like:

```
% condor_q

-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

ID      OWNER      SUBMITTED      RUN_TIME ST PRI SIZE CMD
  7.0    smith      3/26 14:08    0+00:00:00 I  0   0.0  test

1 jobs; 1 idle, 0 running, 0 held
```

After a short time, the Globus resource accepts the job. Again running *condor_q* will now result in

```
% condor_q

-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

ID      OWNER      SUBMITTED      RUN_TIME ST PRI SIZE CMD
  7.0    smith      3/26 14:08    0+00:01:15 R  0   0.0  test

1 jobs; 0 idle, 1 running, 0 held
```

Then, very shortly after that, the queue will be empty again, because the job has finished:

```
% condor_q

-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

ID      OWNER      SUBMITTED      RUN_TIME ST PRI SIZE CMD

0 jobs; 0 idle, 0 running, 0 held
```

A second example of a submit description file runs the Unix *ls* program on a different Globus resource.

```
executable = /bin/ls
transfer_executable = false
universe = grid
grid_resource = gt2 vulture.cs.wisc.edu/jobmanager
output = ls-test.out
log = ls-test.log
queue
```

In this example, the executable (the binary) has been pre-staged. The executable is on the remote machine, and it is not to be transferred before execution. Note that the required **grid_resource** and **universe** commands are present. The command

```
transfer_executable = false
```

within the submit description file identifies the executable as being pre-staged. In this case, the **executable** command gives the path to the executable on the remote machine.

A third example submits a Perl script to be run as a submitted Condor job. The Perl script both lists and sets environment variables for a job. Save the following Perl script with the name `env-test.pl`, to be used as a Condor job executable.

```
#!/usr/bin/env perl

foreach $key (sort keys(%ENV))
{
    print "$key = $ENV{$key}\n"
}

exit 0;
```

Run the Unix command

```
chmod 755 env-test.pl
```

to make the Perl script executable.

Now create the following submit description file. Replace `example.cs.wisc.edu/jobmanager` with a resource you are authorized to use.

```
executable = env-test.pl
universe = grid
grid_resource = gt2 example.cs.wisc.edu/jobmanager
environment = foo=bar; zot=qux
output = env-test.out
log = env-test.log
queue
```

When the job has completed, the output file, `env-test.out`, should contain something like this:

```
GLOBUS_GRAM_JOB_CONTACT = https://example.cs.wisc.edu:36213/30905/1020633947/
GLOBUS_GRAM_MYJOB_CONTACT = URLx-nexus://example.cs.wisc.edu:36214
GLOBUS_LOCATION = /usr/local/globus
```

```

GLOBUS_REMOTE_IO_URL = /home/smith/.globus/.gass_cache/globus_gass_cache_1020633948
HOME = /home/smith
LANG = en_US
LOGNAME = smith
X509_USER_PROXY = /home/smith/.globus/.gass_cache/globus_gass_cache_1020633951
foo = bar
zot = qux

```

Of particular interest is the `GLOBUS_REMOTE_IO_URL` environment variable. Condor-G automatically starts up a GASS remote I/O server on the submit machine. Because of the potential for either side of the connection to fail, the URL for the server cannot be passed directly to the job. Instead, it is placed into a file, and the `GLOBUS_REMOTE_IO_URL` environment variable points to this file. Remote jobs can read this file and use the URL it contains to access the remote GASS server running inside Condor-G. If the location of the GASS server changes (for example, if Condor-G restarts), Condor-G will contact the Globus gatekeeper and update this file on the machine where the job is running. It is therefore important that all accesses to the remote GASS server check this file for the latest location.

The following example is a Perl script that uses the GASS server in Condor-G to copy input files to the execute machine. In this example, the remote job counts the number of lines in a file.

```

#!/usr/bin/env perl
use FileHandle;
use Cwd;

STDOUT->autoflush();
$gassUrl = `cat $ENV{GLOBUS_REMOTE_IO_URL}`;
chomp $gassUrl;

$ENV{LD_LIBRARY_PATH} = $ENV{GLOBUS_LOCATION}. "/lib";
$urlCopy = $ENV{GLOBUS_LOCATION}. "/bin/globus-url-copy";

# globus-url-copy needs a full path name
$pwd = getcwd();
print "$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts\n\n";
`$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts`;

open(file, "temporary.hosts");
while(<file>) {
    print $_;
}

exit 0;

```

The submit description file used to submit the Perl script as a Condor job appears as:

```

executable = gass-example.pl
universe = grid
grid_resource = gt2 example.cs.wisc.edu/jobmanager
output = gass.out
log = gass.log
queue

```

There are two optional submit description file commands of note: **x509userproxy** and **globus_rsl**. The **x509userproxy** command specifies the path to an X.509 proxy. The command is of the form:

```
x509userproxy = /path/to/proxy
```

If this optional command is not present in the submit description file, then Condor-G checks the value of the environment variable `X509_USER_PROXY` for the location of the proxy. If this environment variable is not present, then Condor-G looks for the proxy in the file `/tmp/x509up_uXXXX`, where the characters `XXXX` in this file name are replaced with the Unix user id.

The **globus_rsl** command is used to add additional attribute settings to a job's RSL string. The format of the **globus_rsl** command is

```
globus_rsl = (name=value)(name=value)
```

Here is an example of this command from a submit description file:

```
globus_rsl = (project=Test_Project)
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

The gt4 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit version 4, which speak a protocol called WS GRAM. Please note that this Globus Toolkit version is *not* compatible with the Globus Toolkit version 3.0 or 3.2. Globus Toolkit versions 4.0 and 4.2 use slightly different (and incompatible) versions of WS GRAM, due to a change in the underlying Web Services standards. Condor is able to detect the difference and use the appropriate version for each remote resource automatically. See <http://www-unix.globus.org/toolkit/docs/4.2/index.html> for more information about the Globus Toolkit version 4.2.

For grid jobs destined for **gt4**, the submit description file is much the same as for **gt2** jobs. The **grid_resource** command is still required, and is given in the form of a URL. The syntax follows the form:

```
grid_resource = gt4 [https://]hostname[:port][wsrf/services/ManagedJobFactoryService] scheduler-string
```

or

```
grid_resource = gt4 [https://]IPaddress[:port][wsrf/services/ManagedJobFactoryService] scheduler-string
```

The portions of this syntax specification enclosed within square brackets ([and]) are optional.

The **scheduler-string** field of **grid_resource** indicates which job execution system should to be used on the remote system, to execute the job. One of these values is substituted for **scheduler-string**:

```
Fork
Condor
PBS
LSF
SGE
```

The **globus_xml** command can be used to add additional attributes to the XML-based RSL string that Condor writes to submit the job to GRAM. Here is an example of this command from a submit description file:

```
globus_xml = <project>Test_Project</project>
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

File transfer occurs as expected for a Condor job (for the executable, input, and output), except that all output files other than `stdout` and `stderr` must be explicitly listed using **transfer_output_files**. The underlying transfer mechanism requires a *GridFTP* server to be running on the machine where the job is submitted. Condor will start one automatically. It will appear in the job queue as an additional job. It will leave the queue when there are no more **gt4** jobs in the queue. If the submit machine has a permanent *GridFTP* server running, instruct Condor to use it by setting the `GRIDFTP_URL_BASE` configuration variable. Here is an example setting:

```
GRIDFTP_URL_BASE = gsiftp://mycomp.foo.edu
```

On the submit machine, there is no requirement for any Globus Toolkit 4.0 components. Condor itself installs all necessary framework within the directory `$(LIB)/lib/gt4`. The machine where the job is submitted is required to have Java 1.5.0 or a higher version installed. You should not use the GNU Java interpreter (GCJ). The configuration variable `JAVA` must identify the location of the installation. See page 192 within section 3.3 for the complete description of the configuration variable `JAVA`.

Credential Management with *MyProxy*

Condor-G can use *MyProxy* software to automatically renew GSI proxies for **grid universe** jobs with grid type **gt2**. *MyProxy* is a software component developed at NCSA and used widely throughout the grid community. For more information see: <http://myproxy.ncsa.uiuc.edu/>

Difficulties with proxy expiration occur in two cases. The first case are long running jobs, which do not complete before the proxy expires. The second case occurs when great numbers

of jobs are submitted. Some of the jobs may not yet be started or not yet completed before the proxy expires. One proposed solution to these difficulties is to generate longer-lived proxies. This, however, presents a greater security problem. Remember that a GSI proxy is sent to the remote Globus resource. If a proxy falls into the hands of a malicious user at the remote site, the malicious user can impersonate the proxy owner for the duration of the proxy's lifetime. The longer the proxy's lifetime, the more time a malicious user has to misuse the owner's credentials. To minimize the window of opportunity of a malicious user, it is recommended that proxies have a short lifetime (on the order of several hours).

The *MyProxy* software generates proxies using credentials (a user certificate or a long-lived proxy) located on a secure *MyProxy* server. Condor-G talks to the *MyProxy* server, renewing a proxy as it is about to expire. Another advantage that this presents is it relieves the user from having to store a GSI user certificate and private key on the machine where jobs are submitted. This may be particularly important if a shared Condor-G submit machine is used by several users.

In the a typical case, the following steps occur:

1. The user creates a long-lived credential on a secure *MyProxy* server, using the *myproxy-init* command. Each organization generally has their own *MyProxy* server.
2. The user creates a short-lived proxy on a local submit machine, using *grid-proxy-init* or *myproxy-get-delegation*.
3. The user submits a Condor-G job, specifying:
MyProxy server name (host:port)
MyProxy credential name (optional)
MyProxy password
4. At the short-lived proxy expiration Condor-G talks to the *MyProxy* server to refresh the proxy.

Condor-G keeps track of the password to the *MyProxy* server for credential renewal. Although Condor-G tries to keep the password encrypted and secure, it is still possible (although highly unlikely) for the password to be intercepted from the Condor-G machine (more precisely, from the machine that the *condor_schedd* daemon that manages the grid universe jobs runs on, which may be distinct from the machine from where jobs are submitted). The following safeguard practices are recommended.

1. Provide time limits for credentials on the *MyProxy* server. The default is one week, but you may want to make it shorter.
2. Create several different *MyProxy* credentials, maybe as many as one for each submitted job. Each credential has a unique name, which is identified with the *MyProxyCredentialName* command in the submit description file.
3. Use the following options when initializing the credential on the *MyProxy* server:

```
myproxy-init -s <host> -x -r <cert subject> -k <cred name>
```

The option **-x -r** *<cert subject>* essentially tells the *MyProxy* server to require two forms of authentication:

- (a) a password (initially set with *myproxy-init*)
 - (b) an existing proxy (the proxy to be renewed)
4. A submit description file may include the password. An example contains commands of the form:

```
executable      = /usr/bin/my-executable
universe        = grid
grid_resource    = gt4 condor-unsup-7
MyProxyHost      = example.cs.wisc.edu:7512
MyProxyServerDN  = /O=doesciencegrid.org/OU=People/CN=Jane Doe 25900
MyProxyPassword  = password
MyProxyCredentialName = my_executable_run
queue
```

Note that placing the password within the submit file is not really secure, as it relies upon whatever file system security there is. This may still be better than option 5.

5. Use the **-p** option to *condor_submit*. The submit command appears as

```
condor_submit -p mypassword /home/user/myjob.submit
```

The argument list for *condor_submit* defaults to being publicly available. An attacker with a log in to the local machine could generate a simple shell script to watch for the password.

Currently, Condor-G calls the *myproxy-get-delegation* command-line tool, passing it the necessary arguments. The location of the *myproxy-get-delegation* executable is determined by the configuration variable `MYPROXY_GET_DELEGATION` in the configuration file on the Condor-G machine. This variable is read by the *condor_gridmanager*. If *myproxy-get-delegation* is a dynamically-linked executable (verify this with `ldd myproxy-get-delegation`), point `MYPROXY_GET_DELEGATION` to a wrapper shell script that sets `LD_LIBRARY_PATH` to the correct *MyProxy* library or Globus library directory and then calls *myproxy-get-delegation*. Here is an example of such a wrapper script:

```
#!/bin/sh
export LD_LIBRARY_PATH=/opt/myglobus/lib
exec /opt/myglobus/bin/myproxy-get-delegation $@
```

The Grid Monitor

Condor's Grid Monitor is designed to improve the scalability of machines running Globus Toolkit 2 gatekeepers. Normally, this gatekeeper runs a jobmanager process for every job submitted to the gatekeeper. This includes both currently running jobs and jobs waiting in the queue. Each

jobmanager runs a Perl script at frequent intervals (every 10 seconds) to poll the state of its job in the local batch system. For example, with 400 jobs submitted to a gatekeeper, there will be 400 jobmanagers running, each regularly starting a Perl script. When a large number of jobs have been submitted to a single gatekeeper, this frequent polling can heavily load the gatekeeper. When the gatekeeper is under heavy load, the system can become non-responsive, and a variety of problems can occur.

Condor's Grid Monitor temporarily replaces these jobmanagers. It is named the Grid Monitor, because it replaces the monitoring (polling) duties previously done by jobmanagers. When the Grid Monitor runs, Condor attempts to start a single process to poll all of a user's jobs at a given gatekeeper. While a job is waiting in the queue, but not yet running, Condor shuts down the associated jobmanager, and instead relies on the Grid Monitor to report changes in status. The jobmanager started to add the job to the remote batch system queue is shut down. The jobmanager restarts when the job begins running.

By default, standard output and standard error are streamed back to the submitting machine while the job is running. Streamed I/O requires the jobmanager. As a result, the Grid Monitor cannot replace the jobmanager for jobs that use streaming. If possible, disable streaming for all jobs; this is accomplished by placing the following lines in each job's submit description file:

```
stream_output = False
stream_error  = False
```

The Grid Monitor requires that the gatekeeper support the fork jobmanager with the name *jobmanager-fork*. If the gatekeeper does not support the fork jobmanager, the Grid Monitor will not be used for that site. The *condor_gridmanager* log file reports any problems using the Grid Monitor.

To enable the Grid Monitor, two variables are added to the Condor configuration file. The configuration macro `GRID_MONITOR` is already present in current distributions of Condor, but it may be missing from earlier versions of Condor. Also set the configuration macro `ENABLE_GRID_MONITOR` to `True`.

```
GRID_MONITOR      = $(SBIN)/grid_monitor.sh
ENABLE_GRID_MONITOR = TRUE
```

Limitations of Condor-G

Submitting jobs to run under the grid universe has not yet been perfected. The following is a list of known limitations:

1. No checkpoints.
2. No job exit codes. Job exit codes are not available when using **gt2**.
3. Limited platform availability. Windows support is not yet available.

5.3.3 The nordugrid Grid Type

NorduGrid is a project to develop free grid middleware named the Advanced Resource Connector (ARC). See the NorduGrid web page (<http://www.nordugrid.org>) for more information about NorduGrid software.

Condor jobs may be submitted to NorduGrid resources using the **grid** universe. The **grid_resource** command specifies the name of the NorduGrid resource as follows:

```
grid_resource = nordugrid ng.example.com
```

NorduGrid uses X.509 credentials for authentication, usually in the form a proxy certificate. For more information about proxies and certificates, please consult the Alliance PKI pages at <http://archive.ncsa.uiuc.edu/SCD/Alliance/GridSecurity/>. *condor_submit* looks in default locations for the proxy. The submit description file command **x509userproxy** is used to give the full path name to the directory containing the proxy, when the proxy is not in a default location. If this optional command is not present in the submit description file, then the value of the environment variable `X509_USER_PROXY` is checked for the location of the proxy. If this environment variable is not present, then the proxy in the file `/tmp/x509up_uXXXX` is used, where the characters `XXXX` in this file name are replaced with the Unix user id.

NorduGrid uses RSL syntax to describe jobs. The submit description file command **nordugrid_rsl** adds additional attributes to the job RSL that Condor constructs. The format this submit description file command is

```
nordugrid_rsl = (name=value)(name=value)
```

5.3.4 The unicon Grid Type

Unicon is a Java-based grid scheduling system. See <http://unicon.sourceforge.net> for more information about Unicon.

Condor jobs may be submitted to Unicon resources using the **grid** universe. The **grid_resource** command specifies the name of the Unicon resource as follows:

```
grid_resource = unicon usite.example.com vsite
```

usite.example.com is the host name of the Unicon gateway machine to which the Condor job is to be submitted. **vsite** is the name of the Unicon virtual resource to which the Condor job is to be submitted.

Unicon uses certificates stored in a Java keystore file for authentication. The following submit description file commands are required to properly use the keystore file.

keystore_file Specifies the complete path and file name of the Java keystore file to use.

keystore_alias A string that specifies which certificate in the Java keystore file to use.

keystore_passphrase_file Specifies the complete path and file name of the file containing the passphrase protecting the certificate in the Java keystore file.

5.3.5 The pbs Grid Type

The popular PBS (Portable Batch System) comes in several varieties: OpenPBS (<http://www.openpbs.org>), PBS Pro (<http://www.altair.com/software/pbspro.htm>), and Torque (<http://www.clusterresources.com/pages/products/torque-resource-manager.php>).

Condor jobs are submitted to a local PBS system using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = pbs
```

The pbs grid type requires two variables to be set in the Condor configuration file. PBS_GAHP is the path to the PBS GAHP server binary that is to be used to submit PBS jobs. GLITE_LOCATION is the path to the directory containing the GAHP's configuration file and auxillary binaries. In the Condor distribution, these files are located in `$(LIB)/glite`. The PBS GAHP's configuration file is in `$(GLITE_LOCATION)/etc/batch_gahp.config`. The PBS GAHP's auxillary binaries are to be in the directory `$(GLITE_LOCATION)/bin`. The Condor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
PBS_GAHP       = $(GLITE_LOCATION)/bin/batch_gahp
```

The PBS GAHP's configuration file contains two variables that must be modified to tell it where to find PBS on the local system. `pbs_binpath` is the directory that contains the PBS binaries. `pbs_spoolpath` is the PBS spool directory.

5.3.6 The lsf Grid Type

Condor jobs may be submitted to the Platform LSF batch system. See the Products page of the Platform web page at <http://www.platform.com/Products/> for more information about Platform LSF.

Condor jobs are submitted to a local Platform LSF system using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = lsf
```

The lsf grid type requires two variables to be set in the Condor configuration file. LSF_GAHP is the path to the LSF GAHP server binary that is to be used to submit Platform LSF jobs.

`GLITE_LOCATION` is the path to the directory containing the GAHP's configuration file and auxiliary binaries. In the Condor distribution, these files are located in `$(LIB)/glite`. The LSF GAHP's configuration file is in `$(GLITE_LOCATION)/etc/batch_gahp.config`. The LSF GAHP's auxiliary binaries are to be in the directory `$(GLITE_LOCATION)/bin`. The Condor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
LSF_GAHP       = $(GLITE_LOCATION)/bin/batch_gahp
```

The LSF GAHP's configuration file contains two variables that must be modified to tell it where to find LSF on the local system. `lsf_binpath` is the directory that contains the LSF binaries. `lsf_confpath` is the location of the LSF configuration file.

5.3.7 The amazon Grid Type

Condor jobs may be submitted to Amazon's Elastic Compute Cloud (EC2) service. EC2 is an on-line commercial service that allows you to rent computers by the hour to run computational applications. EC2 runs virtual machine images that have been uploaded to Amazon's online storage service (S3). See the Amazon EC2 webpage at <http://aws.amazon.com/ec2> for more information about EC2.

Amazon EC2 Job Submission

Condor jobs are submitted to EC2 using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = amazon
```

Since the job is a virtual machine image, most of the submit file attributes specifying input or output files aren't applicable. The **executable** attribute is still required, but its value is ignored. You can use it to identify different jobs in the output of *condor_q*.

The VM image for the job must already reside in Amazon's storage service (S3) and be registered with EC2. In the submit file, you must provide the identifier for the image using the **amazon_ami_id** attribute.

You must also provide the files containing the X509 certificate and private key used to authenticate you with the EC2 service:

```
amazon_public_key = /path/to/x509/cert
amazon_private_key = /path/to/private/key
```

Condor and EC2 can create an ssh keypair to allow you to securely log into the virtual machine once it's running. If you set **amazon_keypair_file** in the submit file, Condor will write an ssh private

key into the indicated file. The key can be used to log into the virtual machine. Note that you'll also need to modify the firewall rules for the job to all incoming ssh connections.

EC2 uses a firewall to restrict network access to the virtual machine instances it runs. By default, no incoming connections are allowed. You can define sets of firewall rules and give them names. EC2 calls these security groups. You can then tell Condor what set of security groups should be applied to each VM using the **amazon_security_groups** attribute. If you don't provide this attribute, Condor uses the security group **default**.

EC2 offers several hardware configurations for instances to run on, with varying prices. You can select which configuration to use with the **amazon_instance_type** attribute. The default value is **m1.small**.

Each virtual machine instance can be given up to 16kB of unique data, accessible by the instance by connecting to a well-known address. This makes it easy for many instances to share the same VM image but perform different work. This data can be specified to Condor in one of two ways. First, the data can be provided directly in the submit file using the **amazon_user_data** attribute. Second, the data can be stored in a file, and the filename specified with the **amazon_user_data_file** attribute. The second option allows you to use binary data. Condor performs the base64 encoding that EC2 expects on the data.

Amazon EC2 Configuration Parameters

The amazon grid type requires several parameters to be set in the Condor configuration file:

```
AMAZON_GAHP=$(SBIN)/amazon-gahp
AMAZON_GAHP_LOG=/tmp/AmazonGahpLog.$(USERNAME)
```

You can alter the URL used by Condor to contact the EC2 service using the **AMAZON_EC2_URL** parameter. The default value is `https://ec2.amazonaws.com/`.

If you need to use an HTTP proxy to reach EC2, you tell Condor to use it via the **AMAZON_HTTP_PROXY** parameter.

5.3.8 Matchmaking in the Grid Universe

In a simple usage, the grid universe allows users to specify a single grid site as a destination for jobs. This is sufficient when a user knows exactly which grid site they wish to use, or a higher-level resource broker (such as the European Data Grid's resource broker) has decided which grid site should be used.

When a user has a variety of grid sites to choose from, Condor allows matchmaking of grid universe jobs to decide which grid resource a job should run on. Please note that this form of matchmaking is relatively new. There are some rough edges as continual improvement occurs.

To facilitate Condor's matching of jobs with grid resources, both the jobs and the grid resources are involved. The job's submit description file provides all commands needed to make the job work on a matched grid resource. The grid resource identifies itself to Condor by advertising a ClassAd. This ClassAd specifies all necessary attributes, such that Condor can properly make matches. The grid resource identification is accomplished by using *condor_advertise* to send a ClassAd representing the grid resource, which is then used by Condor to make matches.

Job Submission

To submit a grid universe job intended for a single, specific **gt2** resource, the submit description file for the job explicitly specifies the resource:

```
grid_resource = gt2 grid.example.com/jobmanager-pbs
```

If there were multiple **gt2** resources that might be matched to the job, the submit description file changes:

```
grid_resource      = $$ (resource_name)
requirements      = TARGET.resource_name != UNDEFINED
```

The **grid_resource** command uses a substitution macro. The substitution macro defines the value of `resource_name` using attributes as specified by the matched grid resource. The **requirements** command further restricts that the job may only run on a machine (grid resource) that defines `grid_resource`. Note that this attribute name is invented for this example. To make matchmaking work in this way, both the job (as used here within the submit description file) and the grid resource (in its created and advertised ClassAd) must agree upon the name of the attribute.

As a more complex example, consider a job that wants to run not only on a **gt2** resource, but on one that has the Bamboozle software installed. The complete submit description file might appear:

```
universe          = grid
executable        = analyze_bamboozle_data
output            = aaa.$(Cluster).out
error             = aaa.$(Cluster).err
log               = aaa.log
grid_resource     = $$ (resource_name)
requirements      = (TARGET.HaveBamboozle == True) && (TARGET.resource_name != UNDEFINED)
queue
```

Any grid resource which has the `HaveBamboozle` attribute defined as well as set to `True` is further checked to have the `resource_name` attribute defined. Where this occurs, a match may be made (from the job's point of view). A grid resource that has one of these attributes defined, but not the other results in no match being made.

Note that the entire value of **grid_resource** comes from the grid resource's ad. This means that the job can be matched with a resource of any type, not just **gt2**.

Advertising Grid Resources to Condor

Any grid resource that wishes to be matched by Condor with a job must advertise itself to Condor using a ClassAd. To properly advertise, a ClassAd is sent periodically to the *condor_collector* daemon. A ClassAd is a list of pairs, where each pair consists of an attribute name and value that describes an entity. There are two entities relevant to Condor: a job, and a machine. A grid resource is a machine. The ClassAd describes the grid resource, as well as identifying the capabilities of the grid resource. It may also state both requirements and preferences (called **rank**) for the jobs it will run. See Section 2.3 for an overview of the interaction between matchmaking and ClassAds. A list of common machine ClassAd attributes is given in the Appendix on page 886.

To advertise a grid site, place the attributes in a file. Here is a sample ClassAd that describes a grid resource that is capable of running a **gt2** job.

```
# example grid resource ClassAd for a gt2 job
MyType           = "Machine"
TargetType       = "Job"
Name             = "Example1_Gatekeeper"
Machine          = "Example1_Gatekeeper"
resource_name    = "gt2_grid.example.com/jobmanager-pbs"
UpdateSequenceNumber = 4
Requirements     = (TARGET.JobUniverse == 9)
Rank             = 0.000000
CurrentRank      = 0.000000
```

Some attributes are defined as expressions, while others are integers, floating point values, or strings. The type is important, and must be correct for the ClassAd to be effective. The attributes

```
MyType           = "Machine"
TargetType       = "Job"
```

identify the grid resource as a machine, and that the machine is to be matched with a job. In Condor, machines are matched with jobs, and jobs are matched with machines. These attributes are strings. Strings are surrounded by double quote marks.

The attributes *Name* and *Machine* are likely to be defined to be the same string value as in the example:

```
Name             = "Example1_Gatekeeper"
Machine          = "Example1_Gatekeeper"
```

Both give the fully qualified host name for the resource. The *Name* may be different on an SMP machine, where the individual CPUs are given names that can be distinguished from each other. Each separate grid resource must have a unique name.

Where the job depends on the resource to specify the value of the **grid_resource** command by the use of the substitution macro, the ClassAd for the grid resource (machine) defines this value. The example given as

```
grid_resource = "gt2 grid.example.com/jobmanager-pbs"
```

defines this value. Note that the invented name of this variable must match the one utilized within the submit description file. To make the matchmaking work, both the job (as used within the submit description file) and the grid resource (in this created and advertised ClassAd) must agree upon the name of the attribute.

A machine's ClassAd information can be time sensitive, and may change over time. Therefore, ClassAds expire and are thrown away. In addition, the communication method by which ClassAds are sent implies that entire ads may be lost without notice or may arrive out of order. Out of order arrival leads to the definition of an attribute which provides an ordering. This positive integer value is given in the example ClassAd as

```
UpdateSequenceNumber = 4
```

This value must increase for each subsequent ClassAd. If state information for the ClassAd is kept in a file, a script executed each time the ClassAd is to be sent may use a counter for this value. An alternative for a stateless implementation sends the current time in seconds (since the epoch, as given by the C `time()` function call).

The requirements that the grid resource sets for any job that it will accept are given as

```
Requirements = (TARGET.JobUniverse == 9)
```

This set of requirements state that any job is required to be for the **grid** universe.

The attributes

```
Rank = 0.000000
CurrentRank = 0.000000
```

are both necessary for Condor's negotiation to proceed, but are not relevant to grid matchmaking. Set both to the floating point value 0.0.

The example machine ClassAd becomes more complex for the case where the grid resource allows matches with more than one job:

```
# example grid resource ClassAd for a gt2 job
MyType = "Machine"
TargetType = "Job"
Name = "Example1_Gatekeeper"
Machine = "Example1_Gatekeeper"
resource_name = "gt2 grid.example.com/jobmanager-pbs"
UpdateSequenceNumber = 4
Requirements = (CurMatches < 10) && (TARGET.JobUniverse == 9)
Rank = 0.000000
CurrentRank = 0.000000
WantAdReevaluate = True
CurMatches = 1
```

In this example, the two attributes `WantAdReevaluate` and `CurMatches` appear, and the `Requirements` expression has changed.

`WantAdReevaluate` is a boolean value, and may be set to either `True` or `False`. When `True` in the `ClassAd` and a match is made (of a job to the grid resource), the machine (grid resource) is not removed from the set of machines to be considered for further matches. This implements the ability for a single grid resource to be matched to more than one job at a time. Note that the spelling of this attribute is incorrect, and remains incorrect to maintain backward compatibility.

To limit the number of matches made to the single grid resource, the resource must have the ability to keep track of the number of Condor jobs it has. This integer value is given as the `CurMatches` attribute in the advertised `ClassAd`. It is then compared in order to limit the number of jobs matched with the grid resource.

```
Requirements    = (CurMatches < 10) && (TARGET.JobUniverse == 9)
CurMatches     = 1
```

This example assumes that the grid resource already has one job, and is willing to accept a maximum of 9 jobs. If `CurMatches` does not appear in the `ClassAd`, Condor uses a default value of 0.

For multiple matching of a site `ClassAd` to work correctly, it is also necessary to add the following to the configuration file read by the *condor_negotiator*:

```
NEGOTIATOR_MATCHLIST_CACHING = False
NEGOTIATOR_IGNORE_USER_PRIORITIES = True
```

This `ClassAd` (likely in a file) is to be periodically sent to the *condor_collector* daemon using *condor_advertise*. A recommended implementation uses a script to create or modify the `ClassAd` together with *cron* to send the `ClassAd` every five minutes. The *condor_advertise* program must be installed on the machine sending the `ClassAd`, but the remainder of Condor does not need to be installed. The required argument for the *condor_advertise* command is *UPDATE_STARTD_AD*.

condor_advertise uses UDP to transmit the `ClassAd`. Where this is insufficient, specify the **-tcp** option to *condor_advertise* to use TCP for communication.

Advanced usage

What if a job fails to run at a grid site due to an error? It will be returned to the queue, and Condor will attempt to match it and re-run it at another site. Condor isn't very clever about avoiding sites that may be bad, but you can give it some assistance. Let's say that you want to avoid running at the last grid site you ran at. You could add this to your job description:

```
match_list_length = 1
Rank              = TARGET.Name != LastMatchName0
```

This will prefer to run at a grid site that was not just tried, but it will allow the job to be run there if there is no other option.

When you specify **match_list_length**, you provide an integer N, and Condor will keep track of the last N matches. The oldest match will be LastMatchName0, and next oldest will be LastMatchName1, and so on. (See the *condor_submit* manual page for more details.) The Rank expression allows you to specify a numerical ranking for different matches. When combined with **match_list_length**, you can prefer to avoid sites that you have already run at.

In addition, *condor_submit* has two options to help you control grid universe job resubmissions and rematching. See **globus_resubmit** and **globus_rematch** in the *condor_submit* manual page. These options are independent of **match_list_length**.

There are some new attributes that will be added to the Job ClassAd, and may be useful to you when you write your rank, requirements, *globus_resubmit* or *globus_rematch* option. Please refer to the Appendix on page 879 to see a list containing the following attributes:

- NumJobMatches
- NumGlobusSubmits
- NumSystemHolds
- HoldReason
- ReleaseReason
- EnteredCurrentStatus
- LastMatchTime
- LastRejMatchTime
- LastRejMatchReason

The following example of a command within the submit description file releases jobs 5 minutes after being held, increasing the time between releases by 5 minutes each time. It will continue to retry up to 4 times per Globus submission, plus 4. The plus 4 is necessary in case the job goes on hold before being submitted to Globus, although this is unlikely.

```
periodic_release = ( NumSystemHolds <= ((NumGlobusSubmits * 4) + 4) ) \
  && (NumGlobusSubmits < 4) && \
  ( HoldReason != "via condor_hold (by user $ENV(USER))" ) && \
  ((CurrentTime - EnteredCurrentStatus) > ( NumSystemHolds *60*5 ))
```

The following example forces Globus resubmission after a job has been held 4 times per Globus submission.

```
globus_resubmit = NumSystemHolds == (NumGlobusSubmits + 1) * 4
```

If you are concerned about unknown or malicious grid sites reporting to your *condor_collector*, you should use Condor's security options, documented in Section 3.6.

5.4 Glidein

Glidein is a mechanism by which one or more grid resources (remote machines) temporarily join a local Condor pool. The program *condor_glidein* is used to add a machine to a Condor pool. During the period of time when the added resource is part of the local pool, the resource is visible to users of the pool. But, by default, the resource is only available for use by the user that added the resource to the pool.

After glidein, the user may submit jobs for execution on the added resource the same way that all Condor jobs are submitted. To force a submitted job to run on the added resource, the submit description file could contain a requirement that the job run specifically on the added resource.

5.4.1 What *condor_glidein* Does

condor_glidein works by installing and executing necessary Condor daemons and configuration on the remote resource, such that the resource reports to and joins the local pool. *condor_glidein* accomplishes two separate tasks towards having a remote grid resource join the local Condor pool. They are the set up task and the execution task.

The set up task generates necessary configuration files and locates proper platform-dependent binaries for the Condor daemons. A script is also generated that can be used during the execution task to invoke the proper Condor daemons. These files are copied to the remote resource as necessary. The configuration variable `GLIDEIN_SERVER_URLS` defines a list of locations from which the necessary binaries are obtained. Default values cause binaries to be downloaded from the UW site. See section 3.3.24 on page 223 for a full definition of this configuration variable.

When the files are correctly in place, the execution task starts the Condor daemons. *condor_glidein* does this by submitting a Condor job to run under the grid universe. The job runs the *condor_master* on the remote grid resource. The *condor_master* invokes other daemons, which contact the local pool's *condor_collector* to join the pool. The Condor daemons exit gracefully when no jobs run on the daemons for a preset period of time.

Here is an example of how a glidein resource appears, similar to how any other machine appears. The name has a slightly different form, in order to handle the possibility of multiple instances of glidein daemons inhabiting a multi-processor machine.

```
% condor_status | grep denal
7591386@denal LINUX          INTEL Unclaimed Idle      3.700  24064  0+00:06:35
```

5.4.2 Configuration Requirements in the Local Pool

As remote grid resources join the local pool, these resources must report to the local pool's *condor_collector* daemon. Security demands that the local pool's *condor_collector* list all hosts from

which they will accept communication. Therefore, all remote grid resources accepted for glidein must be given `HOSTALLOW_WRITE` permission. An expected way to do this is to modify the empty variable (within the sample configuration file) `GLIDEIN_SITES` to list all remote grid resources accepted for glidein. The list is a space or comma separated list of hosts. This list is then given the proper permissions by an additional redefinition of the `HOSTALLOW_WRITE` configuration variable, to also include the list of hosts as in the following example.

```
GLIDEIN_SITES = A.example.com, B.example.com, C.example.com
HOSTALLOW_WRITE = $(HOSTALLOW_WRITE) $(GLIDEIN_SITES)
```

Recall that for configuration file changes to take effect, *condor_reconfig* must be run.

If this configuration change to the security settings on the local Condor pool cannot be made, an additional Condor pool that utilizes personal Condor may be defined. The single machine pool may coexist with other instances of Condor. *condor_glidein* is executed to have the remote grid resources join this personal Condor pool.

5.4.3 Running Jobs on the Remote Grid Resource After Glidein

Once the Globus resource has been added to the local Condor pool with *condor_glidein*, job(s) may be submitted. To force a job to run on the Globus resource, specify that Globus resource as a machine requirement in the submit description file. Here is an example from within the submit description file that forces submission to the Globus resource `denali.mcs.anl.gov`:

```
requirements = ( machine == "denali.mcs.anl.gov" ) \
    && FileSystemDomain != " " \
    && Arch != " " && OpSys != " "
```

This example requires that the job run only on `denali.mcs.anl.gov`, and it prevents Condor from inserting the file system domain, architecture, and operating system attributes as requirements in the matchmaking process. Condor must be told not to use the submission machine's attributes in those cases where the Globus resource's attributes do not match the submission machine's attributes and your job really is capable of running on the target machine. You may want to use Condor's file-transfer capabilities in order to copy input and output files back and forth between the submission and execution machine.

5.5 Dynamic Deployment

See section 3.2.9 for a complete description of Condor's dynamic deployment tools.

Condor's dynamic deployment tools (*condor_cold_start* and *condor_glidein*) allow new pools of resources to be incorporated on the fly. While Condor is able to manage compute jobs remotely

through Globus and other grid-computing protocols, dynamic deployment of Condor makes it possible to go one step further. Condor remotely installs and runs portions of itself. This process of Condor gliding in to inhabit computing resources on demand leverages the lowest common denominator of grid middleware systems, simple program execution, to bind together resources in a heterogeneous computing grid, with different management policies and different job execution methods, into a full-fledged Condor system.

The mobility of Condor services also benefits from the development of Condor-C, which provides a richer tool set for interlinking Condor-managed computers. Condor-C is a protocol that allows one Condor scheduler to delegate jobs to another Condor scheduler. The second scheduler could be at a remote site and/or an entry point into a restricted network. Delegating details of managing a job achieves greater flexibility with respect to network architecture, as well as fault tolerance and scalability. In the context of glide in deployments, the beach-head for each compute site is a dynamically deployed Condor scheduler which then serves as a target for Condor-C traffic.

In general, the mobility of the Condor scheduler and job execution agents, and the flexibility in how these are interconnected provide a uniform and feature-rich platform that can expand onto diverse resources and environments when the user requires it.

5.6 The Condor Job Router

The Condor Job Router is an add-on to `condor_schedd` that transforms jobs from one type into another according to a configurable policy. This process of transforming the jobs is called *job routing*.

One example of how the Job Router can be used is the task of sending excess vanilla universe jobs to one or more remote grid sites. The Job Router can transform the vanilla universe jobs into grid universe jobs that use any of the grid types supported by Condor. The rate at which jobs are routed can be matched roughly to the rate at which the site is able to start running them. This makes it possible to balance a large work flow across multiple grid sites, a local condor pool, and any flocked condor pools, without having to guess in advance how quickly jobs will run and complete in each of the different sites.

Job Routing is most appropriate for high throughput work flows, where there are many more jobs than computers, and the goal is to keep as many of the computers busy as possible. Job Routing is less suitable when there are a small number of jobs, and the scheduler needs to choose the best place for each job, in order to finish them as quickly as possible. The Job Router does not know which site will run the jobs faster, but it can decide whether to send more jobs to a site, based on whether jobs already submitted to that site are sitting idle or not, as well as whether the site has experienced recent job failures.

5.6.1 Routing Mechanism

The *condor_job_router* daemon and configuration determine a policy for which vanilla universe jobs may be transformed and sent to grid sites. A vanilla universe job is transformed into a grid universe job by making a copy of the original job ClassAd, modifying some attributes of the job. The copy is called the routed copy, and it shows up in the job queue under a new job id.

Until the routed copy finishes or is removed, the original copy of the job passively mirrors the state of the routed job. During this time, the original job is not available for matchmaking as a normal vanilla universe job, because it is tied to the routed copy. The original jobs also does not evaluate periodic expressions, such as `PeriodicHold`. Periodic expressions are evaluated for the routed copy. When the routed copy completes, the original job ClassAd is updated such that it reflects the final status of the job. If the routed copy is removed, the original job returns to the normal idle state, and is available for matchmaking or rerouting. If, instead, the original job is removed or goes on hold, the routed copy is removed.

The *condor_job_router* daemon utilizes a *routing table*, in which a ClassAd describes each site to where jobs may be sent. The routing table is given using *New ClassAds*, a re-implementation of the current ClassAd language, as currently used by Condor. A good place to learn about the syntax of New ClassAds is the Informal Language Description in the C++ ClassAds tutorial: <http://www.cs.wisc.edu/condor/classad/c++tut.html>. With the exception of a few ClassAd functions, everything in the current ClassAd language is supported by the New ClassAd language. A job that requires the unsupported ClassAd functions (those with names beginning with `stringList`) cannot be considered for Job Routing.

Entries in the routing table are described with the New ClassAd language, but the *condor_job_router* daemon must interact with the remainder of Condor which utilizes the current ClassAd language. A `requirements` attribute in the routing table is given with the New ClassAd language. An attribute for a job, as represented in the job queue uses the current ClassAd language.

Two essential differences distinguish the New ClassAd language from the current one. In the New ClassAd language, each ClassAd is surrounded by square brackets. And, in the New ClassAd language, each assignment statement ends with a semicolon. When the New ClassAd is embedded in a Condor configuration file, it may appear all on a single line, but the readability is often improved by inserting line continuation characters after each assignment statement. This is done in the examples. Unfortunately, this makes the insertion of comments into the configuration file awkward, because of the interaction between comments and line continuation characters in configuration files. An alternative is to use C-style comments (`/* ... */`). Another alternative is to read in the routing table entries from a separate file, rather than embedding them in the Condor configuration file.

5.6.2 Job Submission with Job Routing Capability

If Job Routing is set up, then the following items ought to be considered for vanilla universe jobs to have the necessary prerequisites to be considered for routing.

- Jobs appropriate for routing to the grid must not rely on access to a shared file system, or other services that are only available on the local pool. The job will use Condor's file transfer mechanism, rather than relying on a shared file system to access input files and write output files. In the submit description file, to enable file transfer, there will be a set of commands similar to

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = input1, input2
transfer_output_files = output1, output2
```

Vanilla universe jobs and grid universe jobs differ in the set of files transferred back when the job completes. Vanilla universe jobs transfer back all files created or modified, while grid universe jobs only transfer back the file mapped with **output**, as well as those explicitly listed with **transfer_output_files**. To work properly as a routed copy, explicitly specify all output files that must be transferred upon job completion.

An additional difference between the vanilla universe jobs and **gt2** grid universe jobs is that **gt2** jobs do not return any information about the job's exit status. The exit status as reported in the job ClassAd and user log are always 0. Therefore, jobs that may be routed to a **gt2** grid site must not rely upon a non-zero job exit status.

- One configuration for routed jobs requires the jobs to identify themselves as candidates for Job Routing. This may be accomplished by inventing a ClassAd attribute that the configuration utilizes in setting the policy for job identification, and the job defines this attribute to identify itself. If the invented attribute is called `WantJobRouter`, then the job identifies itself as a job that may be routed by placing in the submit description file:

```
+WantJobRouter = True
```

This implementation can be taken further, allowing the job to first be rejected within the local pool, before being a candidate for Job Routing:

```
+WantJobRouter = LastRejMatchTime != UNDEFINED
```

- As appropriate to the potential grid site, create a grid proxy, and specify it in the submit description file:

```
x509userproxy = /tmp/x509up_u275
```

This is not necessary if the *condor-job-router* daemon is configured to add a grid proxy on behalf of jobs.

Job submission does not change for jobs that may be routed.

```
$ condor_submit job1.sub
```

where `job1.sub` might contain:

```
universe = vanilla
executable = my_executable
output = job1.stdout
error = job1.stderr
log = job1.ulong
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
+WantJobRouter = LastRejMatchTime != UNDEFINED
x509userproxy = /tmp/x509up_u275
queue
```

The status of the job may be observed as with any other Condor job, for example by looking in the job's log file. Before the job completes, *condor_q* shows the job's status. Should the job become routed, a second job will enter the job queue. This is the routed copy of the original job. The command *condor_router_q* shows a more specialized view of routed jobs, as this example shows:

```
$ condor_router_q -S
   JOBS ST Route      GridResource
     40  I Site1      site1.edu/jobmanager-condor
     10  I Site2      site2.edu/jobmanager-pbs
       2  R Site3      condor submit.site3.edu condor.site3.edu
```

condor_router_history summarizes the history of routed jobs, as this example shows:

```
$ condor_router_history
Routed job history from 2007-06-27 23:38 to 2007-06-28 23:38
```

Site	Hours	Jobs Completed	Runs Aborted
Site1	10	2	0
Site2	8	2	1
Site3	40	6	0
TOTAL	58	10	1

5.6.3 An Example Configuration

The following sample configuration sets up potential job routing to three routes (grid sites). Definitions of the configuration variables specific to the Job Router are in section 3.3.21. One route is a Condor site accessed via the Globus gt2 protocol. A second route is a PBS site, also accessed via

Globus gt2. The third site is a Condor site accessed by Condor-C. The *condor_job_router* daemon does not know which site will be best for a given job. The policy implemented in this sample configuration stops sending more jobs to a site, if ten jobs that have already been sent to that site are idle.

These configuration settings belong in the local configuration file of the machine where jobs are submitted. Check that the machine can successfully submit grid jobs before setting up and using the Job Router. Typically, the single required element that needs to be added for GSI authentication is an X.509 trusted certification authority directory, in a place recognized by Condor (for example, */etc/grid-security/certificates*). The VDT (<http://vdt.cs.wisc.edu>) project provides a convenient way to set up and install a trusted CAs, if needed.

```
# These settings become the default settings for all routes
JOB_ROUTER_DEFAULTS = \
[ \
  requirements=target.WantJobRouter is True; \
  MaxIdleJobs = 10; \
  MaxJobs = 200; \
\
  /* now modify routed job attributes */ \
  /* remove routed job if it goes on hold or stays idle for over 6 hours */ \
  set_PeriodicRemove = JobStatus == 5 || \
                      (JobStatus == 1 && (CurrentTime - QDate) > 3600*6); \
  delete_WantJobRouter = true; \
  set_requirements = true; \
]

# This could be made an attribute of the job, rather than being hard-coded
ROUTED_JOB_MAX_TIME = 1440

# Now we define each of the routes to send jobs on
JOB_ROUTER_ENTRIES = \
[ GridResource = "gt2 site1.edu/jobmanager-condor"; \
  name = "Site 1"; \
] \
[ GridResource = "gt2 site2.edu/jobmanager-pbs"; \
  name = "Site 2"; \
  set_GlobusRSL = "(maxwalltime=$(ROUTED_JOB_MAX_TIME))(jobType=single)"; \
] \
[ GridResource = "condor submit.site3.edu condor.site3.edu"; \
  name = "Site 3"; \
  set_remote_jobuniverse = 5; \
]

# Reminder: you must restart Condor for changes to DAEMON_LIST to take effect.
DAEMON_LIST = $(DAEMON_LIST) JOB_ROUTER

# For testing, set this to a small value to speed things up.
# Once you are running at large scale, set it to a higher value
# to prevent the JobRouter from using too much cpu.
JOB_ROUTER_POLLING_PERIOD = 10

#It is good to save lots of schedd queue history
```

```
#for use with the router_history command.  
MAX_HISTORY_ROTATIONS = 20
```

5.6.4 Routing Table Entry ClassAd Attributes

The conversion of a vanilla job to a routed copy requires the job ClassAd to be modified. Defaults for routed copy attributes, as well as instructions for creating needed attributes are listed in the Routing Table.

The following attributes and instructions for creating attributes may appear in a Routing Table entry.

GridResource Specifies the value for the `GridResource` attribute that will be inserted into the routed copy ClassAd.

Name An optional identifier that will be used in log messages concerning this route. If no name is specified, the default used will be the value of `GridResource`. The *condor-job-router* distinguishes routes and advertises statistics based on this attribute's value.

Requirements A `Requirements` expression in New ClassAd syntax that identifies jobs that may be matched to the route. Note that, as with all settings, requirements specified in the configuration variable `JOB_ROUTER_ENTRIES` override the setting of `JOB_ROUTER_DEFAULTS`. To specify global requirements that are not overridden by `JOB_ROUTER_ENTRIES`, use `JOB_ROUTER_SOURCE_JOB_CONSTRAINT`.

MaxJobs An integer maximum number of jobs permitted on the route at one time. The default is 100.

MaxIdleJobs An integer maximum number of routed jobs in the idle state. At or above this value, no more jobs will be sent to this site. This is intended to prevent too many jobs from being sent to sites which are too busy to run them. If the value set for this attribute is too small, the rate of job submission to the site will slow, because the *condor-job-router* daemon will submit jobs up to this limit, wait to see some of the jobs enter the running state, and then submit more. The disadvantage of setting this attribute's value too high is that a lot of jobs may be sent to a site, only to sit idle for hours or days. The default value is 50.

FailureRateThreshold A maximum tolerated rate of job failures. Failure is determined by the expression sets for the attribute `JobFailureTest` expression. The default threshold is 0.03 jobs/second. If the threshold is exceeded, submission of new jobs is throttled until jobs begin succeeding, such that the failure rate is less than the threshold. This attribute implements *black hole throttling*, such that a site at which jobs are sent only to fail (a black hole) receives fewer jobs.

JobFailureTest An expression in New ClassAds syntax evaluated for each job that finishes, to determine whether it was a failure. The default value if no expression is defined assumes all jobs are successful. Routed jobs that are removed are considered to be failures. An example expression to treat all jobs running for less than 30 minutes as failures

is `other.RemoteWallClockTime < 1800`. A more flexible expression might reference a property or expression of the job that specifies a failure condition specific to the type of job.

TargetUniverse An integer value specifying the desired universe for the routed copy of the job. The default value is 9, which is the **grid** universe.

UseSharedX509UserProxy A boolean expression in New ClassAds syntax, that when `True` causes the value of `SharedX509UserProxy` to be the X.509 user proxy for the routed job. Note that if the *condor-job-router* daemon is running as root, the copy of this file that is given to the job will have its ownership set to that of the user running the job. This requires the trust of the user. It is therefore recommended to avoid this mechanism when possible. Instead, require users to submit jobs with `X509UserProxy` set in the submit description file. If this feature is needed, use the boolean expression to only allow specific values of `other.Owner` to use this shared proxy file. The shared proxy file should be owned by the *condor* user. Currently, to use a shared proxy, the job must also turn on sandboxing by having the attribute `JobShouldBeSandboxed`.

SharedX509UserProxy A string representing file containing the X.509 user proxy for the routed job.

JobShouldBeSandboxed A boolean expression in New ClassAd syntax, that when `True` causes the created copy of the job to be sandboxed. A copy of the input files will be placed in the *condor_schedd* daemon's spool area for the target job, and when the job runs, the output will be staged back into the spool area. Once all of the output has been successfully staged back, it will be copied again, this time from the spool area of the sandboxed job back to the original job's output locations. By default, sandboxing is turned off. Only to turn it on if using a shared X.509 user proxy or if direct staging of remote output files back to the final output locations is not desired.

OverrideRoutingEntry A boolean value that when `True`, indicates that this entry in the routing table replaces any previous entry in the table with the same name. When `False`, it indicates that if there is a previous entry by the same name, the previous entry should be retained and this entry should be ignored. The default value is `True`.

Set_<ATTR> Sets the value of `<ATTR>` in the routed job ClassAd to the specified value. An example of an attribute that might be set is `PeriodicRemove`. For example, if the routed job goes on hold or stays idle for too long, remove it and return the original copy of the job to a normal state.

Eval_Set_<ATTR> Defines an expression written in New ClassAd syntax. The expression is evaluated, and the resulting value sets the value of the routed copy's job ClassAd attribute `<ATTR>`. An expected usage is where a New ClassAd expression evaluation is required. Recall that the *condor-job-router* daemon evaluates using New ClassAd syntax, but the routed copy is represented by the current ClassAd language.

Copy_<ATTR> Defined with the name of a routed copy ClassAd attribute. Copies the value of `<ATTR>` from the original job ClassAd into the specified attribute named of the routed copy. Useful to save the value of an expression, before replacing it with something else that references the original expression.

Delete_<ATTR> Deletes <ATTR> from the routed copy ClassAd. A value assigned to this attribute in the routing table entry is ignored.

5.6.5 Example: constructing the routing table from ReSS

The Open Science Grid has a service called ReSS (Resource Selection Service). It presents grid sites as ClassAds in a Condor collector. This example builds a routing table from the site ClassAds in the ReSS collector.

Using `JOB_ROUTER_ENTRIES_CMD`, we tell the *condor-job-router* daemon to call a simple script which queries the collector and outputs a routing table. The script, called `osg_ress_routing_table.sh`, is just this:

```
#!/bin/sh

# you _MUST_ change this:
export condor_status=/path/to/condor_status
# if no command line arguments specify -pool, use this:
export _CONDOR_COLLECTOR_HOST=osg-ress-1.fnal.gov

$condor_status -format '[ ' BeginAd \
               -format 'GridResource = "gt2 %s"; ' GlueCEInfoContactString \
               -format ']\n' EndAd "$@" | uniq
```

Save this script to a file and make sure the permissions on the file mark it as executable. Test this script by calling it by hand before trying to use it with the *condor-job-router* daemon. You may supply additional arguments such as **-constraint** to limit the sites which are returned.

Once you are satisfied that the routing table constructed by the script is what you want, configure the *condor-job-router* daemon to use it:

```
# command to build the routing table
JOB_ROUTER_ENTRIES_CMD = /path/to/osg_ress_routing_table.sh <extra arguments>

# how often to rebuild the routing table:
JOB_ROUTER_ENTRIES_REFRESH = 3600
```

Using the example configuration, use the above settings to replace `JOB_ROUTER_ENTRIES`. Or, leave `JOB_ROUTER_ENTRIES` there and have a routing table containing entries from both sources. When you restart or reconfigure the *condor-job-router* daemon, you should see messages in the Job Router's log indicating that it is adding more routes to the table.

Platform-Specific Information

The Condor Team strives to make Condor work the same way across all supported platforms. However, because Condor is a very low-level system which interacts closely with the internals of the operating systems on which it runs, this goal is not always possible to achieve. The following sections provide detailed information about using Condor on different computing platforms and operating systems.

6.1 Linux

This section provides information specific to the Linux port of Condor. Linux is a difficult platform to support. It changes very frequently, and Condor has some extremely system-dependent code (for example, the checkpointing library).

Condor is sensitive to changes in the following elements of the system:

- The kernel version
- The version of the GNU C library (glibc)
- the version of GNU C Compiler (GCC) used to build and link Condor jobs (this only matters for Condor's Standard universe which provides checkpointing and remote system calls)

The Condor Team tries to provide support for various releases of the distribution of Linux. Red Hat is probably the most popular Linux distribution, and it provides a common set of versions for the above system components at which Condor can aim support. Condor will often work with Linux distributions other than Red Hat (for example, Debian or SuSE) that have the same versions of the

above components. However, we do not usually test Condor on other Linux distributions and we do not provide any guarantees about this.

New releases of Red Hat usually change the versions of some or all of the above system-level components. A version of Condor that works with one release of Red Hat might not work with newer releases. The following sections describe the details of Condor's support for the currently available versions of Red Hat Linux on x86 architecture machines.

6.1.1 Linux Kernel-specific Information

Distributions that rely on the Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not modify the `atime` of the input device file. This leads to difficulty when Condor is run using one of these kernels. The problem manifests itself in that Condor cannot properly detect keyboard or mouse activity. Therefore, using the activity in policy setting cannot signal that Condor should stop running a job on a machine.

Condor version 6.6.8 implements a workaround for PS/2 devices. A better fix is the Linux 2.6.10 kernel patch linked to from the directions posted at <http://www.cs.wisc.edu/condor/kernel.patch.html>. This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

Condor additionally has problems running on some older Xen kernels, which interact badly with assumptions made by the *condor_procd* daemon. See the FAQ entry in section 7.6 for details.

6.1.2 Red Hat Version 9.x

Red Hat version 9.x is fully supported in Condor Version 7.2.3. *condor_compile* works to link user jobs for the Standard universe with the versions of gcc and glibc that come with Red Hat 9.x.

6.1.3 Red Hat Fedora 1, 2, and 3

Redhat Fedora Core 1, 2, and 3 now support the checkpointing of statically linked executables just like previous revisions of Condor for Red Hat. *condor_compile* works to link user jobs for the Standard universe with the versions of gcc that are distributed with Red Hat Fedora Core 1, 2, and 3.

However, there are some caveats: A) You must install and use the dynamic Red Hat 9.x binaries on the Fedora machine and B) if you wish to do run a *condor_compiled* binary in standalone mode (either initially or in resumption mode), then you must prepend the execution of said binary with **setarch i386**. Here is an example: suppose we have a Condor-linked binary called **myapp**, running this application as a standalone executable will result in this command: **setarch i386 myapp**. The subsequent resumption command will be: **setarch i386 myapp -_condor_restart myapp.ckpt**.

When standard universe executables *condor_compiled* under any currently supported Linux architecture of the same kind (including Fedora 1, 2, and 3) are running inside Condor, they will automatically execute in the i386 execution domain. This means that the **exec_shield** functionality (if available) will be turned off and the shared segment layout will default to Red Hat 9 style. There is no need to do the above instructions concerning **setarch** if the executables are being submitted directly into Condor via *condor_submit*.

6.2 Microsoft Windows

Windows is a strategic platform for Condor, and therefore we have been working toward a complete port to Windows. Our goal is to make Condor every bit as capable on Windows as it is on Unix – or even more capable.

Porting Condor from Unix to Windows is a formidable task, because many components of Condor must interact closely with the underlying operating system. Instead of waiting until all components of Condor are running and stabilized on Windows, we have decided to make a clipped version of Condor for Windows. A clipped version is one in which there is no checkpointing and there are no remote system calls.

This section contains additional information specific to running Condor on Windows. Eventually this information will be integrated into the Condor Manual as a whole, and this section will disappear. In order to effectively use Condor, first read the overview chapter (section 1.1) and the user's manual (section 2.1). If you will also be administrating or customizing the policy and set up of Condor, also read the administrator's manual chapter (section 3.1). After reading these chapters, review the information in this chapter for important information and differences when using and administrating Condor on Windows. For information on installing Condor for Windows, see section 3.2.5.

6.2.1 Limitations under Windows

In general, this release for Windows works the same as the release of Condor for Unix. However, the following items are not supported in this version:

- The standard job universe is not present. This means transparent process checkpoint/migration and remote system calls are not supported.
- For **grid** universe jobs, the only supported grid type is **condor**.
- Accessing files via a network share that requires a Kerberos ticket (such as AFS) is not yet supported.

6.2.2 Supported Features under Windows

Except for those items listed above, most everything works the same way in Condor as it does in the Unix release. This release is based on the Condor Version 7.2.3 source tree, and thus the feature set is the same as Condor Version 7.2.3 for Unix. For instance, all of the following work in Condor:

- The ability to submit, run, and manage queues of jobs running on a cluster of Windows machines.
- All tools such as *condor_q*, *condor_status*, *condor_userprio*, are included. Only *condor_compile* is *not* included.
- The ability to customize job policy using ClassAds. The machine ClassAds contain all the information included in the Unix version, including current load average, RAM and virtual memory sizes, integer and floating-point performance, keyboard/mouse idle time, etc. Likewise, job ClassAds contain a full complement of information, including system dependent entries such as dynamic updates of the job's image size and CPU usage.
- Everything necessary to run a Condor central manager on Windows.
- Security mechanisms.
- Support for SMP machines.
- Condor for Windows can run jobs at a lower operating system priority level. Jobs can be suspended, soft-killed by using a WM_CLOSE message, or hard-killed automatically based upon policy expressions. For example, Condor can automatically suspend a job whenever keyboard/mouse or non-Condor created CPU activity is detected, and continue the job after the machine has been idle for a specified amount of time.
- Condor correctly manages jobs which create multiple processes. For instance, if a Condor job spawns multiple processes and Condor needs to kill the job, all processes created by the job will be terminated.
- In addition to interactive tools, users and administrators can receive information from Condor by e-mail (standard SMTP) and/or by log files.
- Condor includes a friendly GUI installation and set up program, which can perform a full install or deinstall of Condor. Information specified by the user in the set up program is stored in the system registry. The set up program can update a current installation with a new release using a minimal amount of effort.
- Condor can give a job access to the running user's Registry hive.

6.2.3 Secure Password Storage

In order for Condor to operate properly, it must at times be able to act on behalf of users who submit jobs. In particular, this is required on submit machines so that Condor can access a job's

input files, create and access the job's output files, and write to the job's log file from within the appropriate security context. It may also be desirable for Condor to execute the job itself under the security context of its submitting user (see 6.2.4 for details on running jobs as the submitting user on Windows).

On Unix systems, arbitrarily changing what user Condor performs its actions as is easily done when Condor is started with root privileges. On Windows, however, performing an action as a particular user requires knowledge of that user's password, even when running at the maximum privilege level.

Condor on Windows supports the notion of *user privilege switching* through the use of a secure password store. Users can provide Condor with their passwords using the *condor_store_cred* tool. Passwords managed by Condor are encrypted and stored in a secure location within the Windows registry. When Condor needs to perform an action as a particular user, it uses the securely stored password to do so.

By default, the secure password store is managed by the *condor_schedd*. This approach works in environments where the user's password is only needed on the submit machine, which is the case unless the **run_as_owner** capability described in section 6.2.4 is needed.

When the **run_as_owner** feature is needed, it is necessary to configure a centralized *condor_credd* daemon to manage the secure password store. This makes each user's password available, via an encrypted connection to the *condor_credd*, to any execute machine that may need it. The following section contains instructions for setting up a *condor_credd* to manage passwords.

6.2.4 Executing Jobs as the Submitting User

By default, Condor executes jobs on Windows using dedicated "run accounts" that have minimal access rights and privileges. As an alternative, Condor can be configured to allow users to run jobs using their Windows login accounts. This may be useful if jobs need access to files on a network share or to other resources that aren't available to a low-privilege run account.

This feature requires use of a *condor_credd* daemon for secure password storage and retrieval. It is first necessary to select a machine on which to run the *condor_credd*. Often, the machine acting as the pool's central manager is a good choice. An important restriction, however, is that the *condor_credd* host must be a machine running Windows.

All configuration settings necessary to enable the *condor_credd* are contained in the example `etc\condor_config.local_credd` file from the Condor distribution. Copy these settings into a local configuration file for the machine that will run the *condor_credd*. Run `condor_restart` for these new settings to take effect, then verify (via Task Manager) that a *condor_credd* process is running.

A second set of configuration parameters must be enabled for all machines in the pool. The following settings should be used, with the `LOCAL_CREDD` setting customized to point to the machine hosting the *condor_credd* and the `ALLOW_CONFIG` setting customized if needed to refer to an administrative account that exists on all Condor nodes. (A copy of these settings is available in

the comments contained in the `etc\condor_config.local.credd` example file.)

```
CREDD_HOST = credd.cs.wisc.edu
CREDD_CACHE_LOCALLY = True

STARTER_ALLOW_RUNAS_OWNER = True

ALLOW_CONFIG = Administrator@*
SEC_CLIENT_AUTHENTICATION_METHODS = NTSSPI, PASSWORD
SEC_CONFIG_NEGOTIATION = REQUIRED
SEC_CONFIG_AUTHENTICATION = REQUIRED
SEC_CONFIG_ENCRYPTION = REQUIRED
SEC_CONFIG_INTEGRITY = REQUIRED
```

The configuration employed here relies on the `PASSWORD` authentication method to facilitate secure communication between execute machines and the *condor_credd*. In order for `PASSWORD` authenticated communication to work, a “pool password” must be chosen and distributed. Once a pool password is decided upon, it must be stored identically on each machine. The pool password first should be stored on the *condor_credd* host, then the other machines in the pool.

To store the pool password on a given machine, run `condor_store_cred -c add` when logged in with the administrative account on that machine, and enter the password when prompted. If the administrative account is shared across all machines (i.e. if it is a domain account or has the same password on all machines), logging in separately to each machine in the pool can be avoided. Instead, the pool password can be securely pushed out to each machine using commands of the form `condor_store_cred -c -n exec01.cs.wisc.edu add`.

Once the pool password is distributed, execute `condor_reconfig -all` from the central manager. This will cause each execute machine to test its ability to authenticate with the *condor_credd*. To see whether this test worked for each machine in the pool, run the command `condor_status -f "%s\t" Name -f "%s\n" ifThenElse(isUndefined(LocalCredd), \"UNDEF\", LocalCredd)`. Any rows in the output with the “UNDEF” string indicate machines where secure communication is not working properly. Verify that the pool password is stored correctly on these machines.

Once these configuration changes are made and the pool password is distributed, a user that wants a job to run using their own account can simply set the **run_as_owner** macro in the job’s submit file to `True`.

6.2.5 Executing Jobs with the User’s Profile Loaded

Condor can be configured when using dedicated run accounts, to load the account’s profile. A user’s profile includes a set of personal directories and a registry hive loaded under `HKEY_CURRENT_USER`.

This may be useful if the job requires direct access to the user’s registry entries. It also may

be useful when the job requires an application, and the application requires registry access. This feature is always enabled on the *condor_startd*, but it is limited to the dedicated run account. For security reasons, the profiles are removed after the job has completed and exited. This ensures that malicious jobs cannot discover what any previous job has done, nor sabotage the registry for future jobs. It also ensures the next job has a fresh registry hive.

A user that then wants a job to run with a profile uses the **load_profile** command in the job's submit description file:

```
load_profile = True
```

This feature is currently not compatible with **run_as_owner**, and will be ignored if both are specified.

6.2.6 Details on how Condor for Windows starts/stops a job

This section provides some details on how Condor starts and stops jobs. This discussion is geared for the Condor administrator or advanced user who is already familiar with the material in the Administrator's Manual and wishes to know detailed information on what Condor does when starting and stopping jobs.

When Condor is about to start a job, the *condor_startd* on the execute machine spawns a *condor_starter* process. The *condor_starter* then creates:

1. a run account on the machine with a login name of "condor-reuse-slotX", where X is the slot number of the *condor_starter*. This account is added to group Users. This step is skipped if the job is to be run using the submitting user's account (see section 6.2.4).
2. a new temporary working directory for the job on the execute machine. This directory is named "dir_XXX", where XXX is the process ID of the *condor_starter*. The directory is created in the \$(EXECUTE) directory as specified in Condor's configuration file. Condor then grants write permission to this directory for the user account newly created for the job.
3. a new, non-visible Window Station and Desktop for the job. Permissions are set so that only the account that will run the job has access rights to this Desktop. Any windows created by this job are not seen by anyone; the job is run in the background. (Note: Setting `USE_VISIBLE_DESKTOP` to True will allow the job to access the default desktop instead of a newly created one.)

Next, the *condor_starter* (called the starter) contacts the *condor_shadow* (called the shadow) process, which is running on the submitting machine, and pulls over the job's executable and input files. These files are placed into the temporary working directory for the job. After all files have been received, the starter spawns the user's executable. Its current working directory set to the temporary working directory (that is, \$(EXECUTE)/dir_XXX, where XXX is the process id of the *condor_starter* daemon).

While the job is running, the starter closely monitors the CPU usage and image size of all processes started by the job. Every 20 minutes the starter sends this information, along with the total size of all files contained in the job's temporary working directory, to the shadow. The shadow then inserts this information into the job's ClassAd so that policy and scheduling expressions can make use of this dynamic information.

If the job exits of its own accord (that is, the job completes), the starter first terminates any processes started by the job which could still be around if the job did not clean up after itself. The starter examines the job's temporary working directory for any files which have been created or modified and sends these files back to the shadow running on the submit machine. The shadow places these files into the **initialdir** specified in the submit description file; if no **initialdir** was specified, the files go into the directory where the user invoked *condor_submit*. Once all the output files are safely transferred back, the job is removed from the queue. If, however, the *condor_startd* forcibly kills the job before all output files could be transferred, the job is not removed from the queue but instead switches back to the Idle state.

If the *condor_startd* decides to vacate a job prematurely, the starter sends a WM_CLOSE message to the job. If the job spawned multiple child processes, the WM_CLOSE message is only sent to the parent process (that is, the one started by the starter). The WM_CLOSE message is the preferred way to terminate a process on Windows, since this method allows the job to cleanup and free any resources it may have allocated. When the job exits, the starter cleans up any processes left behind. At this point, if **transfer_files** is set to *ONEXIT* (the default) in the job's submit description file, the job switches from states, from Running to Idle, and no files are transferred back. If **transfer_files** is set to *ALWAYS*, then any files in the job's temporary working directory which were changed or modified are first sent back to the submitting machine. But this time, the shadow places these so-called intermediate files into a subdirectory created in the \$(SPOOL) directory on the submitting machine (\$(SPOOL) is specified in Condor's configuration file). The job is then switched back to the Idle state until Condor finds a different machine on which to run. When the job is started again, Condor places into the job's temporary working directory the executable and input files as before, *plus* any files stored in the submit machine's \$(SPOOL) directory for that job.

NOTE: A Windows console process can intercept a WM_CLOSE message via the Win32 SetConsoleCtrlHandler() function if it needs to do special cleanup work at vacate time; a WM_CLOSE message generates a CTRL_CLOSE_EVENT. See SetConsoleCtrlHandler() in the Win32 documentation for more info.

NOTE: The default handler in Windows for a WM_CLOSE message is for the process to exit. Of course, the job could be coded to ignore it and not exit, but eventually the *condor_startd* will become impatient and hard-kill the job (if that is the policy desired by the administrator).

Finally, after the job has left and any files transferred back, the starter deletes the temporary working directory, the temporary account (if one was created), the WindowStation, and the Desktop before exiting. If the starter should terminate abnormally, the *condor_startd* attempts the clean up. If for some reason the *condor_startd* should disappear as well (that is, if the entire machine was power-cycled hard), the *condor_startd* will clean up when Condor is restarted.

6.2.7 Security Considerations in Condor for Windows

On the execute machine (by default), the user job is run using the access token of an account dynamically created by Condor which has bare-bones access rights and privileges. For instance, if your machines are configured so that only Administrators have write access to `C:\WINNT`, then certainly no Condor job run on that machine would be able to write anything there. The only files the job should be able to access on the execute machine are files accessible by the Users and Everyone groups, and files in the job's temporary working directory. Of course, if the job is configured to run using the account of the submitting user (as described in section 6.2.4), it will be able to do anything that the user is able to do on the execute machine it runs on.

On the submit machine, Condor impersonates the submitting user, therefore the File Transfer mechanism has the same access rights as the submitting user. For example, say only Administrators can write to `C:\WINNT` on the submit machine, and a user gives the following to *condor_submit* :

```
executable = mytrojan.exe
initialdir = c:\winnt
output = explorer.exe
queue
```

Unless that user is in group Administrators, Condor will not permit *explorer.exe* to be overwritten.

If for some reason the submitting user's account disappears between the time *condor_submit* was run and when the job runs, Condor is not able to check and see if the now-defunct submitting user has read/write access to a given file. In this case, Condor will ensure that group "Everyone" has read or write access to any file the job subsequently tries to read or write. This is in consideration for some network setups, where the user account only exists for as long as the user is logged in.

Condor also provides protection to the job queue. It would be bad if the integrity of the job queue is compromised, because a malicious user could remove other user's jobs or even change what executable a user's job will run. To guard against this, in Condor's default configuration all connections to the *condor_schedd* (the process which manages the job queue on a given machine) are authenticated using Windows' eSSPI security layer. The user is then authenticated using the same challenge-response protocol that Windows uses to authenticate users to Windows file servers. Once authenticated, the only users allowed to edit job entry in the queue are:

1. the user who originally submitted that job (i.e. Condor allows users to remove or edit their own jobs)
2. users listed in the *condor_config* file parameter `QUEUE_SUPER_USERS`. In the default configuration, only the "SYSTEM" (LocalSystem) account is listed here.

WARNING: Do not remove "SYSTEM" from `QUEUE_SUPER_USERS`, or Condor itself will not be able to access the job queue when needed. If the LocalSystem account on your machine is compromised, you have all sorts of problems!

To protect the actual job queue files themselves, the Condor installation program will automatically set permissions on the entire Condor release directory so that only Administrators have write access.

Finally, Condor has all the IP/Host-based security mechanisms present in the full-blown version of Condor. See section 3.6.9 starting on page 307 for complete information on how to allow/deny access to Condor based upon machine host name or IP address.

6.2.8 Network files and Condor

Condor can work well with a network file server. The recommended approach to having jobs access files on network shares is to configure jobs to run using the security context of the submitting user (see section 6.2.4). If this is done, the job will be able to access resources on the network in the same way as the user can when logged in interactively.

In some environments, running jobs as their submitting users is not a feasible option. This section outlines some possible alternatives. The heart of the difficulty in this case is that on the execute machine, Condor creates a temporary user that will run the job. The file server has never heard of this user before.

Choose one of these methods to make it work:

- METHOD A: access the file server as a different user via a net use command with a login and password
- METHOD B: access the file server as guest
- METHOD C: access the file server with a "NULL" descriptor
- METHOD D: create and have Condor use a special account
- METHOD E: use the contrib module from the folks at Bristol University

All of these methods have advantages and disadvantages.

Here are the methods in more detail:

METHOD A - access the file server as a different user via a net use command with a login and password

Example: you want to copy a file off of a server before running it....

```
@echo off
net use \\myserver\someshare MYPASSWORD /USER:MYLOGIN
copy \\myserver\someshare\my-program.exe
my-program.exe
```

The idea here is to simply authenticate to the file server with a different login than the temporary Condor login. This is easy with the "net use" command as shown above. Of course, the obvious disadvantage is this user's password is stored and transferred as clear text.

METHOD B - access the file server as guest

Example: you want to copy a file off of a server before running it as GUEST

```
@echo off
net use \\myserver\someshare
copy \\myserver\someshare\my-program.exe
my-program.exe
```

In this example, you'd contact the server MYSERVER as the Condor temporary user. However, if you have the GUEST account enabled on MYSERVER, you will be authenticated to the server as user "GUEST". If your file permissions (ACLs) are setup so that either user GUEST (or group EVERYONE) has access the share "someshare" and the directories/files that live there, you can use this method. The downside of this method is you need to enable the GUEST account on your file server. WARNING: This should be done **with extreme caution** and only if your file server is well protected behind a firewall that blocks SMB traffic.

METHOD C - access the file server with a "NULL" descriptor

One more option is to use NULL Security Descriptors. In this way, you can specify which shares are accessible by NULL Descriptor by adding them to your registry. You can then use the batch file wrapper like:

```
net use z: \\myserver\someshare /USER:" "
z:\my-program.exe
```

so long as 'someshare' is in the list of allowed NULL session shares. To edit this list, run regedit.exe and navigate to the key:

```
HKEY_LOCAL_MACHINE\
  SYSTEM\
    CurrentControlSet\
      Services\
        LanmanServer\
          Parameters\
            NullSessionShares
```

and edit it. unfortunately it is a binary value, so you'll then need to type in the hex ASCII codes to spell out your share. each share is separated by a null (0x00) and the last in the list is terminated with two nulls.

although a little more difficult to set up, this method of sharing is a relatively safe way to have one quasi-public share without opening the whole guest account. you can control specifically which shares can be accessed or not via the registry value mentioned above.

METHOD D - create and have Condor use a special account

Create a permanent account (called condor-guest in this description) under which Condor will run jobs. On all Windows machines, and on the file server, create the condor-guest account.

On the network file server, give the condor-guest user permissions to access files needed to run Condor jobs.

Securely store the password of the condor-guest user in the Windows registry using *condor_store_cred* on all Windows machines.

Tell Condor to use the condor-guest user as the owner of jobs, when required. Details for this are in section 3.6.11.

METHOD E - access with the contrib module from Bristol

Another option: some hardcore Condor users at Bristol University developed their own module for starting jobs under Condor NT to access file servers. It involves storing submitting user's passwords on a centralized server. Below I have included the README from this contrib module, which will soon appear on our website within a week or two. If you want it before that, let me know, and I could e-mail it to you.

Here is the README from the Bristol Condor contrib module:

README

Compilation Instructions

Build the projects in the following order

CondorCredSvc

CondorAuthSvc

Crun

Carun

AfsEncrypt

RegisterService

DeleteService

Only the first 3 need to be built in order. This just makes sure that the RPC stubs are correctly rebuilt if required. The last 2 are only helper applications to install/remove the services. All projects are Visual Studio 6 projects. The nmakefiles have been exported for each. Only the project for Carun should need to be modified to change the location of the AFS libraries if needed.

Details

CondorCredSvc

CondorCredSvc is a simple RPC service that serves the domain account credentials. It reads the account name and password from the registry of the machine it's running on. At the moment these details are stored in clear text under the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The account name and password are held in REG_SZ values "Account" and "Password" respectively. In addition there is an optional REG_SZ value "Port" which holds the clear text port number (e.g. "1234"). If this value is not present the service defaults to using port 3654.

At the moment there is no attempt to encrypt the username/password when it is sent over the wire - but this should be reasonably straightforward to change. This service can sit on any machine so keeping the registry entries secure ought to be fine. Certainly the ACL on the key could be set to only allow administrators and SYSTEM access.

CondorAuthSvc and Crun

These two programs do the hard work of getting the job authenticated and running in the right place. CondorAuthSvc actually handles the process creation while Crun deals with getting the winstation/desktop/working directory and grabbing the console output from the job so that Condor's output handling mechanisms still work as advertised. Probably the easiest way to see how the two interact is to run through the job creation process:

The first thing to realize is that condor itself only runs Crun.exe. Crun treats its command line parameters as the program to really run. e.g. "Crun \\mymachine\myshare\myjob.exe" actually causes \\mymachine\myshare\myjob.exe to be executed in the context of the domain account served by CondorCredSvc. This is how it works:

When Crun starts up it gets its window station and desktop - these are the ones created by condor. It also gets its current directory - again already created by condor. It then makes sure that SYSTEM has permission to modify the DACL on the window station, desktop and directory. Next it creates a shared memory section and copies its environment variable block into it. Then, so that it can get hold of STDOUT and STDERR from the job it makes two named pipes on the machine it's running on and attaches a thread to each which just prints out anything that comes in on the pipe to the appropriate stream. These pipes currently have a NULL DACL, but only one instance of each is allowed so there shouldn't be any issues involving malicious people putting garbage into them. The shared memory section and both named pipes are tagged with the ID of Crun's process in case we're on a multi-processor machine that might be running more than one job. Crun then makes an RPC call to CondorAuthSvc to actually start the job, passing

the names of the window station, desktop, executable to run, current directory, pipes and shared memory section (it only attempts to call CondorAuthSvc on the same machine as it is running on). If the job starts successfully it gets the process ID back from the RPC call and then just waits for the new process to finish before closing the pipes and exiting. Technically, it does this by synchronizing on a handle to the process and waiting for it to exit. CondorAuthSvc sets the ACL on the process to allow `EVERYONE` to synchronize on it.

[Technical note: Crun adds `"C:\WINNT\SYSTEM32\CMD.EXE /C"` to the start of the command line. This is because the process is created with the network context of the caller i.e. `LOCALSYSTEM`. Pre-pending `cmd.exe` gets round any unexpected "Access Denied" errors.]

If Crun gets a `WM_CLOSE` (`CTRL_CLOSE_EVENT`) while the job is running it attempts to stop the job, again with an RPC call to CondorAuthSvc passing the job's process ID.

CondorAuthSvc runs as a service under the `LOCALSYSTEM` account and does the work of starting the job. By default it listens on port 3655, but this can be changed by setting the optional `REG_SZ` value "Port" under the registry key

`HKEY_LOCAL_MACHINE\Software\Condor\AuthService`

(Crun also checks this registry key when attempting to contact CondorAuthSvc.) When it gets the RPC to start a job CondorAuthSvc first connects to the pipes for `STDOUT` and `STDERR` to prevent anyone else sending data to them. It also opens the shared memory section with the environment stored by Crun. It then makes an RPC call to CondorCredSvc (to get the name and password of the domain account) which is most likely running on another system. The location information is stored in the registry under the key

`HKEY_LOCAL_MACHINE\Software\Condor\CredService`

The name of the machine running CondorCredSvc must be held in the `REG_SZ` value "Host". This should be the fully qualified domain name of the machine. You can also specify the optional "Port" `REG_SZ` value in case you are running CondorCredSvc on a different port.

Once the domain account credentials have been received the account is logged on through a call to `LogonUser`. The DACLs on the window station, desktop and current directory are then modified to allow the domain account access to them and the job is started in that window station and desktop with a call to `CreateProcessAsUser`. The starting directory is set to the same as sent by Crun, `STDOUT` and `STDERR` handles are set to the named pipes

and the environment sent by Crun is used. CondorAuthSvc also starts a thread which waits on the new process handle until it terminates to close the named pipes. If the process starts correctly the process ID is returned to Crun.

If Crun requests that the job be stopped (again via RPC), CondorAuthSvc loops over all windows on the window station and desktop specified until it finds the one associated with the required process ID. It then sends that window a WM_CLOSE message, so any termination handling built in to the job should work correctly.

[Security Note: CondorAuthSvc currently makes no attempt to verify the origin of the call starting the job. This is, in principal, a bad thing since if the format of the RPC call is known it could let anyone start a job on the machine in the context of the domain user. If sensible security practices have been followed and the ACLs on sensitive system directories (such as C:\WINNT) do not allow write access to anyone other than trusted users the problem should not be too serious.]

Carun and AFSEncrypt

Carun and AFSEncrypt are a couple of utilities to allow jobs to access AFS without any special recompilation. AFSEncrypt encrypts an AFS username/password into a file (called .afs.xxx) using a simple XOR algorithm. It's not a particularly secure way to do it, but it's simple and self-inverse. Carun reads this file and gets an AFS token before running whatever job is on its command line as a child process. It waits on the process handle and a 24 hour timer. If the timer expires first it briefly suspends the primary thread of the child process and attempts to get a new AFS token before restarting the job, the idea being that the job should have uninterrupted access to AFS if it runs for more than 25 hours (the default token lifetime). As a security measure, the AFS credentials are cached by Carun in memory and the .afs.xxx file deleted as soon as the username/password have been read for the first time.

Carun needs the machine to be running either the IBM AFS client or the OpenAFS client to work. It also needs the client libraries if you want to rebuild it.

For example, if you wanted to get a list of your AFS tokens under Condor you would run the following:

```
Crun \\mymachine\myshare\Carun tokens.exe
```

Running a job

To run a job using this mechanism specify the following in your job submission (assuming Crun is in C:\CondorAuth):

```
Executable= c:\CondorAuth\Crun.exe
Arguments = \\mymachine\myshare\carun.exe
\\anothermachine\anothershare\myjob.exe
Transfer_Input_Files = .afs.xxx
```

along with your usual settings.

Installation

A basic installation script for use with the Inno Setup installation package compiler can be found in the Install folder.

6.2.9 Interoperability between Condor for Unix and Condor for Windows

Unix machines and Windows machines running Condor can happily co-exist in the same Condor pool without any problems. Jobs submitted on Windows can run on Windows or Unix, and jobs submitted on Unix can run on Unix or Windows. Without any specification (using the `requirements` expression in the submit description file), the default behavior will be to require the execute machine to be of the same architecture and operating system as the submit machine.

There is absolutely no need to run more than one Condor central manager, even if you have both Unix and Windows machines. The Condor central manager itself can run on either Unix or Windows; there is no advantage to choosing one over the other. Here at University of Wisconsin-Madison, for instance, we have hundreds of Unix (Solaris, Linux, etc) and Windows machines in our Computer Science Department Condor pool. Our central manager is running on Linux. All is happy.

6.2.10 Some differences between Condor for Unix -vs- Condor for Windows

- On Unix, we recommend the creation of a “*condor*” account when installing Condor. On Windows, this is not necessary, as Condor is designed to run as a system service as user `LocalSystem`.
- On Unix, Condor finds the `condor_config` main configuration file by looking in `~condor`, in `/etc`, or via an environment variable. On NT, the location of `condor_config` file is determined via the registry key `HKEY_LOCAL_MACHINE/Software/Condor`. You can override this value by setting an environment variable named `CONDOR_CONFIG`.
- On Unix, in the `VANILLA` universe at job vacate time Condor sends the job a softkill signal defined in the submit-description file (defaults to `SIGTERM`). On NT, Condor sends a `WM_CLOSE` message to the job at vacate time.
- On Unix, if one of the Condor daemons has a fault, a core file will be created in the `$(Log)` directory. On Condor NT, a “core” file will also be created, but instead of a memory dump of the process it will be a very short ASCII text file which describes what fault occurred and where it happened. This information can be used by the Condor developers to fix the problem.

6.3 Macintosh OS X

This section provides information specific to the Macintosh OS X port of Condor. The Macintosh port of Condor is more accurately a port of Condor to Darwin, the BSD core of OS X. Condor uses the Carbon library only to detect keyboard activity, and it does not use Cocoa at all. Condor on the Macintosh is a relatively new port, and it is not yet well-integrated into the Macintosh environment.

Condor on the Macintosh has a few shortcomings:

- Users connected to the Macintosh via *ssh* are not noticed for console activity.
- The memory size of threaded programs is reported incorrectly.
- No Macintosh-based installer is provided.
- The example start up scripts do not follow Macintosh conventions.
- Kerberos is not supported.

Condor does not yet provide Universal binaries for MacOSX. There are separate down loadable packages for both PowerPC (ppc) and Intel (x86) architectures, so please ensure you are using the right Condor binaries for the platform you are trying to run on.

6.4 AIX

This section provides information specific to the AIX ports of Condor.

6.4.1 AIX 5.2L

The version of Condor for AIX 5.2L has the same limitations as Condor for the AIX5.1L platform.

In addition, the Condor binaries for AIX 5.2L will *not* execute on an AIX 5.1L machine.

6.4.2 AIX 5.1L

This is a relatively new port of Condor to the AIX architecture, and as such there are a few things that are not finished. Over time, these will be fixed.

Condor limitations on AIX 5.1L:

- Keyboard Idle and Mouse Idle are wrong.
- The memory size of threaded programs is reported incorrectly.

- The memory and usage statistics of completed jobs is sometimes wrong.
- The standard universe is not supported.
- The LSF and PBS grid-types of the grid universe are not supported.

Note that Condor for the AIX 5.1L machine *will* execute correctly on an AIX 5.2L machine.

Frequently Asked Questions (FAQ)

This is where you can find quick answers to some commonly asked questions about Condor.

7.1 Obtaining & Installing Condor

Where can I download Condor?

Condor can be downloaded from the mirrors listed at <http://www.cs.wisc.edu/condor/downloads>.

When I click to download Condor, it sends me back to the downloads page!

If you are trying to download Condor through a web proxy, try disabling it. Our web site uses the “referring page” as you navigate through our download menus in order to give you the right version of Condor, but sometimes proxies block this information from reaching our web site.

What platforms do you support?

See Section 1.5, on page 5. Also, you might want to read the platform-specific information in Chapter 6 on page 529.

What versions of Red Hat Linux does Condor support?

See Section 6.1 on page 529.

Do you distribute source code?

For 7.0.0 and later releases, the Condor source code is available for public download alongside the binary distributions.

How do I upgrade the Unix machines in my pool from 6.4.x to 6.6.x?

This series of steps explains how to upgrade a pool of machines from running Condor version 6.4.x to version 6.6.x. Read through the entire set of directions before following them.

Briefly, the steps are to download the new version in order to replace your current binaries with the new binaries. Condor will notice that there are new binaries, since it checks for this every few minutes. The next time it checks, the new binaries will be used.

Step 1: (Optional) Place test jobs in queue This optional first step safeguards jobs currently in the queue when you upgrade. By completing this extra step, you will not lose any partially completed jobs, even if something goes wrong with your upgrade.

Manufacture test jobs that utilize each universe you use in your Condor pool. Submit each job, and put the job in the hold state, using *condor_hold*.

Step 2: Place all jobs on hold Place all jobs into the hold state while replacing binaries.

Step 3: Download Condor 6.6.x To ensure that both new and current binaries are within the same volume, make a new directory within your current release directory where 6.6.x will go. Unix commands will be of the form

```
cd <release-dir>
mkdir new
cd new
```

Locate the correct version of the Condor binary, and download into this new directory.

Do *not* install the downloaded version. Do *uncompress* and then *untar* the downloaded version. Further *untar* the release directory (called *release.tar*). This will create the directories

```
bin
etc
include
sbin
```

```
libexec
lib
man
```

From this list of created directories, `bin`, `include`, `sbin`, `libexec`, and `lib` will be used to replace current directories. Note that older versions of Condor do not have a `libexec` directory.

Step 4: Configuration files The downloaded version 6.6.x configuration file will have extra, new suggestions for configuration macro settings, to go with new features in Condor. These extra configuration macros are not be required in order to run version Condor 6.6.x.

Make a backup copy of the current configuration, to safeguard backing out of the upgrade, if something goes wrong.

Work through the new example configuration file to see if there is anything useful and merge with your site-specific (current) configuration file.

Note that starting in Condor 6.6.x, security sessions are turned on by default. If you will be retaining some 6.4.x series Condor installations in your pool, you must turn security sessions off in your 6.6.x configuration files. This can be accomplished by setting

```
SEC_DEFAULT_NEGOTIATION = NEVER
```

Also in 6.6.x, the definition of Hawkeye / Startd Cron jobs has changed. The old syntax allowed the following

```
HAWKEYE_JOBS = \
job1:job1_:/path/to/job1:1h \
job2:job2_:/path/to/job2:5m \
...
```

This is no longer supported, and must be replaced with the following

```
HAWKEYE_JOBS = job1:job1_:/path/to/job1:1h
HAWKEYE_JOBS = $(HAWKEYE_JOBS) job2:job2_:/path/to/job2:5m
HAWKEYE_JOBS = $(HAWKEYE_JOBS) ...
```

It should also be noted that in 6.6.x, the *condor_collector* and *condor_negotiator* can be set to run on non-standard ports. This will cause older (6.4.x and earlier) Condor installations in that pool to no longer function.

Step 5: Replace release directories For each of the directories that is to be replaced, move the current one aside, and put the new one in its place. The Unix commands to do this will be of the form

```
cd <release-dir>

mv bin bin.v64
mv new/bin bin

mv include include.v64
mv new/include include

mv sbin sbin.v64
mv new/sbin sbin

mv lib lib.v64
mv new/lib lib
```

Do this series of directory moves at one sitting, especially avoiding a long time lag between the moves relating to the `sbin` directory. Condor imposes a delay by design, but it does not idly wait for the new binaries to be in place.

Step 6: Observe propagation of new binaries Use `condor_status` to observe the propagation of the upgrade through the pool. As the machines notice and use the new binaries, their version number will change. Complete propagation should occur in five to ten minutes.

The command

```
condor_status -format "%s" Machine -format " %s\n" CondorVersion
```

gives a single line of information about each machine in the pool, containing only the machine name and version of Condor it is running.

Step 7: (Optional) Release test jobs Release the test jobs that were placed into the hold state in Step 1. If these test jobs complete successfully, then the upgrade is successful. If these test jobs fail (possibly by leaving the queue before finishing), then the upgrade is unsuccessful. If unsuccessful, back out of the upgrade by replacing the new configuration file with the backup copy and moving the Version 6.4.x release directories back to their previous location. Also send e-mail to `condor-admin@cs.wisc.edu`, explaining the situation and we'll help you work through it.

Step 8: Release all jobs Release all jobs in the queue, but running `condor_release`.

Step 9: (Optional) Install manual pages The `man` directory was new with Condor version 6.4.x. It contains manual pages. Note that installation of manual pages is optional; the chapter containing manual pages are in section 9.

To install the manual pages, move the `man` directory from `<release-dir>/new` to the desired location. Add the path name to this directory to the `MANPATH`.

What is Personal Condor?

Personal Condor is a term used to describe a specific style of Condor installation suited for individual users who do not have their own pool of machines, but want to submit Condor jobs to run elsewhere.

A Personal Condor is essentially a one-machine, self-contained Condor pool which can use *flocking* to access resources in other Condor pools. See Section 5.2, on page 492 for more information on flocking.

What do I do now? My installation of Condor does not work.

What to do to get Condor running properly depends on what sort of error occurs. One common error category are communication errors. Condor daemon log files report a failure to bind. For example:

```
(date and time) Failed to bind to command ReliSock
```

Or, the errors in the various log files may be of the form:

```
(date and time) Error sending update to collector(s)
(date and time) Can't send end_of_message
(date and time) Error sending UDP update to the collector

(date and time) failed to update central manager

(date and time) Can't send EOM to the collector
```

This problem can also be observed by running *condor_status*. It will give a message of the form:

```
Error: Could not fetch ads --- error communication error
```

To solve this problem, understand that Condor uses the first network interface it sees on the machine. Since machines often have more than one interface, this problem usually implies that the wrong network interface is being used. It also may be the case that the system simply has the wrong IP address configured.

It is incorrect to use the localhost network interface. This has IP address 127.0.0.1 on all machines. To check if this incorrect IP address is being used, look at the contents of the CollectorLog file on the pool's your central manager right after it is started. The contents will be of the form:

```
5/25 15:39:33 *****
5/25 15:39:33 ** condor_collector (CONDOR_COLLECTOR) STARTING UP
5/25 15:39:33 ** $CondorVersion: 6.2.0 Mar 16 2001 $
5/25 15:39:33 ** $CondorPlatform: INTEL-LINUX-GLIBC21 $
5/25 15:39:33 ** PID = 18658
5/25 15:39:33 *****
5/25 15:39:33 DaemonCore: Command Socket at <128.105.101.15:9618>
```

The last line tells the IP address and port the collector has bound to and is listening on. If the IP address is 127.0.0.1, then Condor is definitely using the wrong network interface.

There are two solutions to this problem. One solution changes the order of the network interfaces. The preferred solution sets which network interface Condor should use by adding the following parameter to the local Condor configuration file:

```
NETWORK_INTERFACE = machine-ip-address
```

Where `machine-ip-address` is the IP address of the interface you wish Condor to use.

After an installation of Condor, why do the daemons refuse to start, placing this message in the log files?

```
ERROR "The following configuration macros appear to contain default values
that must be changed before Condor will run.  These macros are:
hostallow_write
(found on line 1853 of /scratch/adesmet/TRUNK/work/src/localdir/condor_config)"
at line 217 in file condor_config.C
```

As of Condor 6.8.0, if Condor sees the bare key word: `YOU_MUST_CHANGE_THIS_INVALID_CONDOR_CONFIGURATION_VALUE` as the value of a configuration file entry, Condor daemons will log the given error message and exit.

By default, an installation of Condor 6.8.0 and later releases will have the configuration file entry `HOSTALLOW_WRITE` set to the above sentinel value. The Condor administrator must alter this value to be the correct domain or IP addresses that the administrator desires. The wild card character (*) may be used to define this entry, but that allows anyone, from anywhere, to submit jobs into your pool. A better value will be of the form `*.domainname.com`.

Why do standard universe jobs never run after an upgrade?

Standard universe jobs that remain in the job queue across an upgrade from any Condor release previous to 6.7.15 to any Condor release of 6.7.15 or more recent cannot run. They are missing a required ClassAd attribute (`LastCheckpointPlatform`) added for all standard universe jobs as of Condor version 6.7.15. This new attribute describes the platform where a job was running when it produced a checkpoint. The attribute is utilized to identify platforms capable of continuing the job (using the checkpoint).

This attribute becomes necessary due to bugs in some Linux kernels. A standard universe job may be continued on some, but not all Linux machines. And, the `CkptOpSys` attribute is not specific enough to be utilized.

There are two possible solutions for these standard universe jobs that cannot run, yet are in the queue:

1. Remove and resubmit the standard universe jobs that remain in the queue across the upgrade. This includes all standard universe jobs that have flocked in to the pool. Note that the resubmitted jobs will start over again from the beginning.

2. For each standard universe job in the queue, modify its job ClassAd such that it can possibly run within the upgraded pool. If the job has already run and produced a checkpoint on a machine before the upgrade, determine the machine that produced the checkpoint using the `LastRemoteHost` attribute in the job's ClassAd. Then look at that machine's ClassAd (after the upgrade) to determine and extract the value of the `CheckpointPlatform` attribute. Add this (using *condor_qedit*) as the value of the new attribute `LastCheckpointPlatform` in the job's ClassAd. Note that this operation must also have to be performed on standard universe jobs flocking in to an upgraded pool. It is recommended that pools that flock between each other upgrade to a post 6.7.15 version of Condor.

Note that if the upgrade to Condor takes place at the same time as a platform change (such as booting an upgraded kernel), there is no way to properly set the `LastCheckpointPlatform` attribute. The only option is to remove and resubmit the standard universe jobs.

7.2 Setting up Condor

How do I set up a central manager on a machine with multiple network interfaces?

Please see section 3.7.2 on page 329.

How do I get more than one job to run on my SMP machine?

Condor will automatically recognize a SMP machine and advertise each CPU of the machine separately. For more details, see section 3.12.7 on page 399.

How do I configure a separate policy for the CPUs of an SMP machine?

Please see section 3.12.7 on page 399 for a lengthy discussion on this topic.

How do I set up my machines so that only specific users' jobs will run on them?

Restrictions on what jobs will run on a given resource are enforced by only starting jobs that meet specific constraints, and these constraints are specified as part of the configuration.

To specify that a given machine should only run certain users' jobs, and always run the jobs regardless of other activity on the machine, load average, etc., place the following entry in the machine's Condor configuration file:

```
START = ( (RemoteUser == "userfoo@baz.edu") || \
          (RemoteUser == "userbar@baz.edu") )
```

A more likely scenario is that the machine is restricted to run only specific users' jobs, contingent on the machine not having other interactive activity and not being heavily loaded. The following entries are in the machine's Condor configuration file. Note that extra configuration variables are defined to make the `START` variable easier to read.

```
# Only start jobs if:
# 1) the job is owned by the allowed users, AND
# 2) the keyboard has been idle long enough, AND
# 3) the load average is low enough OR the machine is currently
#    running a Condor job, and would therefore accept running
#    a different one
AllowedUser    = ( (RemoteUser == "userfoo@baz.edu") || \
                  (RemoteUser == "userbar@baz.edu") )
KeyboardUnused = (KeyboardIdle > $(StartIdleTime))
NoOwnerLoad    = ($(CPUIidle) || (State != "Unclaimed" && State != "Owner"))
START          = $(AllowedUser) && $(KeyboardUnused) && $(NoOwnerLoad)
```

To configure multiple machines to do so, create a common configuration file containing this entry for them to share.

How do I configure Condor to run my jobs only on machines that have the right packages installed?

This is a two-step process. First, you need to tell the machines to report that they have special software installed, and second, you need to tell the jobs to require machines that have that software.

To tell the machines to report the presence of special software, first add a parameter to their configuration files like so:

```
HAS_MY_SOFTWARE = True
```

And then, if there are already `STARTD_ATTRS` defined in that file, add `HAS_MY_SOFTWARE` to them, or, if not, add the line:

```
STARTD_ATTRS = HAS_MY_SOFTWARE, $(STARTD_ATTRS)
```

NOTE: For these changes to take effect, each *condor_startd* you update needs to be reconfigured with *condor_reconfig -startd*.

Next, to tell your jobs to only run on machines that have this software, add a requirements statement to their submit files like so:

```
Requirements = (HAS_MY_SOFTWARE =?= True)
```

NOTE: Be sure to use `==?` instead of `==` so that if a machine doesn't have the `HAS_MY_SOFTWARE` parameter defined, the job's Requirements expression will not evaluate to "undefined", preventing it from running anywhere!

How do I configure Condor to only run jobs at night?

A commonly requested policy for running batch jobs is to only allow them to run at night, or at other pre-specified times of the day. Condor allows you to configure this policy with the use of the `ClockMin` and `ClockDay` *condor_startd* attributes. A complete example of how to use these attributes for this kind of policy is discussed in subsection 3.5.9 on page 276.

How do I configure Condor such that all machines do not produce checkpoints at the same time?

If machines are configured to produce checkpoints at fixed intervals, a large number of jobs are queued (submitted) at the same time, and these jobs start on machines at about the same time, then all these jobs will be trying to write out their checkpoints at the same time. It is likely to cause rather poor performance during this burst of writing.

The `RANDOM_INTEGER()` macro can help in this instance. Instead of defining `PERIODIC_CHECKPOINT` to be a fixed interval, each machine is configured to randomly choose one of a set of intervals. For example, to set a machine's interval for producing checkpoints to within the range of two to three hours, use the following configuration:

```
PERIODIC_CHECKPOINT = $(LastCkpt) > ( 2 * $(HOUR) + \
    $RANDOM_INTEGER(0,60,10) * $(MINUTE) )
```

The interval used is set at configuration time. Each machine is randomly assigned a different interval (2 hours, 2 hours and 10 minutes, 2 hours and 20 minutes, etc.) at which to produce checkpoints. Therefore, the various machines will not all attempt to produce checkpoints at the same time.

Why will the *condor_master* not run when a local configuration file is missing?

If a `LOCAL_CONFIG_FILE` is specified in the global configuration file, but the specified file does not exist, the *condor_master* will not start up, and it prints a variation of the following example message.

```
ERROR: Can't read config file /mnt/condor/hosts/bagel/condor_config.local
```

This is not a bug; it is a feature! Condor has always worked this way on purpose. There is a potentially large security hole if Condor is configured to read from a file that does not exist. By creating that file, a malicious user could change all sorts of Condor settings. This is an easy way to gain root access to a machine, where the daemons are running as root.

The intent is that if you've set up your global configuration file to read from a local configuration file, and the local file is not there, then something is wrong. It is better for the *condor_master* to exit right away and log an error message than to start up.

If the *condor_master* continued with the local configuration file missing, either A) someone could breach security or B) you will have potentially important configuration information missing. Consider the example where the local configuration file was on an NFS partition and the server was down. There would be all sorts of really important stuff in the local configuration file, and Condor might do bad things if it started without those settings.

If supplied it with an empty file, the *condor_master* works fine.

7.3 Running Condor Jobs

I'm at the University of Wisconsin-Madison Computer Science Dept., and I am having problems!

Please see the web page <http://www.cs.wisc.edu/condor/uwcs>. As it explains, your home directory is in AFS, which by default has access control restrictions which can prevent Condor jobs from running properly. The above URL will explain how to solve the problem.

I'm getting a lot of e-mail from Condor. Can I just delete it all?

Generally you shouldn't ignore **all** of the mail Condor sends, but you can reduce the amount you get by telling Condor that you don't want to be notified every time a job successfully completes, only when a job experiences an error. To do this, include a line in your submit file like the following:

```
Notification = Error
```

See the Notification parameter in the *condor_q* man page on page 799 of this manual for more information.

Why will my vanilla jobs only run on the machine where I submitted them from?

Check the following:

1. Did you submit the job from a local file system that other computers can't access?
See Section 3.3.7, on page 168.
2. Did you set a special requirements expression for vanilla jobs that's preventing them from running but not other jobs?
See Section 3.3.7, on page 168.
3. Is Condor running as a non-root user?
See Section 3.6.11, on page 316.

My job starts but exits right away with signal 9.

This can occur when the machine your job is running on is missing a shared library required by your program. One solution is to install the shared library on all machines the job may execute on. Another, easier, solution is to try to re-link your program statically so it contains all the routines it needs.

Why aren't any or all of my jobs running?

Problems like the following are often reported to us:

```
I have submitted 100 jobs to my pool, and only 18 appear to be
running, but there are plenty of machines available.  What should I
do to investigate the reason why this happens?
```

Start by following these steps to understand the problem:

1. Run `condor_q -analyze` and see what it says.
2. Look at the User Log file (whatever you specified as "log = XXX" in the submit file).
See if the jobs are starting to run but then exiting right away, or if they never even start.
3. Look at the SchedLog on the submit machine after it negotiates for this user. If a user doesn't have enough priority to get more machines the SchedLog will contain a message like "lost priority, no more jobs".
4. If jobs are successfully being matched with machines, they still might be dying when they try to execute due to file permission problems or the like. Check the ShadowLog on the submit machine for warnings or errors.
5. Look at the NegotiatorLog during the negotiation for the user. Look for messages about priority, "no more machines", or similar.

Another problem shows itself with statements within the log file produced by the *condor_schedd* daemon (given by `$(SCHEDD_LOG)`) that say the following:

```
2/3 17:46:53 Swap space estimate reached! No more jobs can be run!
12/3 17:46:53 Solution: get more swap space, or set RESERVED_SWAP = 0
12/3 17:46:53 0 jobs matched, 1 jobs idle
```

Condor computes the total swap space on your submit machine. It then tries to limit the total number of jobs it will spawn based on an estimate of the size of the *condor_shadow* daemon's memory footprint and a configurable amount of swap space that should be reserved. This is done to avoid the situation within a very large pool in which all the jobs are submitted from a single host. The huge number of *condor_shadow* processes would overwhelm the submit machine, it would run out of swap space, and thrash.

Things can go wrong if a machine has a lot of physical memory and little or no swap space. Condor does not consider the physical memory size, so the situation occurs where Condor thinks it has no swap space to work with, and it will not run the submitted jobs.

To see how much swap space Condor thinks a given machine has, use the output of a *condor_status* command of the following form:

```
condor_status -schedd [hostname] -long | grep VirtualMemory
```

If the value listed is 0, then this is what is confusing Condor. There are two ways to fix the problem:

1. Configure your machine with some real swap space.
2. Disable this check within Condor. Define the amount of reserved swap space for the submit machine to 0. Set `RESERVED_SWAP` to 0 in the configuration file:

```
RESERVED_SWAP = 0
```

and then send a *condor_restart* to the submit machine.

Why does the `requirementsexpression` for the job I submitted have extra things that I did not put in my submit description file?

There are several extensions to the submitted `requirements` that are automatically added by Condor. Here is a list:

- Condor automatically adds `arch` and `opsys` if not specified in the submit description file. It is assumed that the executable needs to execute on the same platform as the machine on which the job is submitted.

- Condor automatically adds the expression `(Memory * 1024 > ImageSize)`. This ensures that the job will run on a machine with at least as much physical memory as the memory footprint of the job.
- Condor automatically adds the expression `(Disk >= DiskUsage)` if not already specified. This ensures that the job will run on a machine with enough disk space for the job's local I/O (if there is any).
- A pool administrator may define configuration variables that cause expressions to be added to `requirements`. These configuration variables are `APPEND_REQUIREMENTS`, `APPEND_REQ_VANILLA`, and `APPEND_REQ_STANDARD`. These configuration variables give pool administrators the flexibility to set policy for a local pool.
- Older versions of Condor needed to add confusing clauses about `WINNT` and the `FileSystemDomain` to vanilla universe jobs. This made sure that the jobs ran on a machine where files were accessible. The Windows version supported automatically transferring files with the vanilla job, while the Unix version relied on a shared file system. Since the Unix version of Condor now supports transferring files, these expressions are no longer added to the `requirements` for a job.

When I use *condor_compile* to produce a job, I get an error that says, "Internal ld was not invoked!". What does this mean?

condor_compile enforces a specific behavior in the compilers and linkers that it supports (for example *gcc*, *g77*, *cc*, *CC*, *ld*) where a special linker script provided by Condor must be invoked during the final linking stages of the supplied compiler or linker.

In some rare cases, as with *gcc* compiled with the options `-with-as` or `-with-ld`, the enforcement mechanism we rely upon to have *gcc* choose our supplied linker script is not honored by the compiler. When this happens, an executable is produced, but the executable is devoid of the Condor libraries which both identify it as a Condor executable linked for the standard universe and implement the feature sets of remote I/O and transparent process checkpointing and migration.

Often, the only fix in order to use the compiler desired, is to reconfigure and recompile the compiler itself, such that it does not use the errant options mentioned.

With Condor's standard universe, we highly recommend that your source files are compiled with the supported compiler for your platform. See section 1.5 for the list of supported compilers. For a Linux platform, the supported compiler is the default compiler that came with the distribution. It is often found in the directory `/usr/bin`.

Can I submit my standard universe SPARC Solaris 2.6 jobs and have them run on a SPARC Solaris 2.7 machine?

No. You may only use binary compatibility between SPARC Solaris 2.5.1 and SPARC Solaris 2.6 and between SPARC Solaris 2.7 and SPARC Solaris 2.8, but not between SPARC Solaris 2.6 and

SPARC Solaris 2.7. We may implement support for this feature in a future release of Condor.

Can I submit my standard universe SPARC Solaris 2.8 jobs and have them run on a SPARC Solaris 2.9 machine?

No. Although normal executables are binary compatible, technical details of taking checkpoints currently prevents this particular combination. Note that this applies to standard universe jobs only.

Why have standard universe jobs in Condor 6.6.x have begun unexpectedly segmentation faulting during a checkpoint after an upgrade of Redhat Enterprise Linux 3 to current update levels?

Redhat has apparently back-ported a 2.6 kernel feature called “exec_shield” to the current patch levels of the RHEL3 product line. This feature is designed to make buffer overflow attacks incredibly difficult to exploit. However, it has the unfortunate side effect of completely breaking all user land checkpointing algorithms including the one Condor utilizes. The solution is to turn off the kernel feature for each execution of a standard universe job in the Condor system. The method employed to do this is with `USER_JOB_WRAPPER` and a shell script that looks much like this one:

```
#!/bin/sh

sa="/usr/bin/setarch"

if [ -f $sa ]; then
    exec $sa i386 ${1+"$@"}
fi

exec ${1+"$@"}
```

Place this shell script into the `$(SBIN)` directory of your Condor installation with the name of `fix_std_univ` and make sure to `chmod 755 fix_std_univ` it. Then, set `USER_JOB_WRAPPER = $(SBIN)/fix_std_univ` in your global config file (or the config files which will affect your Linux install of Condor). Then do a `condor_reconfig` of your pool. When a standard universe job is run on a machine, if the `setarch` program is available (under Linux with the “exec_shield” feature), then it will run the executable in the i386 personality, which turns off the “exec_shield” kernel feature.

Why do my vanilla jobs keep cycling between suspended and unsuspended?

Condor tries to provide a number, the “Condor Load Average” (reported in the machine ClassAd as `CondorLoadAvg`), which is intended to represent the total load average on the system caused

by any running Condor job(s). Unfortunately, it is impossible to get an accurate number for this without support from the operating system. This is not available. So, Condor does the best it can, and it mostly works in most cases. However, there are a number of ways this statistic can go wrong.

The old default Condor policy was to suspend if the non-Condor load average went over a certain threshold. However, because of the problems providing accurate numbers for this (described below), some jobs would go into a cycle of getting suspended and resumed. The default suspend policy now shipped with Condor uses the solution explained here.

While there are too many technical details of why `CondorLoadAvg` might be wrong for a short answer here, a brief explanation is presented. When a job has periodic behavior, and the load it places upon a machine is changing over time, the system load also changes over time. However, Condor thinks that the job's share of the system load (what it uses to compute the `CondorLoad`) is also changing. So, when the job was running, and then stops, both the system load and the Condor load start falling. If it all worked correctly, they'd fall at the exact same rate, and `NonCondorLoad` would be constant. Unfortunately, `CondorLoadAvg` falls faster, since Condor thinks the job's share of the total load is falling, too. Therefore, `CondorLoadAvg` falls faster than the system load, `NonCondorLoad` goes up, and the old default `SUSPEND` expression becomes true.

It appears that Condor should be able to avoid this problem, but for a host of reasons, it can not. There is no good way (without help from the operating systems Condor runs on; the help does not exist) to get this right. The only way to compute these numbers more accurately without support from the operating system is to sample everything at such a high rate that Condor itself would create a large load average, just to try to compute the load average. This is Heisenberg's uncertainty principle in action.

A similar sampling error can occur when Condor is starting a job within the vanilla universe with many processes and with a heavy initial load. Condor mistakenly decides that the load on the machine has gotten too high while the job is in the initialization phase and kicks the job off the machine.

To correct this problem, Condor needs to check to see if the load of the machine has been high over an interval of time. There is an attribute, `CpuBusyTime` that can be used for this purpose. This macro returns the time `$(CpuBusy)` (defined in the default configuration file) has been true, or 0 if `$(CpuBusy)` is false. `$(CpuBusy)` is usually defined in terms of non-Condor load. These are the default settings:

```
NonCondorLoadAvg = (LoadAvg - CondorLoadAvg)
HighLoad         = 0.5
CPUBusy          = ($(NonCondorLoadAvg) >= $(HighLoad))
```

To take advantage of `CpuBusyTime`, you can use it in your `SUSPEND` expression.

Here is an example:

```
SUSPEND = (CpuBusyTime > 3 * $(MINUTE)) && ((CurrentTime - JobStart) > 90)
```

The above policy says to only suspend the job if the CPU has been busy with non-Condor load at least three minutes and it has been at least 90 seconds since the start of the job.

Why might my job be preempted (evicted)?

There are four circumstances under which Condor may evict a job. They are controlled by different expressions.

Reason number 1 is the user priority: controlled by the `PREEMPTION_REQUIREMENTS` expression in the configuration file. If there is a job from a higher priority user sitting idle, the *condor_negotiator* daemon may evict a currently running job submitted from a lower priority user if `PREEMPTION_REQUIREMENTS` is True. For more on user priorities, see section 2.7 and section 3.4.

Reason number 2 is the owner (machine) policy: controlled by the `PREEMPT` expression in the configuration file. When a job is running and the `PREEMPT` expression evaluates to True, the *condor_startd* will evict the job. The `PREEMPT` expression should reflect the requirements under which the machine owner will not permit a job to continue to run. For example, a policy to evict a currently running job when a key is hit or when it is the 9:00am work arrival time, would be expressed in the `PREEMPT` expression and enforced by the *condor_startd*. For more on the `PREEMPT` expression, see section 3.5.

Reason number 3 is the owner (machine) preference: controlled by the `RANK` expression in the configuration file (sometimes called the *startd* rank or machine rank). The `RANK` expression is evaluated as a floating point number. When one job is running, a second idle job that evaluates to a higher `RANK` value tells the *condor_startd* to prefer the second job over the first. Therefore, the *condor_startd* will evict the first job so that it can start running the second (preferred) job. For more on `RANK`, see section 3.5.

Reason number 4 is if Condor is to be shutdown: on a machine that is currently running a job. Condor evicts the currently running job before proceeding with the shutdown.

Condor does not stop the Condor jobs running on my Linux machine when I use my keyboard and mouse. Is there a bug?

There is no bug in Condor. Unfortunately, recent Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not post proper state information, such that Condor can detect keyboard and mouse activity. Condor implements workarounds to piece together the needed state information for PS/2 devices. A better fix of the problem utilizes the kernel patch linked to from the directions posted at <http://www.cs.wisc.edu/condor/kernel.patch.html>. This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

What signals get sent to my jobs when Condor needs to preempt or kill them, or when I remove them from the queue? Can I tell Condor which signals to send?

The answer is dependent on the universe of the jobs.

Under the scheduler universe, the signal jobs get upon *condor_rm* can be set by the user in the submit description file with the form of

```
remove_kill_sig = SIGWHATEVER
```

If this command is not defined, Condor further looks for a command in the submit description file with the form

```
kill_sig = SIGWHATEVER
```

And, if that command is also not given, Condor uses SIGTERM.

For all other universes, the jobs get the value of the submit description file command *kill_sig*, which is SIGTERM by default.

If a job is killed or evicted, the job is sent a *kill_sig*, unless it is on the receiving end of a hard kill, in which case it gets SIGKILL.

Under all universes, the signal is sent only to the parent PID of the job, namely, the first child of the *condor_starter*. If the child itself is forking, the child must catch and forward signals as appropriate. This in turn depends on the user's desired behavior. The exception to this is (again) where the job is receiving a hard kill. Condor sends the value SIGKILL to all the PIDs in the family.

Why does my Linux job have an enormous ImageSize and refuse to run anymore?

Sometimes Linux jobs run, are preempted and can not start again because Condor thinks the image size of the job is too big. This is because Condor has a problem calculating the image size of a program on Linux that uses threads. It is particularly noticeable in the Java universe, but it also happens in the vanilla universe. It is not an issue in the standard universe, because threaded programs are not allowed.

On Linux, each thread appears to consume as much memory as the entire program consumes, so the image size appears to be (number-of-threads * image-size-of-program). If your program uses a lot of threads, your apparent image size balloons. You can see the image size that Condor believes your program has by using the *-l* option to *condor_q*, and looking at the ImageSize attribute.

When you submit your job, Condor creates or extends the requirements for your job. In particular, it adds a requirement that your job must run on a machine with sufficient memory:

```
Requirements = ... ((Memory * 1024) >= ImageSize) ...
```

(Note that memory is the execution machine's memory in megabytes while ImageSize is in kilobytes). When your application is threaded, the image size appears to be much larger than it really is, and you may not have a machine with sufficient memory to handle this requirement.

Unfortunately, calculating the correct ImageSize is rather hard to fix on Linux, and we do not yet have a good solution. Fortunately, there is a workaround while we work on a good solution for a future release.

In the Requirements expression above, Condor added (Memory * 1024) >= ImageSize) on your behalf. You can prevent Condor from doing this by giving it your own expression about memory in your submit file, just as:

```
Requirements = Memory > 1024
```

You will need to change 1024 to a reasonably good estimate of the actual image size of your program, in kilobytes. This expression says that your program requires 1 megabyte of memory. If you underestimate the memory your application needs, you may have bad performance if you job runs on machines that have insufficient memory.

In addition, if you have modified your machine policies to preempt jobs when they get big a ImageSize, you will need to change those policies.

Why does the time output from *condor_status* appear as [????] ?

Condor collects timing information for a large variety of uses. Collection of the data relies on accurate times. Being a distributed system, clock skew among machines causes errant timing calculations. Values can be reported too large or too small, with the possibility of calculating negative timing values.

This problem may be seen by the user when looking at the output of *condor_status*. If the **ActivityTime** field appears as [????], then this calculated statistic was negative. *condor_status* recognizes that a negative amount of time will be nonsense to report, and instead displays this string.

The solution to the problem is to synchronize the clocks on these machines. An administrator can do this using a tool such as *ntp*.

The user condor's home directory cannot be found. Why?

This problem may be observed after installation, when attempting to execute

```
~condor/condor/bin/condor_config_val -tilde
```

and there is a user named condor. The command prints a message such as

```
Error: Specified -tilde but can't find condor's home directory
```

In this case, the difficulty stems from using NIS, because the Condor daemons fail to communicate properly with NIS to get account information. To fix the problem, a dynamically linked version of Condor must be installed.

Condor commands (including *condor_q*) are really slow. What is going on?

Some Condor programs will react slowly if they expect to find a *condor_collector* daemon, yet cannot contact one. Notably, *condor_q* can be very slow. The *condor_schedd* daemon will also be slow, and it will log lots of harmless messages complaining. If you are not running a *condor_collector* daemon, it is important that the configuration variable `COLLECTOR_HOST` be set to nothing. This is typically done by setting `CONDOR_HOST` with

```
CONDOR_HOST=
COLLECTOR_HOST=$(CONDOR_HOST)
```

or

```
COLLECTOR_HOST=
```

Where are my missing files? The command `when_to_transfer_output = ON_EXIT_OR_EVICT` is in the submit description file.

Although it may appear as if files are missing, they are not. The transfer does take place whenever a job is preempted by another job, vacates the machine, or is killed. Look for the files in the directory defined by the `SPOOL` configuration variable. See section 2.5.4, on page 26 for details on the naming of the intermediate files.

Why are my vm universe VMware jobs failing and being put on hold?

Strange behavior has been noted when Condor tries to run a **vm** universe VMware job using a path to a VMX file that contains a symbolic link. An example of an error message that may appear in such a job's user log:

```
Error from starter on master_vmuniverse_strtd@nostos.cs.wisc.edu: register(/scratch/gquinn/condor/git/CONDOR_SRC/src/condor_tests/31426/31426vmuniverse/execute/dir_31534/vmN3hylp_condor.vmx) = 1/Error: Command failed: A file was not found/(ERROR) Can't create snapshot for vm(/scratch/gquinn/condor/git/CONDOR_SRC/src/condor_tests/31426/31426vmuniverse/execute/dir_31534/vmN3hylp_condor.vmx)
```

To work around this problem:

- If using file transfer (the submit description file contains **vmware_should_transfer_files = true**), then modify any configuration variable **EXECUTE** values on all execute machines, such that they do not contain symbolic link path components.
- If using a shared file system, ensure that the submit description file command **vmware_dir** does not use symbolic link path name components.

7.4 Condor on Windows

Will Condor work on a network of mixed Unix and Windows machines?

You can have a Condor pool that consists of both Unix and Windows machines.

Your central manager can be either Windows or Unix. For example, even if you had a pool consisting strictly of Unix machines, you could use a Windows box for your central manager, and vice versa.

Submitted jobs can originate from either a Windows **or** a Unix machine, and be destined to run on Windows **or** a Unix machine. Note that there are still restrictions on the supported universes for jobs executed on Windows machines.

So, in summary:

1. A single Condor pool can consist of both Windows and Unix machines.
2. It does not matter at all if your Central Manager is Unix or Windows.
3. Unix machines can submit jobs to run on other Unix or Windows machines.
4. Windows NT machines can submit jobs to run on other Windows or Unix machines.

What versions of Windows will Condor run on?

See Section 1.5, on page 5.

My Windows program works fine when executed on its own, but it does not work when submitted to Condor.

First, make sure that the program really does work outside of Condor under Windows, that the disk is not full, and that the system is not out of user resources.

As the next consideration, know that some Windows programs do not run properly because they are dynamically linked, and they cannot find the .dll files that they depend on. Version 6.4.x of Condor sets the PATH to be empty when running a job. To avoid these difficulties, do one of the following

1. statically link the application
2. wrap the job in a script that sets up the environment
3. submit the job from a correctly-set environment with the command

```
getenv = true
```

in the submit description file. This will copy your environment into the job's environment.

4. send the required .dll files along with the job using the submit description file command **transfer_input_files**.

Why is the *condor_master* daemon failing to start, giving an error about "In StartServiceCtrlDispatcher, Error number: 1063"?

In Condor for Windows, the *condor_master* daemon is started as a service. Therefore, starting the *condor_master* daemon as you would on Unix will not work. Start Condor on Windows machines using either

```
net start condor
```

or start the Condor service from the Service Control Manager located in the Windows Control Panel.

Jobs submitted from Windows give an error referring to a credential.

Jobs submitted from a Windows machine require a stashed password in order for Condor to perform certain operations on the user's behalf. Refer to section 6.2.3 for information about password storage on Windows. The command which stashes a password for a user is *condor_store_cred*. See the manual page on page 793 for usage details.

The error message that Condor gives if a user has not stashed a password is of the form:

```
ERROR: No credential stored for username@machinename
```

```
Correct this by running:  
condor_store_cred add
```

Jobs submitted from Unix to execute on Windows do not work properly.

A difficulty with defaults causes jobs submitted from Unix for execution on a Windows platform to remain in the queue, but make no progress. For jobs with this problem, log files will contain error messages pointing to shadow exceptions.

This difficulty stems from the defaults for whether file transfer takes place. The workaround for this problem is to place the line

```
TRANSFER_FILES = ALWAYS
```

into the submit description file for jobs submitted from a Unix machine for execution on a Windows machine.

When I run *condor_status* I get a communication error, or the Condor daemon log files report a failure to bind.

Condor uses the first network interface it sees on your machine. This problem usually means you have an extra, inactive network interface (such as a RAS dial up interface) defined before to your regular network interface.

To solve this problem, either change the order of your network interfaces in the Control Panel, or explicitly set which network interface Condor should use by adding the following parameter to your Condor configuration file:

```
NETWORK_INTERFACE = ip-address
```

Where *ip-address* is the IP address of the interface you wish Condor to use.

My job starts but exits right away with status 128.

This can occur when the machine your job is running on is missing a DLL (Dynamically Linked Library) required by your program. The solution is to find the DLL file the program needs and put it in the `TRANSFER_INPUT_FILES` list in the job's submit file.

To find out what DLLs your program depends on, right-click the program in Explorer, choose Quickview, and look under "Import List".

How can I access network files with Condor on Windows?

Five methods for making access of network files work with Condor are given in section 6.2.8.

What is wrong when *condor_off* cannot find my host, and *condor_status* does not give me a complete host name?

Given the command

```
condor_off hostname2
```

an error message of the form

```
Can't find address for master hostname2.somewhere.edu
```

appears. Yet, when looking at the host names with

```
condor_status -master
```

the output is of the form

```
hostname1.somewhere.edu
hostname2
hostname3.somewhere.edu
```

To correct this incomplete host name, add an entry to the configuration file for `DEFAULT_DOMAIN_NAME` that specifies the domain name to be used. For the example given, the configuration entry will be

```
DEFAULT_DOMAIN_NAME = somewhere.edu
```

After adding this configuration file entry, use *condor_restart* to restart the Condor daemons and effect the change.

Does `USER_JOB_WRAPPER` work on Windows machines?

The `USER_JOB_WRAPPER` configuration variable does work on Windows machines. The wrapper must be either a batch script with a file extension of `.bat` or `.cmd`, or an executable with a file extension of `.exe` or `.com`.

An example of a batch script sets environment variables:

```
REM set some environment variables
set LICENSE_SERVER=192.168.1.202:5012
set MY_PARAMS=2

REM Run the actual job now
%*
```

***condor_store_cred* is failing, and I'm sure I'm typing my password correctly.**

First, make sure the *condor_schedd* is running.

Next, check the SchedLog. It will contain more detailed information about the failure. Frequently, the error is a result of PERMISSION DENIED errors. You can read more about properly configuring security settings on page 307.

My submit machine cannot have more than 120 jobs running concurrently. Why?

Windows is likely to be running out of desktop heap. Confirm this to be the case by looking in the log for the *condor_schedd* daemon to see if *condor_shadow* daemons are immediately exiting with status 128. If this is the case, increase the desktop heap size. Open the registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SubSystems\Windows
```

The SharedSection value can have three values separated by commas. The third value controls the desktop heap size for non-interactive desktops, which the Condor service uses. The default is 512 (Kbytes). 60 *condor_shadow* daemons consume about 256 Kbytes, hence 120 shadows can run with the default value. To be able to run a maximum of 300 *condor_shadow* daemons, set this value at 1280.

Reboot the system for the changes to take effect. For more information, see Microsoft Article Q184802.

Why do Condor daemons exit after logging a 10038 (WSAENOTSOCK) error on some machines?

Usually when Condor daemons exit in this manner, it is because the system in question has a non-standard Winsock Layered Service Provider (LSP) installed on it. An LSP is, in effect, a plug-in for the TCP/IP protocol stack. LSPs have been installed as part of anti-virus software and other security-related packages.

There are several tools available to check your system for the presence of LSPs. One with which we have had success is *LSP-Fix*, available at <http://www.cexx.org/lspfix.htm>. Any non-Microsoft LSPs identified by this tool may potentially be causing the WSAENOTSOCK error in Condor. Although the *LSP-Fix* tool allows the direct removal of an LSP, it is likely advisable to completely remove the application for which the LSP is a part via the Control Panel.

Another approach is to completely reset the TCP/IP stack to its original state. This can be done using the *netsh* tool:

```
netsh int ip reset reset-stack.log
```

The command will return the TCP/IP stack back to the state it was in when the OS was first installed. The log file defined above will record all the configuration changes made by netsh.

Why do Condor daemons exit with "Unexpected performance counter size", "unable to spawn the ProcD" or "loadavg thread died, restarting. (exit code=2)" errors?

Condor relies on some of Windows' built-in performance counters for its operation. It is possible (via the registry) for performance counters that Condor requires to be disabled. If this is the case, you may see daemons exit with a message like:

```
1/26 09:16:42 (fd:2) (pid:5732) ERROR: "Unexpected performance counter
      size for total CPU: 0 (expected: 8)" at line 2846 in file
      ..\src\condor_procap\procap.cpp
```

or:

```
1/20 15:29:14 (pid:2484) ERROR "unable to spawn the ProcD" at line 136
      in file ..\src\condor_c++_util\proc_family_proxy.C
```

and even:

```
4/16 10:49:13 loadavg thread died, restarting. (exit code=2)
```

To enable the performance counters, check the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PerfProc\Performance`. If a value `Disable Performance Counters` exists, delete it or set it to 0.

Why does the Windows Installer fail with "Error 2738. Could not access VBScript run time for custom action"?

This error results when the VBScript engine is not registered. Since Condor's installer depends on the VBScript engine for custom steps, the installer will fail if it cannot find the VBScript engine.

The fix is to register the VMScript engine. With Administrative privilege:

1. Launch the Command Prompt (*cmd.exe*) as the Administrator.
2. At the Command Prompt, change directories to the `System32` folder, within the Windows folder.
3. Issue the command

```
regsvr32 vbscript.dll
```

If successful, the message

```
DllRegisterServer in vbscript.dll succeeded.
```

is printed.

Why does Condor sometimes fail to parse floating point numbers?

Condor assumes that all floating point numbers are of the form `x.y`, which, depending on a computer's current locale, may not always be the case. This problem occurs even if Condor is running under an account that has had the locale configured correctly. The problem lies in the template user account which is used to create Condor's dynamic accounts. Even if the entire system is configured to use a new locale, this template account seems to retain the original system locale. The following steps can be used to fix this problem.

To create a default user profile, you must be logged on as **Administrator** or be a member of the **Administrators** group. Create a new user profile for all new user accounts on a computer to be based on. To create subsequent profiles, you can use the new user account as a template. Here is how to use the new user profile as a template account to use as a new user's profile:

1. **Log on** to the computer as the new user, and customize the desktop if appropriate.
2. Optionally, install and configure any applications to be shared by user accounts made from this template.
3. **Log off**, and then log on as the **Administrator**.
4. In the **Control Panel**, open the **System** Control Panel applet.
 - On **Vista** you will need to click on the **Advanced system settings Task** listed in the left pane.
5. On the **Advanced** tab, under **User Profiles**, click **Settings**.
6. Under **Profiles stored on this computer**, select the user you created to be the template, and then click **Copy To**.
7. To create the default user profile for the computer, type the path to the default user:
 - On Windows 2000: `%WinDir%\Profiles\Default;`
 - On Windows XP: `%SystemDrive%\Documents and Settings\Default;`
 - On Vista: `%SystemDrive%\Users\Default.`
8. In the **Copy To** dialog box, under **Permitted to use**, click **Change**.

9. In the **Select User or Group** dialog box, in the **Enter the object name to select** text box, type: **Everyone** and click **OK**.
10. Click **OK** to dismiss the **Copy To** dialog box.
11. Click **OK** again to dismiss the **User Profiles** dialog box.
12. Finally, click **OK** one last time to dismiss the **System Properties** dialog.

If Condor has already created some dynamic accounts, you will need to remove them so that Condor can re-create them with the new template account.

7.5 Grid Computing

What must be installed to access grid resources?

A single machine with Condor installed such that jobs may be submitted is the minimum software necessary. If matchmaking or glidein is desired, then a single machine must not only be running Condor such that jobs may be submitted, but also fill the role of a central manager. A Personal Condor installation may satisfy both.

I am the administrator at Physics, and I have a 64-node cluster running Condor. The administrator at Chemistry is also running Condor on her 64-node cluster. We would like to be able to share resources. How do we do this?

Condor's flocking feature allows multiple Condor pools to share resources. By setting configuration variables within each pool, jobs may be executed on either cluster. See the manual section on flocking, section 5.2, for details.

What is glidein?

Glidein provides a way to temporarily add a resource to a local Condor pool. Glidein uses Globus resource-management software to run jobs on the resource. Those jobs are initially portions of Condor software, such that Condor is running on the resource, configured to be part of the local pool. Then, Condor may execute the user's jobs. There are several benefits to working in this way. Standard universe jobs may be submitted to run on the resource. Condor can also dynamically schedule jobs across the grid.

See the section on Glidein, section 5.4 of the manual for further information.

Using my Globus gatekeeper to submit jobs to the Condor pool does not work. What is wrong?

The Condor configuration file is in a non-standard location, and the Globus software does not know how to locate it, when you see either of the following error messages.

first error message

```
% globus-job-run \
  globus-gate-keeper.example.com/jobmanager-condor /bin/date
```

Neither the environment variable CONDOR_CONFIG, /etc/condor/, nor ~condor/ contain a condor_config file. Either set CONDOR_CONFIG to point to a valid config file, or put a "condor_config" file in /etc/condor or ~condor/ Exiting.

GRAM Job failed because the job failed when the job manager attempted to run it (error code 17)

second error message

```
% globus-job-run \
  globus-gate-keeper.example.com/jobmanager-condor /bin/date
```

ERROR: Can't find address of local schedd GRAM Job failed because the job failed when the job manager attempted to run it (error code 17)

As described in section 3.2.2, Condor searches for its configuration file using the following ordering.

1. File specified in the CONDOR_CONFIG environment variable
2. /etc/condor/condor_config
3. ~condor/condor_config
4. \$(GLOBUS_LOCATION)/etc/condor_config

Presuming the configuration file is not in a standard location, you will need to set the CONDOR_CONFIG environment variable by hand, or set it in an initialization script. One of the following solutions for an initialization may be used.

1. Wherever *globus-gatekeeper* is launched, replace it with a minimal shell script that sets CONDOR_CONFIG and then starts *globus-gatekeeper*. Something like the following should work:

```
#!/bin/sh
CONDOR_CONFIG=/path/to/condor_config
export CONDOR_CONFIG
exec /path/to/globus/sbin/globus-gatekeeper "$@"
```

2. If you are starting *globus-gatekeeper* using *inetd*, *xinetd*, or a similar program, set the environment variable there. If you are using *inetd*, you can use the *env* program to set the environment. This example does this; the example is shown on multiple lines, but it will be all on one line in the *inetd* configuration.

```
globus-gatekeeper stream tcp nowait root /usr/bin/env
env CONDOR_CONFIG=/path/to/condor_config
/path/to/globus/sbin/globus-gatekeeper
-co /path/to/globus/etc/globus-gatekeeper.conf
```

If you're using *xinetd*, add an *env* setting something like the following:

```
service gsgatekeeper
{
    env = CONDOR_CONFIG=/path/to/condor_config
    cps = 1000 1
    disable = no
    instances = UNLIMITED
    max_load = 300
    nice = 10
    protocol = tcp
    server = /path/to/globus/sbin/globus-gatekeeper
    server_args = -conf /path/to/globus/etc/globus-gatekeeper.conf
    socket_type = stream
    user = root
    wait = no
}
```

7.6 Troubleshooting

If I see **PERMISSION DENIED** in my log files, what does that mean?

Most likely, the Condor installation has been misconfigured and Condor's access control security functionality is preventing daemons and tools from communicating with each other. Other symptoms of this problem include Condor tools (such as *condor_status* and *condor_q*) not producing any output, or commands that appear to have no effect (for example, *condor_off* or *condor_on*).

The solution is to properly configure the *HOSTALLOW_** and *HOSTDENY_** settings (for host/IP based authentication) or to configure strong authentication and set *ALLOW_** and *DENY_** as appropriate. Host-based authentication is described in section 3.6.9 on page 307. Information about other forms of authentication is provided in section 3.6.1 on page 282.

What happens if the central manager crashes?

If the central manager crashes, jobs that are already running will continue to run unaffected. Queued jobs will remain in the queue unharmed, but can not begin running until the central manager is restarted and begins matchmaking again. Nothing special needs to be done after the central manager is brought back on line.

Why did the *condor_schedd* daemon die and restart?

The *condor_schedd* daemon receives signal 25, dies, and is restarted when the history file reaches a 2 Gbyte size limit. Until a larger history file size or the rotation of the history file is supported in Condor, try one of these work arounds:

1. When the history file becomes large, remove it. Note that this causes a loss of the information in the history file, but the *condor_schedd* daemon will not die.
2. When the history file becomes large, move it.
3. Stop keeping the history. Only *condor_history* accesses the history file, so this particular functionality will be gone. To stop keeping the history, place

HISTORY=

in the configuration, followed by a *condor_reconfig* command to recognize the change in currently executing daemons.

When I ssh/telnet to a machine to check particulars of how Condor is doing something, it is always vacating or unclaimed when I know a job had been running there!

Depending on how your policy is set up, Condor will track *any* tty on the machine for the purpose of determining if a job is to be vacated or suspended on the machine. It could be the case that after you ssh there, Condor notices activity on the tty allocated to your connection and then vacates the job.

What is wrong? I get no output from *condor_status*, but the Condor daemons are running.

One likely error message within the collector log of the form

```
DaemonCore: PERMISSION DENIED to host <xxx.xxx.xxx.xxx> for command 0 (UPDATE_STARTD_AD)
```

indicates a permissions problem. The *condor_startd* daemons do not have write permission to the *condor_collector* daemon. This could be because you used domain names in your HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros, but the domain name server (DNS) is not properly configured at your site. Without the proper configuration, Condor cannot resolve the IP addresses of your machines into fully-qualified domain names (an inverse look up). If this is the problem, then the solution takes one of two forms:

1. Fix the DNS so that inverse look ups (trying to get the domain name from an IP address) works for your machines. You can either fix the DNS itself, or use the `DEFAULT_DOMAIN_NAME` setting in your Condor configuration file.
2. Use numeric IP addresses in the `HOSTALLOW_WRITE` and/or `HOSTDENY_WRITE` configuration macros instead of domain names. As an example of this, assume your site has a machine such as `foo.your.domain.com`, and it has two subnets, with IP addresses `129.131.133.10`, and `129.131.132.10`. If the configuration macro is set as

```
HOSTALLOW_WRITE = *.your.domain.com
```

and this does not work, use

```
HOSTALLOW_WRITE = 192.131.133.*, 192.131.132.*
```

Alternatively, this permissions problem may be caused by being too restrictive in the setting of your `HOSTALLOW_WRITE` and/or `HOSTDENY_WRITE` configuration macros. If it is, then the solution is to change the macros, for example from

```
HOSTALLOW_WRITE = condor.your.domain.com
```

to

```
HOSTALLOW_WRITE = *.your.domain.com
```

or possibly

```
HOSTALLOW_WRITE = condor.your.domain.com, foo.your.domain.com, \
bar.your.domain.com
```

Another likely error message within the collector log of the form

```
DaemonCore: PERMISSION DENIED to host <xxx.xxx.xxx.xxx> for command 5 (QUERY_STARTD_ADS)
```

indicates a similar problem as above, but read permission is the problem (as opposed to write permission). Use the solutions given above.

Why does Condor leave mail processes around?

Under FreeBSD and Mac OSX operating systems, misconfiguration of a system's outgoing mail causes Condor to inadvertently leave paused and zombie mail processes around when Condor attempts to send notification e-mail. The solution to this problem is to correct the mailer configuration.

Execute the following command as the user under which Condor daemons run to determine whether outgoing e-mail works.

```
$ uname -a | mail -v your@emailaddress.com
```

If no e-mail arrives, then outgoing e-mail does not work correctly.

Note that this problem does not manifest itself on non-BSD Unix platforms, such as Linux.

Why are there spurious Condor errors on some machines running Xen kernels?

Some older Xen kernels had a problem where the kernel's jiffy counter could jump backwards in time. This breaks an assumption made by the *condor_proc*. This problem can only be worked around by upgrading the Xen kernel to a version that fixes the issue with the jiffy counter. Running Condor on an affected Xen kernel often results in failures of the following forms in Condor daemon log files:

- error: parent process's birthday is later than our own
- ERROR: No family with the given PID is registered

7.7 Other questions

Is there a Condor mailing-list?

Yes. There are two useful mailing lists. First, we run an extremely low traffic mailing list solely to announce new versions of Condor. Follow the instructions for Condor World at <http://www.cs.wisc.edu/condor/mail-lists/>. Second, our users can be extremely knowledgeable, and they help each other solve problems using the Condor Users mailing list. Again, follow the instructions for Condor Users at <http://www.cs.wisc.edu/condor/mail-lists/>.

My question isn't in the FAQ!

If you have any questions that are not listed in this FAQ, try looking through the rest of the manual. Try joining the Condor Users mailing list, where our users support each other in finding answers to problems. Follow the instructions at <http://www.cs.wisc.edu/condor/mail-lists/>. If you still can't find an answer, feel free to contact us at condor-admin@cs.wisc.edu.

Note that Condor's free e-mail support is provided on a best-effort basis, and at times we may not be able to provide a timely response. If guaranteed support is important to you, please inquire about our paid support services.

Version History and Release Notes

8.1 Introduction to Condor Versions

This chapter provides descriptions of what features have been added or bugs fixed for each version of Condor. The first section describes the Condor version numbering scheme, what the numbers mean, and what the different *release series* are. The rest of the sections each describe a specific release series, and all the Condor versions found in that series.

8.1.1 Condor Version Number Scheme

Starting with version 6.0.1, Condor adopted a new, hopefully easy to understand version numbering scheme. It reflects the fact that Condor is both a production system and a research project. The numbering scheme was primarily taken from the Linux kernel's version numbering, so if you are familiar with that, it should seem quite natural.

There will usually be two Condor versions available at any given time, the *stable* version, and the *development* version. Gone are the days of “patch level 3”, “beta2”, or any other random words in the version string. All versions of Condor now have exactly three numbers, seperated by “.”

- The first number represents the major version number, and will change very infrequently.
- *The thing that determines whether a version of Condor is “stable” or “development” is the second digit. Even numbers represent stable versions, while odd numbers represent development versions.*
- The final digit represents the minor version number, which defines a particular version in a given release series.

8.1.2 The Stable Release Series

People expecting the stable, production Condor system should download the stable version, denoted with an even number in the second digit of the version string. Most people are encouraged to use this version. We will only offer our paid support for versions of Condor from the stable release series.

On the stable series, new minor version releases will only be made for bug fixes and to support new platforms. No new features will be added to the stable series. People are encouraged to install new stable versions of Condor when they appear, since they probably fix bugs you care about. Hopefully, there won't be many minor version releases for any given stable series.

8.1.3 The Development Release Series

Only people who are interested in the latest research, new features that haven't been fully tested, etc, should download the development version, denoted with an odd number in the second digit of the version string. We will make a best effort to ensure that the development series will work, but we make no guarantees.

On the development series, new minor version releases will probably happen frequently. People should not feel compelled to install new minor versions unless they know they want features or bug fixes from the newer development version.

Most sites will probably never want to install a development version of Condor for any reason. Only if you know what you are doing (and like pain), or were explicitly instructed to do so by someone on the Condor Team, should you install a development version at your site.

After the feature set of the development series is satisfactory to the Condor Team, we will put a code freeze in place, and from that point forward, only bug fixes will be made to that development series. When we have fully tested this version, we will release a new stable series, resetting the minor version number, and start work on a new development release from there.

8.2 Upgrade Surprises

Occasional changes to Condor can cause unexpected errors or results to users. Here is a list of changes to note and be aware of.

- We believe that Condor 7.2.x and 7.0.x are wire-compatible, and can be freely mixed between computers in a Condor pool. However, we do not regularly test this compatibility and cannot guarantee it. Therefore, we recommend using a single release of Condor when possible. Please note that although Condor 7.2.x and 7.0.x may be mixed within a pool, they ought not to be mixed them on a single computer. That is, a *condor_master* daemon running 7.0.x cannot run Condor daemons from version 7.2.x, or vice-versa.

8.3 Stable Release Series 7.2

This is a stable release series of Condor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 7.2.x releases. New features will be added in the 7.3.x development series.

The details of each version are described below.

Version 7.2.3

Release Notes:

- The header files for ClassAds are now included within the release.

New Features:

- Enhanced the Debian 5.0 Condor port on the x86_64 platform to include support for the standard universe.

Configuration Variable Additions and Changes:

- The new integer configuration variable `SEC_TCP_SESSION_DEADLINE` specifies the number of seconds after which the client should give up its attempt to establish a security session with a daemon that it is connecting to. The default value is 120 seconds.
- The new configuration variables `SCHEDD_CLUSTER_INITIAL_VALUE` and `SCHEDD_CLUSTER_INCREMENT_VALUE` are integers that specify the cluster number to use for the first job submission, and the stride used to increment the cluster id upon successive submissions. See 3.3.11 and 3.3.11 for the complete definitions of these variables.

Bugs Fixed:

- Fixed a memory leak in the *condor_collector* daemon. The growth in memory over time was approximately 10Mbytes per day per 1000 slots. This bug was introduced in Condor version 7.2.0.
- Fixed a problem that caused integrity checking of most UDP packets longer than about 40Kbytes to fail. This bug affected all previous versions of Condor.
- By adding the new configuration variable `SEC_TCP_SESSION_DEADLINE`, fixed a problem that has existed since Condor version 7.1.2. The problem was that non-blocking read operations in the security handshake had no timeout, and could therefore lead to a socket remaining

allocated indefinitely, if the other side of the connection did not respond. When this problem was observed, the following message appeared in the log written by the *condor_schedd* daemon:

```
file descriptor safety level exceeded
```

- Fixed a rarely observed bug in the event log reader code that could cause it to not detect missed events.
- A bug in the Chirp java client has been fixed. The *ChirpInputStream*'s *read()* method was returning negative values when encountering binary data.
- *condor_dagman* now rejects negative node retry values.
- *condor_dagman* no longer generates a rescue DAG if the DAG is aborted, but is considered successful; this is when ABORT-DAG-ON returns the value 0.
- The user log event numbered 27, named "Job submitted to grid resource", is now written for all grid universe jobs. Previously, it was not written for pbs, lsf, nordugrid, or unicore grid types.
- Fixed a bug where a Condor-C job with both *remote_<foo>* and *remote_remote_<foo>* attributes would not have a *remote_<foo>* attribute when submitted to the remote *condor_schedd* daemon.
- Fixed a bug in *condor_configure* and *condor_install* that would leave the configuration variable *CONDOR_HOST* unset when configuring a central manager without using the *-central-manager* command-line argument.
- Fixed a bug that could cause the *condor_schedd* daemon to leak memory and file descriptors when using the *EVENT_LOG* configuration variable.
- Fixed a bug in the *condor_gridmanager* that could cause it to not send a clean up signal to the GRAM jobmanager for removed gt2 jobs.
- Fixed a bug that caused parallel jobs to not work when encryption was enabled.
- Fixed a bug in the Windows installer that caused it to fail to start Condor.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.2.2

Release Notes:

- None.

New Features:

- Added a full port of Condor to Debian 5.0 on the x86 platform.
- Added a clipped port of Condor to Debian 5.0 on the x86_64 platform.
- Added the **-DumpRescue** command-line flag to *condor_dagman* and *condor_submit_dag*. This flag is intended mainly for testing.
- Added support for the **-debug** option to *condor_qedit*.
- The Job Router now uses a time slice timer for periodic expression evaluation, similar to the *condor_schedd* daemon. The evaluation interval is controlled by the configuration variable `PERIODIC_EXPR_INTERVAL`, and defaults to 60 seconds, the same default value used by the *condor_schedd* daemon.
- The Job Router now resets the source job, if a failure occurs when updating the *condor_schedd* daemon for a periodic expression that evaluated to `True`. The job's periodic expressions should be evaluated again some time in the future with a successful update.

Configuration Variable Additions and Changes:

- The new boolean configuration variable `EVENT_LOG_FSYNC` provides control of the behavior of Condor when writing events to the event log. Previously, the behavior was as if this parameter were set to `False`. See 3.3.4 for the complete definition of this variable.
- The new boolean configuration variable `EVENT_LOG_LOCKING` provides control of the behavior of Condor when writing events to the event log. Previously, the behavior was controlled by `ENABLE_USERLOG_LOCKING`. See 3.3.4 for the complete definition of this variable.
- The new string configuration variable `TRANSFERER` specifies the path to the *condor_transferer* program which is invoked by the *condor_replication* daemon to perform the actual transfer of the file set by `STATE_FILE`. This is part of the high availability framework. Prior to Condor 7.2.2, the value of `TRANSFERER` was hard coded to `$(RELEASE_DIR)/sbin/condor_transferer`. The use of this hard coded behavior should be considered obsolete behavior, and will be removed in a future version of Condor.
- The `PREEMPTION_REQUIREMENTS` and the `RANK` expression in the matchmaker can now reference many more ClassAd attributes than just `SubmittorPrio`. New attributes allow this expression to take into account resources currently in use, as well as group

usage and quota info. New attributes are: SubmitterUserResourcesInUse, RemoteUserResourcesInUse, RemoteGroupResourcesInUse, RemoteGroupQuota, SubmitterGroupResourcesInUse, SubmitterGroupQuota.

- Added `JOB_ROUTER_ATTRS_TO_COPY` configuration option. This is a comma separated list of attributes that the Job Router should copy from the routed ad to the source ad in addition to internally hard coded attributes that are copied.
- Added `JOB_ROUTER_RELEASE_ON_HOLD` configuration option that will control whether the Job Router will reset the source job to an untouched state if it needs to yield the job because the routed job went on hold. The option defaults to resetting the source job.
- The new configuration variables `PREEMPTION_REQUIREMENTS_STABLE` and `PREEMPTION_RANK_STABLE` identify for Condor if all attributes in the variables `PREEMPTION_REQUIREMENTS` and `PREEMPTION_RANK` will not change within a negotiation interval.
- The new configuration variables `OFFLINE_LOG` and `OFFLINE_EXPIRE_ADS_AFTER` specify the location of persistent machine ClassAds for hibernating machines, as well as the lifetime of the persistent ClassAds.

Bugs Fixed:

- Fixed the *condor_collector* daemon such that hibernating machines never time out.
- Fixed incorrectly set ClassAd attribute values of machines entering a hibernation state. All hibernating machines are unclaimed and idle, they have no load, the CPU is not busy, and the keyboard and console appear as if they had been idle for a long time.
- Fixed a bug where if any idle slot satisfied the `HIBERNATE` expression, Condor would put the machine into a sleep state irrespective of any active slots.
- Fixed a bug on Windows that made it impossible to use the defined string "S5" for hibernation.
- Fixed a bug in the *condor_starter* where it would be running as real uid condor after job hooks are invoked which causes issues when accessing files.
- Fixed a bug where some machines would send a final update ad to the *condor_collector*, invalidating the persistent one that was previously sent (when `HIBERNATE` evaluates to `True`). This had the effect of dropping the machine out of the pool once the ad had grown stale.
- Fixed a bug where any two Condor daemons on Windows were able to bind to the same port at the same time.
- Fixed the behavior of the *condor_negotiator* so that when a Condor-G matchmaking ad matches, the machine's ad will be shuffled to the end for round-robin matching to multiple gatekeepers with the same rank.

- Resolved a bug in which the submit description file command **vm_macaddr** was improperly parsed, and thus ignored, by *condor_submit* for vm universe jobs.
- Condor's Windows zip file distribution now includes the new C/C++ runtime libraries.
- Fixed a Windows platform bug for jobs that enable streaming I/O. The bug caused the *condor_starter* to crash upon invocation of the job.
- Fixed a bug in which an ill-formed network packet could crash a Condor daemon. This would not be seen in normal Condor operation, but sometimes port-scanning software could trigger such a crash.
- Fixed a bug in which *condor_q* would sometimes exit with the value zero, indicating success, when it could not connect to a *condor_schedd* daemon. It now exits with an error code.
- Fixed two seemingly small memory leaks in Condor's SOAP interface. A small amount of memory was lost per SOAP transaction. On a high traffic machine, this leak would eventually render the *condor_schedd* daemon unresponsive.
- Fixed a bug in the parallel universe where periodic expressions involving the `JobStatus` attribute would not function properly.
- Fixed a bug where Condor daemons could segmentation fault while trying to write a core file to disk in the Unix ports.
- Fixed a bug in which the use of dedicated execute accounts (indicated by use of the configuration variable `DEDICATED_EXECUTE_ACCOUNT_REGEX`) did not work properly in PrivSep mode: those with the configuration variable `PRIVSEP_ENABLED` set to `True`.
- Fixed an erroneous log message that reported that the hook defined by `HOOK_UPDATE_JOB_INFO` had run, but would print the `$(HOOK_PREPARE_JOB)` path. The correct hook ran, so this was only a logging error. The log message is only visible at the `D_FULLDEBUG` level.
- Fixed a bug that caused *condor_dagman* to crash if the `dagman.out` file reached a size of 2 GBytes.
- Fixed a problem affecting the *condor_starter* when in PrivSep mode. After the user job exited, an error was printed in the *condor_starter* log file complaining that it failed to *chown* the sandbox to Condor ownership. This error was not actually harmful, just noisy.
- Fixed a bug in the *condor_master* that caused it to not have `REPLICATION` in its default list for `DC_DAEMON_LIST`. The example configuration file for HAD has been updated to match, as well.
- Fixed the *condor_transferer* daemon and documentation to consistently use the value of the configuration variable `MAX_TRANSFERER_LIFETIME` in High Availability code.
- Fixed a bug that caused *condor_dagman* to crash, if a splice DAG has node categories.

- Changed splice-related *condor_dagman* debug messages to *not* be printed at the default verbosity. They are now mostly printed at debug level 4. For definitions of the debug levels, see the *condor_dagman* manual page at section 9.
- Fixed a bug that caused the *condor_replication* daemon, as part of the high availability framework, to start the *condor_transferer* client incorrectly; the end result was that the *condor_transferer* was unable to authenticate via GSI using host-based certificates.
- Fixed a bug in which the ClassAd attribute `RemoteWallClockTime` could get too big after a restart of the *condor_schedd* daemon, for jobs that were running at the time of the restart.
- Fixed a bug that was causing the *condor_startd* to log the error message

```
ioctl(SIOCETHtool/GWOL) failed: Operation not permitted (1)
```

when started as a Personal Condor on Linux. The message is now suppressed in this case. When the message is printed, an additional message is logged informing the user that this error can be ignored, unless hibernation is being used.

- Fixed a bug that was causing the *condor_startd* to sometimes publish the network adapter's hardware address incorrectly in its ClassAd.
- Fixed a case in which *condor_history* could get into an infinite loop when searching through a corrupted history file.
- Fixed a bug in the user log reader code that could cause it to get into an inconsistent state after detecting missed events.
- Condor version 7.2.2 and previous releases do not support communication with Condor 7.3.x daemons using the new 7.3.x configuration variables `CCB_ADDRESS` or `PRIVATE_NETWORK_NAME`. The version 7.2.2 *condor_collector* daemon now recognizes when it is receiving ClassAds from such daemons, and it will reject them. In prior versions, Condor would accept the ClassAds, but attempts to use them led to unexpected behavior.

Known Bugs:

- None.

Additions and Changes to the Manual:

- A manual page for *condor_power* now appears in the manual. *condor_power* sends a packet to a machine in a low power state, to cause the machine to wake from that state.
- Reorganized the user manual section that describes DAGMan.
- Added a note about the fact that environment values specified with the **environment** submit description file command override values from the submitter's environment, as imported with **getenv = True**.
- Added new information to the section on Power Management pertaining to the handling of hibernating machines.

Version 7.2.1

Release Notes:

- This release addresses reported 7.2.0 problems with the Windows distribution.

New Features:

- Condor now has a clipped port to i386 Debian 5.0 (Lenny).
- Added standard universe support for *gfortran*.
- Added support for standard output and standard error to be greater than 2 Gigabytes.

Configuration Variable Additions and Changes:

- The configuration variable `JAVA_MAXHEAP_ARGUMENT` now defaults to the value **-Xmx1024m**. The installation process of Condor resets this value to `UNDEFINED` in the local configuration file, if the detected JVM is not from Sun Microsystems.
- A new feature has been added to the *condor_master* that makes it possible to append to the `DC_DAEMON_LIST` configuration variable, instead of overwriting it. To take advantage of this, place the plus character ('+') as the first character in the `DC_DAEMON_LIST` definition. For example:

```
DAEMON_LIST      = NEW_DAEMON
DC_DAEMON_LIST   = +NEW_DAEMON
```

- The new configuration variable `DAGMAN_COPY_TO_SPOOL` controls whether the *condor_dagman* binary gets copied to the spool directory when a DAG is submitted. See 3.3.25 for details.
- Added **-version** and **-help** command line options to *condor_submit_dag*.

Bugs Fixed:

- Fixed a bug in the *condor_collector* which could cause it to hang indefinitely while reading network input in rare conditions.
- Fixed a bug in *condor_chirp* for Windows which was causing it to crash on invocation.
- Fixed a bug in the Windows *condor_mail* program, which was causing it to become unresponsive when run. If left running, the application also increased its memory consumption.
- Fixed a bug that could cause the *condor_schedd* to never evaluate periodic expressions.

- Fixed a bug on Unix platforms where *condor_configure* would provide incorrect defaults for the `JAVA_MAXHEAP_ARGUMENT` attribute in the installed configuration files. The new current default for Sun Java JVMs is **-Xmx1024m**.
- Fixed a bug on Unix platforms where *condor_configure* would imply that using the Unix user `root` or UID 0 for the **-owner** option is a good thing. It is not, and would then complain that it could not find user `root` in the password file.
- Fixed a bug on Unix platforms where *condor_configure* would emit errors about not being able to execute *ldd* when installing Condor on the Mac OS X 10.5 platform. *condor_configure* now correctly detects shared library requirements when installing the Condor binaries on the Mac OS X 10.5 platform.
- Fixed a bug where execute-side daemons started before the *condor_credd* would fail to match with Windows jobs with **run_as_owner** set. This condition persisted until the execute-side daemons were either restarted or reconfigured.
- Fixed a problem affecting the Job Router and Condor-C. When jobs spool input files, they enter a temporary hold state, which could trigger actions by a naive periodic remove or release expression. Periodic expressions are no longer evaluated when in this temporary hold state, which has the hold reason "Spooling input data files".
- The example init script *condor.boot.generic* erroneously claimed that the *condor_master* would begin sending SIGKILL to child processes after 20 seconds if SIGQUIT (the fast shutdown) failed. The *condor_master* will actually wait `$(SHUTDOWN_FAST_TIMEOUT)` seconds, a value that currently defaults to 300 seconds.
- Environment variable names are now properly treated as case-insensitive on Windows. The most common symptom of this bug was the inability to specify a custom `PATH` environment variable for a job from its submit description file.
- Changed *condor_submit -debug* to issue a warning when ignoring environment variables. This occurs with **getenv = True** set in a submit description file.
- Fixed a long-standing memory leak in SOAP interface. This caused the leak of a few hundred bytes of memory for each connection. This could eventually have caused the *condor_schedd* daemon to crash.
- Fixed Job Router hooks so that their output is properly propagated where appropriate.
- Implemented a fix for the *condor_startd* that prevents it from crashing if the user specified the configuration variable `NUM_SLOTS_TYPE_N`, without also specifying `SLOT_TYPE_N`.
- The sample configuration files now correctly set the default universe to vanilla. This default has been true since 7.2.0, but was not reflected in the sample configuration files.
- Fixed a bug that incorrectly set the value of the job ClassAd attribute `RequestMemory` to be 1024 times its correct size due to a mismatch in units; the attribute `RequestMemory` is given in Mbytes, while the attribute `ImageSize` is given in Kbytes.

- Fixed a memory leak in *condor_dagman* that leaked a small amount of memory for each job submitted.
- Fixed a bug that was causing the network mask to be advertised as a Condor sinful string, rather than a dotted-quad.
- Fixed a handle leak in the *condor_procd* on Windows.

Known Bugs:

- None.

Additions and Changes to the Manual:

- Added a FAQ entry for Windows describing how machines with miss-configured performance counters may cause the *condor_procd* to crash.
- Added a manual page for the command *condor_router_history*.

Version 7.2.0

Release Notes:

- A bug in some older Xen kernels can result in Condor errors due to a broken assumption in the *condor_procd* daemon. See the FAQ entry at section 7.6 for details.
- A problem has been discovered when using snapshot disks with **vm** universe VMware jobs, if the path that the *condor_vm-gahp* uses to refer to the virtual machine's VMX file contains a symbolic link. See the FAQ entry at section 7.3 for details.
- The name of the Amazon EC2 GAHP binary has changed from *amazon-gahp* to *amazon_gahp*. This makes it consistent with the naming of other Condor binaries.

New Features:

- The default **universe** for jobs is now **vanilla**, instead of **standard**. The default can be changed using the configuration variable `DEFAULT_UNIVERSE`.
- VMware **vm** universe jobs now have any BIOS settings saved in an `nvr` file in the **vmware_dir** given in the job's submit file transferred to the execute machine, so that they apply to the job's execution.
- Daemons that become unresponsive are now killed using the SIGABRT signal, which causes a core file to be dropped. Setting the configuration variable `NOT_RESPONDING_WANT_CORE` to `False` will revert to the previous behavior that used the SIGKILL signal.

- The *condor_job_router* and the *condor_q* command with the **-better-analyze** option now support more ClassAd functions than they previously did. They now support all ClassAd functions, except for those with names beginning with the string `stringList`.
- *condor_status* given the options **-submitters -xml** no longer emits a single blank line when there are no submitters, instead it prints valid XML output with an empty body.

Configuration Variable Additions and Changes:

- The HAD configuration variable `NEGOTIATOR_STATE_FILE` has changed its name to `STATE_FILE`.

Bugs Fixed:

- Security Item: A flaw was found and fixed that could allow an unauthenticated user to cause Condor daemons to shut down, and could allow running jobs to be removed from the queue.
- Fixed a bug that caused *condor_dagman* to stay in the Condor queue, if *condor_dagman* was accidentally submitted with an empty DAG input file.
- *condor_submit_dag* now generates a `.condor.sub` file with the submit description file command **copy_to_spool** set to `True`, to ease version upgrades while large DAGs are running.
- Fixed a problem in the *condor_startd* when using `STARTD_SLOT_EXPRS` for attributes that are sometimes present and sometimes absent from the machine ClassAd. This is most typical of attributes that enter the machine ClassAd from the job, via `STARTD_JOB_EXPRS`. When the attribute went away from slot X (for example, because the job on slot X finished), the corresponding `SlotX-<AttributeName>` attribute was not reliably removed from all of the other slots.
- Removed some redundant information from the *condor_startd* advertisements to the *condor_collector*, from within the private ClassAd that is not user-visible. This fix reduces UDP traffic and memory usage generated by the *condor_startd* by about 20% in the *condor_collector* and *condor_negotiator* daemons.
- Fixed the *condor_master* daemon to correctly preserve all command-line arguments when restarting itself. In some cases, not preserving `argv[0]` confused external utilities that monitor the *condor_master* process by looking at the output of *ps* or similar programs. Also, not preserving **-pid** and **-runfor** could cause unexpected behavior.
- Fixed a bug that exhibited itself when the configuration variable `NEGOTIATOR_CONSIDER_PREEMPTION` was set to `False`, in which jobs would not be matched to slots in the backfill state. Corrected, slots doing backfill are included in the matchmaking process.

- The *condor_job_router* did not work while managing jobs from multiple users when read access to the *condor_schedd* required authentication. The *condor_job_router* was also not able to use authentication methods other than FS. Now it can use any authentication method, as long as the resulting identity is listed in the configuration variable `QUEUE_SUPER_USERS` or the *condor_job_router* and *condor_schedd* are running as a Personal Condor in non-root mode.
- Fixed a bug in the *condor_schedd* daemon that could cause it to write an incorrect Unique ID to the event log's header.
- Fixed a bug in the user log reader API that could cause it to incorrectly return a `ULOG_NO_EVENT` in rare cases.
- Fixed a bug in the user log reader API that could cause it to crash if the application attempted to re-initialize the `ReadUserLog` object. The code now detects this condition, and returns an error when the application attempts to re-initialization an already initialized `ReadUserLog` object.
- Fixed a bug that limited the size of `stdin`, `stdout`, and `stderr` files in the vanilla universe to 2GBytes.
- Fixed a bug that could cause the *condor_starter* to `EXCEPT` upon completion or eviction of a **vm** universe job. The error message that appeared in the `StarterLog` file was

```
Write_Pipe: invalid pipe end
```

- When a held job is removed, the values of the attributes `HoldReason`, `HoldReasonCode` and `HoldReasonSubCode` are moved to `LastHoldReason`, `LastHoldReasonCode` and `LastHoldReasonSubCode`. Before, a hold reason could be lost if a removed job was subsequently held.
- The executable attribute for amazon grid universe jobs no longer needs to be a valid file path.
- Improved error reporting when a Xen or VMware command fails in the **vm** universe.
- For **vm** universe jobs, virtual floppy disks are no longer disabled.
- Fixed a bug introduced in Condor 7.1.4 that caused Condor to ignore the virtual machine status reported by Xen in the **vm** universe.
- Fixed a 20-second delay in the start up of the *condor_c-gahp* and the *condor_vm-gahp*.
- Fixed a bug which caused the net mask to be published into the machine `ClassAd` incorrectly.
- Fixed a bug introduced in Condor 7.1.4 which could cause any Condor daemon to crash if the level of debugging output `D_ALL` is enabled when a *condor_reconfig* command is issued.
- Fixed a bug introduced in Condor 7.1.4 which caused standard universe jobs to fail to start up, if security authentication, but not encryption was enabled between the submit side and the execute side.
- Fixed a bug with streaming `stdin`, `stdout`, and `stderr` when using *glexec*.

Known Bugs:

- None.

Additions and Changes to the Manual:

- Initial documentation for dynamic provisioning is available in section 3.12.7.
- Documentation for Kerberos authentication (see section 3.6.3) and associated configuration variables has been updated.

8.4 Development Release Series 7.1

This is the development release series of Condor. The details of each version are described below.

Version 7.1.4

Release Notes:

- The owner of the log file for the *condor_vm-gahp* has changed to the `condor` user. In Condor 7.1.2 and previous versions, it was owned by the user that the virtual machine is started under. Therefore, the owner of and permissions on an existing log file are likely to be incorrect. To correct the problem, an administrator may modify file permissions such that the `condor` user may read and write the log file. Alternatively, an administrator may delete the file, and Condor will create a new file with the expected owner and permissions. In addition, the definition for `VM_GAHP_LOG` in the `condor_config.generic` file has changed for Condor 7.1.3.
- The **vm** universe no longer supports the use of the **xm** command for running Xen virtual machines. The **virsh** tool should be used instead.
- Condor no longer supports the standard universe feature in its ports to Solaris. We may resurrect this feature in the future if demand for it on this port grows again to sufficient levels.

New Features:

- Local entries in the configuration file may now be specified by pre-pending a local name and a period to the normal name. Local settings take precedence over the other settings. The local name can be specified on the command line to all daemons via the new **-local-name** command line option.

See section 3.3.1 for more details on how the local name will be used in the configuration, and section 3.9.2 for more details on the command line parameters.

- Dynamic Startd Provisioning: New configuration options allow for slots to be broken into job-sized pieces. While this feature is still under ongoing development, we felt that what we had so far, although not yet fulfilling our complete vision, is useful enough in its present form to bring value to some installations.
- *condor_submit_dag* is now automatically run recursively on nested DAGs (unless the new **-no_recurse** option is specified). See ?? for details.
- Added the new SUBDAG EXTERNAL keyword (for specifying nested DAGs) to *condor_dagman*. See ?? for details.
- It is now possible to have multiple rotations of the “event log” file, such as “EventLog”, “EventLog.1”, “EventLog.2”, ...
- The VM universe can now run VMware virtual machines on machines using privilege separation without requiring the *condor_vm-gahp* binary to be setuid root. Running the *condor_vm-gahp* as setuid root is no longer supported for VMware or Xen.
- Condor now supports the ability for the *condor_master* to run a program as it shuts down. This can be particularly useful for doing a graceful shutdown, followed by, a reboot. This is accomplished through the new MASTER_SHUTDOWN_<Name> configuration variable. The configuration variable MASTER_SHUTDOWN_<Name> is defined on page 175), and the manual page for *condor_set_shutdown* is on page 781.
- The *condor_lease_manager* is a new daemon. It provides a mechanism for managing leases to resources described by Condor’s ClassAd mechanism. These resources and leases are managed to be persistent.
- VM universe now works with privilege separation (PrivSep) for VMware jobs. Xen is still not supported in PrivSep mode.
- Added the *DIR* directive for the *SPLICE* keyword in the DAGMan language. Please read section 2.10.6 on page 80 for more information.
- For gt4 type grid jobs (i.e. WS GRAM), include a request to retry failed attempts at file clean-up in the RSL job description.
- Improved the scalability of some algorithms used by the *condor_schedd* and *condor_negotiator* when dealing with large numbers of startds.
- Added the ability for the *condor_master* (actually, any DaemonCore process with children) to kill child processes that have quit responding SIGABRT instead of SIGKILL. This is for debugging purposes on UNIX systems, and is controlled by the new NOT_RESPONDING_WANT_CORE configuration parameter. If the child process is configured with CREATE_CORE_FILES enabled, the child process will then generate a core dump. This feature is currently implemented only on UNIX systems.

See NOT_RESPONDING_WANT_CORE on page 164, NOT_RESPONDING_TIMEOUT on page 164, and CREATE_CORE_FILES on page 156 for more details.

- Condor can now be configured to keep a backup of the job queue log on a local file system in case *condor_schedd* operations involving writes, flushes, or syncs to the job queue log fail. This is most likely to happen when the job queue log is stored on a network file system like NFS. Such a backup enables an administrator to see that a job failed to submit, but does not perform any automatic recovery. See below for the these configuration parameters.
- Added preliminary support for “Green Computing”. This is supported only on Linux and Windows. See section 3.15 on page 426 on “Power Management” for more details.

Configuration Variable Additions and Changes:

- Local versions of configuration parameters can now be specified via the use of the “-local-name” command line parameters (see the above “New Features” entry).
- A new configuration parameter `EVENT_LOG_MAX_ROTATIONS` has been added to allow multiple rotations of the event log file. See 161 for details.
- A new configuration parameter `EVENT_LOG_ROTATION_LOCK` has been added to allow allow configuration of an alternate file for Condor to use while rotating event log files. See 161 for details.
- The configuration parameter `MAX_EVENT_LOG` has been renamed to `EVENT_LOG_MAX_SIZE`. For backward compatibility, if `EVENT_LOG_MAX_SIZE` is not defined, Condor will also try `MAX_EVENT_LOG`. See 161 for details.
- The *condor_vm-gahp* no longer requires its own configuration file. It now uses the normal Condor configuration file. Parameters that used to reside in the *condor_vm-gahp*’s file should now be placed in the Condor configuration file.
- The following VM universe-related configuration parameters have been removed:
 - `VM_GAHP_CONFIG`
 - `VM_MAX_MEMORY`
 - `XEN_CONTROLLER`
 - `XEN_VIF_PARAMETER`
 - `XEN_NAT_VIF_PARAMETER`
 - `XEN_BRIDGE_VIF_PARAMETER`
 - `XEN_IMAGE_IO_TYPE`

`VMWARE_LOCAL_SETTINGS_FILE` and `XEN_LOCAL_SETTINGS_FILE` have been added. They allow a machine administrator to add settings to the virtual machine configuration files written by Condor for VMware and Xen. See 234 and 235 for details.

- The configuration parameter family `MASTER_SHUTDOWN_<Name>` can be used in conjunction with *condor_set_shutdown* to cause the *condor_master* to execute a specified program as it shuts down. See 175 and *condor_set_shutdown* manual page for more details.

- The configuration parameter `NOT_RESPONDING_WANT_CORE` controls the type of signal sent to child processes that DaemonCore has determined are no longer responding. See the above discussion of the addition of this feature and `NOT_RESPONDING_WANT_CORE` on page 164 for details.
- The configuration parameter `LOCAL_QUEUE_BACKUP_DIR` should be set to the pathname of a directory that is writable by the Condor user and is located on a non-network file system. This is part of the “Job Queue Backup” feature, above.
- The configuration parameter `LOCAL_XACT_BACKUP_FILTER` controls whether or not the *condor_schedd* will attempt to keep backups of transactions that were not written the job queue log. If it is set to `FAILED`, the *condor_schedd* will attempt to keep a backup of the transaction in the local queue backup directory, defined by `LOCAL_QUEUE_BACKUP_DIR`, only if operations fail on the job queue log. If it is set to `NONE`, no backups should be performed even in the event of failure. If it is set to `ALL`, then at all transactions should be backed up. The `ALL` value will create quite a large number of files and slow the *condor_schedd* substantially; it is only likely to be useful for users who are developing or debugging Condor. This is part of the Job Queue Backup feature.

Bugs Fixed:

- In some rare cases, the *condor_startd* failed to fully preempt jobs. The job itself was killed, but the *condor_starter* process watching over it would not be killed. The slot would then stay in the Preempting state indefinitely.
- *condor_q* performed poorly when querying a remote pool, using **-pool**. It was using an older latency-bound protocol even when the remote *condor_schedd* was new enough to use the improved protocol that first appeared in version 6.9.3.
- When using `USE_VISIBLE_DESKTOP` the user’s (slot or owner) access-control entry removed from the Desktop’s access-control list. This fixes the previous behavior where users were added and never removed, resulting in an overflow in access-control list, which can only contain a fixed number of access-control entries.
- Fixed a bug where if log line caching was enabled in *condor_dagman* and *condor_dagman* failed during the recovery process, the cache would stay active. Now the cache is disabled in all cases at the end of recovery.
- Fixed a couple of bugs relevant only to the `GLEEXEC_STARTER` mode of operation. One bug would result in the `SPOOL` directory being deleted if local universe jobs (which are not supported in `GLEEXEC_STARTER` mode) were submitted. The other bug prevented COD jobs from running. Neither of these are problems for the newer recommended `GLEEXEC_JOB` mode.
- Fixed a bug that could cause the *condor_procd* to crash, depending on the timing of its process snapshots.
- Fixed a bug that caused job status notifications from WS GRAM 4.2 servers to be lost.

- Fixed a file descriptor leak in the *condor_vm-gahp*.
- Jobs now go on hold with a clear hold reason if a path to a directory is put in the transfer files list. Previously, the attempt to run the job would simply fail and return to the idle state.
- If `MAX_EVENT_LOG` set to 0, then let event log grow without bounds. Previously this behavior was broken, and setting `MAX_EVENT_LOG` to 0 resulted in the log rotating with every event. Now it works as documented.

Known Bugs:

- When fixing the `USE_VISIBLE_DESKTOP` bug, a new one was inadvertently introduced. The bug manifests irrespective of the definition of `USE_VISIBLE_DESKTOP`: the new code attempts to remove the current user's access-control entry from the Desktop's access-control list even when it was not added by Condor. This has the effect of inhibiting the creation of new process for the logged on user.

Additions and Changes to the Manual:

- The extra space character injected into the names of Condor daemons and programs has been removed.
- Previously undocumented Condor Perl module subroutines have been documented.

Version 7.1.3

Release Notes:

- This developer release includes the majority of the bug fixes released in stable version 7.0.5, including the security patches documented in that release. See section 8.5 below.
- Updated the version of Globus Toolkit: The Condor binaries are now linked against Globus v4.2.0.
- Updated the version of OpenSSL: The Condor binaries are now linked against OpenSSL 0.9.8h.
- Updated the version of GCB: The Condor binaries are now linked against GCB 1.5.6.
- Changes to the `ALLOW_*` and `DENY_*` configuration variables no longer require the use of the **-full** option to *condor_reconfig* upon reconfiguration.

New Features:

- Added a new mechanism termed *Concurrency Limits*. This mechanism allows the Condor pool administrator to define an arbitrary number of consumable resources in the configuration file of the matchmaker. The availability of these consumable resources will be taken into account during the matchmaking process. Individual jobs can specify how many of each type of consumable resource is required. Typical applications of Concurrency Limits could include management of software licenses, database connections, or any other consumable resource that is external to Condor. NOTE: Documentation still being written on this feature. See section 3.12.11) for documentation.
- Added support for Condor to manage serial high throughput computing workloads on the IBM Blue Gene supercomputer. The IBM Blue Gene/P is now a supported platform.
- Extended Job Hooks (see section 4.4) to allow for alternate transformation and/or monitoring engines for the Job Router (see section 5.6. Routing is still controlled by the Job Router, but if Job Router Hooks are configured, then external programs or scripts can be used to transform and monitor the job instead of Condor's internal engine.
- Added support for the new protocol for WS GRAM introduced in Globus 4.2. For each WS GRAM resource, Condor automatically determines whether it is speaking the 4.0 or 4.2 version of the protocol and responds appropriately. When setting **grid_resource** in the submit file, use **gt4** for both WS GRAM 4.0 and 4.2.
- Added the ability for Windows slot users to load and run their jobs within the context of their profile. This includes the My Documents directory hierarchy, its monikers, and the user's registry hive. To use the profile, add a **load_profile** command to the submit description file. A current restriction prevents the use of **load_profile** in conjunction with **run_as_owner**. Please refer to section 6.2.5 for further details.
- The StarterLog file for local universe jobs now displays the job id in each line in the file, so that interleaved messages relevant to different jobs running concurrently can be identified.
- Added the **-AllowVersionMismatch** command line option to *condor_submit_dag* and *condor_dagman* to (if absolutely necessary) allow a version mismatch between *condor_dagman* and the *.condor.sub* file used to submit it. This permits a Condor version mismatch between *condor_submit_dag* and *condor_dagman*.
- Streamlined the protocol between submit and execute machines; in some instances, fewer messages will be exchanged over the network.
- When network requests are denied because of the authorization policy, Condor now logs an explanation in the daemon log that denied the request. This helps the administrator understand why the policy denied the request, in case it is not obvious. A similar explanation may be logged for requests that are accepted. This is only generated if **D_SECURITY** is added to the daemon's debug options.

Configuration Variable Additions and Changes:

- Added the new configuration variable `MAX_PENDING_STARTD_CONTACTS`. This limits the number of simultaneous connection attempts by the *condor_schedd* when it is requesting claims from the *condor_startds*. The intention is to protect the *condor_schedd* from being overloaded by authentication operations. The default is 0, which indicates no limit.
- Added the new configuration variable `SEC_INVALIDATE_SESSIONS_VIA_TCP`, which defaults to `True`. Previously, attempts to use an invalid security session resulted in a UDP rather than a TCP response. In networks with different firewall rules for UDP and TCP, the filtering of the session invalidation messages was easily overlooked, since it would not typically happen during the initial vetting of the pool. If these packets were filtered out, then at the subsequent *condor_collector* restart, no daemons would be able to advertise themselves to the pool until their existing security sessions expired. The old behavior can be achieved by setting this configuration parameter to `False`.
- Added the new configuration variable `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION`. This is a special authentication mechanism designed to minimize overhead in the *condor_schedd* when communicating with the execute machine. Essentially, matchmaking results in a secret being shared between the *condor_schedd* and *condor_startd*, and this is used to establish a strong security session between the execute and submit daemons without going through the usual security negotiation protocol. This is especially important when operating at large scale over high latency networks, as in a glidein pool with one submit machine and thousands of execute machines on a network with 0.1 second round trip times. See 231 for details.
- Added configuration entry `GLEEXEC_JOB` which replaces the functionality previously encapsulated in `GLEEXEC_STARTER`. Using `GLEEXEC_JOB` enables privilege separation in Condor via `glxexec` in a manner much more consistent with how Condor's own privilege separation mechanism works. Specifically, the user identity switching will now occur between the *condor_starter* and the actual user job.
- Added configuration parameter `AMAZON_GAHP_WORKER_MAX_NUM` to specify a ceiling on the number of threads spawned on the submit machine to support jobs running on Amazon EC2. Defaults to 5.

Bugs Fixed:

- Includes bug fixes from Condor v7.0.5, including the security fixes. See section 8.5.
- Fixed a bug in the *condor_schedd* that would cause it to except if a crontab entry was incorrectly formatted.
- Fixed a bug in the CondorView server (collector) that caused it to except (crash) when it received a machine ClassAd without a valid state. It now logs this under level `D_ALWAYS` and ignores the ClassAd.
- Fixed a bug from Condor version 7.1.2 that would cause Condor daemons to start consuming a lot of cpu time after rare types of communication failures during security negotiation.

- Fixed a bug from Condor version 7.1.2 that in rare cases could cause Condor to fail to recognize when a call to `exec()` fails on Unix platforms.
- Fixed problems with configuration parameter `JOB_INHERITS_STARTER_ENVIRONMENT` when using PrivSep.
- Improved the deletion of Amazon EC2 jobs when the server is unreachable.
- Fixed problems with Condor parallel universe jobs when recovering from a reboot of the submit machine.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.1.2

Release Notes:

- None.

New Features:

- Added `formatTime()`, a built-in ClassAd function to create a formatted representation of the time. A detailed description of this function is available in section 4.1.1, which documents all of the available built-in ClassAd functions.
- Improved Condor's authentication handshake, so that daemons such as the *condor_schedd*, which initiate connections to other daemons, spend less time waiting for responses. Authentication over high latency networks is still rather expensive in Condor, so it still may be necessary to scale up by running more *condor_schedd* and *condor_collector* daemons than one would need for equivalent workloads on a low latency network. Additional improvements in this area are planned.

Configuration Variable Additions and Changes:

- None.

Bugs Fixed:

- Fixed a memory leak, introduced in Condor version 7.1.1, which caused the *condor_startd* daemon to grow without bound.
- Fixed a bug in *condor_dagman* that caused the user log file of the first node job in a DAG to get created with 0600 permissions, regardless of the user's umask. Note that this fix involved removing the **-condorlog** and **-storklog** command-line arguments from *condor_submit_dag* and *condor_dagman*.
- Fixed a problem from Condor version 7.1.1 that in some cases caused the *condor_starter* to stop sending updates about the job status or to send updates too frequently.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.1.1**Release Notes:**

- None.

New Features:

- Added a new feature to *condor_dagman* which caches the log lines emitted to the dagman.out file when in recovery mode and emits the cache as one call to the logging subsystem when the cache size limit is reached. Under NFS conditions, this prevents an open and close per line of the log and greatly improves performance. This feature is off by default and is controlled by `DAGMAN_DEBUG_CACHE_ENABLE`, which takes a boolean, and `DAGMAN_DEBUG_CACHE_SIZE`, which is an integer in bytes of how big the cache should be before flushing.
- Included some Windows example jobs (submit files and binaries).
- Added a new feature to the DAGMan language called splicing. Please read section 2.10.6 on page 80.
- The Prepare Job Hook can now modify the job ClassAd before execution. For a complete description of the new hook system, read section 4.4 on page 460.

- Condor now coerces the result of `$$([])` expressions within submit description files to strings. This means that submit files can do simple arithmetic. For example, you can describe a command-line argument as:
`arguments = $$([$(PROCESS)+100])`
and *condor_submit* will expand the argument to be the expected value.
- Condor daemons now periodically update the `ctime` of their log files, instead of the `mtime`, as they previously did. At start up, the daemons use this `ctime` to determine how long they may have been down.
- Added the capability to the *condor_startd* to allow it to power down machines based a user specified policy. See section 3.15 on 426 on Power Management for more details.
- *condor_off* now supports the **-peaceful** option for the *condor_schedd*, in addition to the existing support that already existed for the *condor_startd*. When peacefully shut down, the *condor_schedd* stops starting new jobs and waits for all running jobs to finish before exiting. The default shut down behavior is still **-graceful**, which checkpoints and stops all running standard universe jobs and gracefully disconnects from other types of jobs in the hopes of later restarting and reconnecting to them without any disturbance to the running job.
- The *condor_job_router* now supports deletion of attributes when transforming job ClassAds from vanilla to grid universe. It also behaves more deterministically when choosing from multiple possible routes. Rather than picking one at random, it uses a round-robin selection.
- *condor_dagman* now checks that its submit file was generated by a *condor_submit_dag* with the same version as *condor_dagman* itself. It is a fatal error for the versions to differ.

Configuration Variable Additions and Changes:

- Added `DAGMAN_DEBUG_CACHE_ENABLE` and `DAGMAN_DEBUG_CACHE_SIZE` which allow DAGMan to maintain a cache of log lines and write out the cache as one open/write/close sequence. `DAGMAN_DEBUG_CACHE_ENABLE` is a boolean which turns on the ability for caching and defaults to `False`. `DAGMAN_DEBUG_CACHE_SIZE` is a positive integer and represents the size of the cache in bytes and defaults to 5 Megabytes.
- The existing `BIND_ALL_INTERFACES` configuration variable now defaults to `True`.
- Added the `HIBERNATE` expression, which, when evaluated in the context of each slot, determines if a machine should enter a low power state. See page 193 for more information.
- Added the `HIBERNATE_CHECK_INTERVAL` configuration variable, which, if set to a non-zero value, enables the *condor_startd* to place the machine in a low power state based on the evaluation of the `HIBERNATE` expression. See page 193 for more information.
- The existing `VALID_SPOOL_FILES` configuration variable now automatically includes `SCHEDD.lock`, the lock file used for high availability *condor_schedd* fail over. Other high availability lock files are not currently included.

- Added the `SEC_DEFAULT_AUTHENTICATION_TIMEOUT` configuration variable, where the definition `DEFAULT` may be replaced by the usual list of contexts for security settings (for example, `CLIENT`, `READ`, and `WRITE`). This specifies the number of seconds that Condor should allow for the authentication of network connections to complete. Previously, GSI authentication was hard-coded to allow 5 minutes for authentication. Now it uses the same default as all other methods: 20 seconds.
- Added the `STARTER_UPDATE_INTERVAL_TIMESLICE` configuration variable, which specifies the highest fraction of time that the *condor_starter* should spend collecting monitoring information about the job, such as disk usage. It defaults to 0.1. If checking the disk usage of the job takes a long time, the *condor_starter* will monitor less frequently than specified by `STARTER_UPDATE_INTERVAL`.

Bugs Fixed:

- Fixed a bug introduced in 7.1.0 affecting configurations in which authentication of all communication between the *condor_shadow* and *condor_schedd* is required. This caused failure in the final update after the job had finished running. The result was that the job would return to the idle state to run again.
- Fixed a bug in Java universe where each slot would be told to potentially use all the memory on the machine. Now, each JVM receives the physical memory divided by the number of slots.
- On Windows, slot users would sometimes show up in the Windows Welcome Screen. This has now been resolved. The slot users need to be manually removed for this to take effect and the machine may need to be rebooted for the setting to be honored.
- Fixed a bug in the `ClassAd string()` function. The function now properly converts integers and floats to their string representation.
- The Windows Installer is now completely internationalized: it will no longer fail to install because of a missing "Users" group; instead, it will use the regionally appropriate group.
- Interoperability with Samba (as a PDC) has been improved. Condor uses a fast form of login during credential validation. Unfortunately, this login procedure fails under Samba, even if the credentials are valid. The new behavior is to attempt the fast login, and on failure, fall back to the slower form.
- Windows slot users no longer have the Batch Privilege added, nor does Condor first attempt a Batch login for slot users. This was causing permission problems on hardened versions of Windows, such as Windows Sever 2003, in that not interactive users lacked the permission to run batch files (via the *cmd.exe* tool). This affected any user submitting jobs that used batch files as the executable.
- If the `IWD` is not defined in a job classified ad that was either fetched by the *condor_startd* via job hooks, or pushed to the *condor_startd* via COD, the *condor_starter* no longer treats this as a fatal error, and instead uses the temporary job execution sandbox as the initial working directory.

- Made some fixes to the new-style rescue DAG feature:
 - *condor_submit_dag* no longer needs the **-force** flag if a rescue DAG will be run, even if the files generated by *condor_submit_dag* already exist.
 - *condor_submit_dag* with the **-force** flag now renames any existing new-style rescue DAG files, and therefore runs the original DAG.
- Fixed a problem that caused new-style rescue DAGs to fail when *condor_submit_dag* is invoked with the **-usedagdir** flag.

Known Bugs:

- None.

Additions and Changes to the Manual:

- The manual now contains Windows installation instructions for controlling the configuration for the **vm** universe.

Version 7.1.0

Release Notes:

- Upgrading to 7.1.0 from previous versions of Condor will make existing Standard Universe jobs that have already run fail to match to machines running Condor 7.1.0 unless the job previously ran on a machine using the Red Hat 5.0 release of Condor. This is because the value of the `CheckpointPlatform` attribute of the machine ClassAd has changed in order to better represent checkpoint compatibility. If this affects you, you can use *condor_qedit* to change the `LastCheckpointPlatform` attribute of existing Standard Universe jobs to match the new `CheckpointPlatform` advertised by the machine ClassAd where the job last ran.
- Condor no longer supports root configuration files (for example, `/etc/condor/condor_config.root`, `condor/condor_config.root`, and the file defined by the configuration variable `LOCAL_ROOT_CONFIG_FILE`). This feature was intended to give limited powers to a Unix administrator to configure some aspects of Condor without gaining root powers. However, given the flexibility of the configuration system, we decided that this was not practical. As long as Condor is started up as root, it should be clearly understood that whoever has the ability to edit the Condor configuration files can effectively run arbitrary programs as root.

New Features:

- In the past, Condor has always sent work to the execute machines by pushing jobs to the *condor_startd*, either from the *condor_schedd* or via *condor_cod*. As of version 7.1.0, The *condor_startd* now has the ability to pull work by fetching jobs via a system of plug-ins or hooks. Additional hooks are invoked by the *condor_starter* to help manage work (especially for fetched jobs, but the *condor_starter* hooks can be defined and invoked for other kinds of jobs as well). For a complete description of the new hook system, read section 4.4 on page 460.
- Added the capability to insert commands into the *.condor.sub* file produced by *condor_submit_dag* with the **-append** and **-insert_sub_file** command-line arguments to *condor_submit_dag* and the `DAGMAN_INSERT_SUB_FILE` configuration variable. See the *condor_submit_dag* manual page on page 825 and the configuration variable definition on page 228 for more information.
- For platforms running a Windows operating system, the Arch machine ClassAd attribute more correctly reflects the architectures supported. Instead of values "INTEL" and "UNDEFINED", the values will now be: "INTEL" for x86, "IA64" for Intel Itanium, and "X86_64" for both AMD and Intel 64-bit processors. These values are listed in the unnumbered subsection labeled Machine ClassAd Attributes on page 886.
- The Windows MSI installer now supports extended **vm** universe options. These new options include: the ability to set the networking type, how much memory the **vm** universe can use on a host, and the ability to set the version of *VMware* installed on the host.
- The *condor_status* and *condor_q* command line tools now have a version option which prints the version of those specific tools. This can be useful when multiple versions of Condor are installed on the same machine.
- The configuration variable `CONDOR_VIEW_HOST` may now contain a port number and may (if desired) refer to a *condor_collector* daemon running on the same host as the *condor_collector* that is forwarding ads. It is also now possible to use the forwarded ads for matchmaking purposes. For example, several collectors could forward ads to a single aggregating collector which a *condor_negotiator* then uses as its source of information for matchmaking.
- *condor_dagman* deals with rescue DAGs in a more sophisticated way; this is especially helpful for nested DAGs. See the rescue DAG subsection ?? of the *condor_dagman* manual section for more information.
- Additional logging details for unusual error cases to help identify problems.
- A new (optional) daemon named *condor_job_router* has been added, so far only on Unix. It may be configured to transform vanilla universe jobs into grid universe jobs, for example to send excess jobs to other sites via Condor-C or Condor-G. For details, see page 521.
- Previously, *condor_q -better-analyze* was supported on most but not all versions of Linux. It is now supported on all Unix platforms but not yet on Windows.

Configuration Variable Additions and Changes:

- Added new configuration variables `ALLOW_CLIENT` and `DENY_CLIENT` as client-side authorization controls. When using a mutual authentication method (such as GSI, SSL, or Kerberos), these variables allow the specification of which authenticated servers the Condor tools and daemons should trust when they form a connection to the server. Because of the addition of these variables, the GSI-specific, client-side authorization configuration variable `GSI_DAEMON_NAME` is retired, and no longer valid.
- Added the `DAGMAN_INSERT_SUB_FILE` variable, which allows a file of commands to be inserted into `.condor.sub` files generated by `condor_submit_dag`. See page 228 for more information.
- The semantics of `CLAIM_WORKLIFE` were previously not clearly defined before the start of the first job. A delay between the `condor_schedd` claiming a slot and the `condor_shadow` starting a job could be caused by the submit machine being very busy or by `JOB_START_DELAY`. Previously, such a delay would unpredictably result in the first job being rejected if `CLAIM_WORKLIFE` expired during that time. Now, `CLAIM_WORKLIFE` is defined to apply only after the first job has started. Therefore, setting it to zero has the effect of allowing exactly one job per claim to run. The default is still the special value -1, which places no limit on how long the slot may continue accepting new jobs from the `condor_schedd` that claimed it.
- Added the `DAGMAN_OLD_RESCUE` variable, which controls whether `condor_dagman` writes rescue DAGs in the old way. See page 228 for more information.
- Added the `DAGMAN_AUTO_RESCUE` variable, which controls whether `condor_dagman` automatically runs an existing rescue DAG. See page 228 for more information.
- Added the `DAGMAN_MAX_RESCUE_NUM` variable, which controls the maximum "new-style" rescue DAG number written or automatically run by `condor_dagman`. See page 228 for more information.

Bugs Fixed:

- The Condor Build ID is now printed by `condor_version` and placed in the logs for machines running a Windows operating system.
- `condor_quill` and the `condor_dbmsd` correctly register themselves with the Windows firewall.
- `condor_submit_dag` now avoids possibly running off the end of the argument list if an argument requiring a value does not have one.
- The `condor_submit_dag -debug` argument now must be specified with at least `-de` to avoid conflict with the `-dagman` argument.
- Added missing information about the `-config` argument to `condor_submit_dag`'s usage message.
- `condor_dagman` no longer considers duplicate edges in a DAG a fatal error (it is now a warning).

Known Bugs:

- No hook is invoked if a fetched job does not contain enough data to be spawned by a *condor_starter* or if other errors prevent the job from being run after the *condor_startd* agrees to accept the work. This limitation will be addressed in a future version of Condor, most likely via the addition of a new hook invoked whenever the *condor_starter* fails to spawn a job. For more information about the new hook system included in Condor version 7.1.0, read section 4.4 on page 460.

Additions and Changes to the Manual:

- Added "WINNT60" for the Vista operating system to the documented list of possible values for the machine ClassAd attribute OpSys.

8.5 Stable Release Series 7.0

This is a stable release series of Condor. It is based on the 6.9 development series. All new features added or bugs fixed in the 6.9 series are available in the 7.0 series. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 7.0.x releases. New features will be added in the 7.1.x development series.

On backwards compatibility: we believe that Condor 7.0.x and 6.8.x are wire-compatible, and can be freely mixed between computers in a Condor pool. However, we do not regularly test this compatibility and cannot guarantee it, so we recommend using a single release of Condor when possible. Please note that although you can mix Condor 7.0.x and 6.8.x in a pool, you cannot mix them on a single computer. That is, a *condor_master* daemon running 6.8.x cannot run Condor daemons from version 7.0.x, or vice-versa.

The details of each version are described below.

Version 7.0.6

Release Notes:

- None.

New Features:

- None.

Configuration Variable Additions and Changes:

- None.

Bugs Fixed:

- In some rare cases, the *condor_startd* failed to fully preempt jobs. The job itself was killed, but the *condor_starter* process watching over it would not be killed. The slot would then stay in the Preempting state indefinitely.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.0.5**Release Notes:**

This release contains many bug fixes and some improvements to error handling of Local Universe jobs. Note that some of the bug fixes are security-related; therefore, we recommend sites either upgrade Condor, or restrict permissions on who is allowed to submit Condor jobs to trusted users. Bug fixes that are security related are clearly marked in the Bugs Fixed section below along with a description of the potential security impact. The Condor Project believes in the full disclosure of information, and therefore complete vulnerability details can be found at <http://www.cs.wisc.edu/condor/security/>. However, in order to give an adequate upgrade window for production installations, we will delay posting the full vulnerability details fixed in this release for 30 days (until the week of November 3rd 2008).

New Features:

- Local universe jobs now go on hold for the same specific reasons that vanilla jobs may go on hold. Examples are missing input or executable files. Previously, when local universe jobs failed in this manner, the jobs returned to the idle state in the job queue, repetitively attempting to run, and failing over and over until the job is removed.
- Local universe jobs now have the ClassAd attribute `NumShadowStarts`. Although local universe jobs do not have a *condor_shadow* process, this attribute is introduced to keep management of local universe as similar to vanilla universe as possible. For local universe jobs, this attribute is identical to the attribute `JobRunCount`, which indicates how many times a local *condor_starter* process has been created to run the job.

Configuration Variable Additions and Changes:

- None.

Bugs Fixed:

- *Security Item:* A flaw was found and fixed in the way Condor processes user submitted jobs. It was possible for a user who had permissions to submit jobs into Condor to do so in a way that could cause that job to run as any other non-root user. We have not had any reported incidents exploiting this flaw. (CVE-2008-3826)
- *Security Item:* A stack-based buffer overflow flaw was found and fixed in the *condor_schedd* daemon. A user who had permissions to submit a job could do so in a manner that could cause the *condor_schedd* to crash, or potentially, execute arbitrary code on the submit machine with the *condor_schedd*'s identity. We are not aware of any known exploits for this flaw. We have not had any reported incidents exploiting this flaw. (CVE-2008-3828)
- *Security Item:* A denial-of-service flaw was found and fixed in the *condor_schedd* daemon. A user who had permissions to submit a job could have done so in a manner that would cause *condor_schedd* to crash. We have not had any reported incidents exploiting this flaw. (CVE-2008-3829)
- *Security Item:* A flaw was found and fixed in the way Condor processes allow and deny net masks for access control. If Condor's configuration file contained overlapping net masks in the allow or deny rules, it could have caused those rules to be ignored, potentially allowing unintended access to users in Condor's deny authorization lists. We have not had any reported incidents exploiting this flaw. (CVE-2008-3830)
- Fixed a segmentation fault bug with *condor_submit -dump* when `universe=grid` or `x509userproxy=<anything>`.
- Fixed a stack overflow bug in the *condor_negotiator* daemon.
- Fixed *condor_submit -dump* such that it would function with the standard universe.
- Fixed a memory leak in the *condor_startd*, which occurred during the handling of a *condor_reconfig* command.
- When the configuration variable `NEGOTIATOR_CONSIDER_PREEMPTION` is defined to be `False`, this no longer results in machines in the Owner state being ignored during matchmaking. Previously, even if `START` was `True`, machines in the Owner state were disregarded.
- Setting `JobLeaseDuration` to be less than 15 minutes caused the *condor_schedd* daemon to abort and restart the next time a *condor_reconfig* command was executed. The error message in the *condor_schedd* log appeared as:

`ALIVE_INTERVAL` in the condor configuration is too high (300).

- Fixed a slow memory leak affecting the *condor_startd*, *condor_schedd*, and *condor_collector* daemons. This leak would probably require many months of continuous operation before causing noticeable problems.
- Fixed a bug that caused a *condor_schedd* daemon crash. The crash occurred during a fast shut down of the *condor_schedd* daemon as it dealt with local universe jobs or with any job that required reconnection when the *condor_schedd* daemon started up.
- Local and scheduler universe jobs were failing to increment the `JobRunCount` attribute in the job `ClassAd` when an attempt to run the job was made. This problem was introduced in 6.9.5.
- Some rare types of failures during file transfer caused the Condor daemon conducting the transfer to hang indefinitely. For example, if the file transfer process created by the *condor_schedd* was killed by an administrator or crashed due to an internal error, the *condor_schedd* would become unresponsive.
- GCB was updated, fixing minor bugs with GCB temporary files (typically the file(s) `/tmp/gcb-inherit-*`). These bugs did not impact GCB functionality. Earlier versions would leave temporary files behind. Temporary files would have permissions of 000. With the fix, under normal operations the files should be deleted, and the `condor` user should have read and write access to the files.
- Evaluation of the configuration variable `STARTD_AD_REEVAL_EXPR` did not work for many types of expressions. The problem resulted in the following message in the *condor_negotiator* daemon log:

```
Can't evaluate STARTD_AD_REEVAL_EXPR ...
```

- Reconnecting to parallel universe jobs after a restart of the *condor_schedd* daemon, would sometimes fail. The failure was caused by the *condor_shadow* trying to connect to the address of the previous instance of the *condor_schedd* rather than the address of the current instance.
- Made the *condor_gridmanager* less aggressive in forwarding refreshed proxies for gt2 grid universe jobs. Now, the refreshed proxy will not be forwarded until the old proxy has less than six hours of life until expiration.
- Fixed a bug in the *condor_gridmanager* that could result in job status updates from the Grid Monitor to be ignored.
- The Grid Monitor no longer changes the last-modified time of GRAM state files whose job's status is `FAILED`. This should make it easier for file cleaners to remove the the GRAM state files of old, abandoned jobs.
- Fixed a problem that could cause flocked jobs to fail due to authorization errors in the *condor_starter*. Such failures were more likely to occur for long-running jobs or if the *condor_schedd* were issued a full reconfig during the job's execution.
- Fixed a *condor_gridmanager* crash on Windows. This crash only appeared if `GRIDMANAGER_DEBUG` were set to a higher level than the default.

- In PrivSep mode, a job would previously fail if it created a symlink in its sandbox pointing to a file owned by a UID other than that used to run the job. This behavior has been fixed.

Known Bugs:

- None.

Additions and Changes to the Manual:

- Descriptions of previously undocumented Condor Perl module subroutines have been added to the manual. See section ??.

Version 7.0.4

Release Notes:

- This release fixes a problem causing possible incorrect handling of wild cards in authorization lists. Examples of the configuration variables that specify authorization lists are

```
ALLOW_WRITE
DENY_WRITE
HOSTALLOW_WRITE
HOSTDENY_WRITE
```

If a configuration variable uses the asterisk character (*) in configuration variables that specify the authorization policy, it is advisable to upgrade. This is especially true for the use of wild cards in any DENY list, since this problem could result in access being allowed, when it should have been denied. This issue affects all previous versions of Condor.

- The default daemon-to-daemon security session duration has been changed from 100 days to 1 day. This should reduce memory usage in the *condor_collector* in pools with short-lived *condor_startds* (e.g. glidein pools or pools whose machines are rebooted every night).

New Features:

- Added functionality to periodically update timestamps on lock files. This prevents administrative programs from deleting in-use lock files and causing undefined behavior.
- When the configuration variable SCHEDD_NAME ends in the @ symbol, Condor will no longer append the fully qualified host name to the value. This makes it possible to configure a high availability job queue that works with the remote submission of jobs.

Configuration Variable Additions and Changes:

- Added configuration variable: `LOCK_FILE_UPDATE_INTERVAL` . Please see page 165 for a complete description.
- Changed the default value of configuration variable `SEC_DEFAULT_SESSION_DURATION` from 8640000 seconds (100 days) to 86400 seconds (1 day).

Bugs Fixed:

- Fixed a bug in the *condor_c-gahp* that caused it to fail repeatedly on Windows, if more than two Condor-C jobs were submitted at the same time.
- Fixed a problem that caused the *condor_collector*'s memory usage to increase dramatically, if *condor_findhost* was run repeatedly.
- Fixed a bug where Windows jobs suspended by Condor would never be continued, despite log files indicating successful continuation. This problem has existed since the 6.9.2 release of Condor.
- Fixed a problem that could cause *condor_dagman* to core dump if straced, especially if the *dagman.out* file is on a shared file system.
- Fixed a problem introduced in 7.0.1 that could cause the *condor_schedd* daemon to crash when starting parallel or MPI universe jobs. In some cases, the problem would result in the following log message:

```
ERROR ``Assertion ERROR on (mrec->request_claim_sock == sock)'' \
  at line 1361 in file dedicated_scheduler.C
```

- The *condor_procd* daemon now periodically updates the timestamps on the named pipe file system objects that it uses for communication. This prevents these objects from being cleaned up by programs like *tmpwatch*, which would result in Condor daemon exceptions.
- Fixed a problem introduced in Condor 7.0.2 that would cause daemons to fail on start up on Windows 2000.
- Fixed a problem where standard universe jobs would fail to start when using PrivSep, if the `PROCD_ADDRESS` configuration variable was not defined.
- If the X509 proxy of a vanilla universe job has been refreshed, the updated file will no longer be returned when the job completes.
- If ClassAd attributes `StreamOut` or `StreamErr` are missing from the job ClassAd of a grid universe job, the default value for these attributes is now `False`.

Known Bugs:

- A bug in 7.0.4 affects jobs using Condor file transfer on submit machines that are configured to deny write access from execute machines. The result is that output from jobs may fail to be copied back to the submit machine. The problem may or may not affect jobs that run for less than eight hours, but it definitely will affect jobs that run for more than eight hours. An example of a configuration vulnerable to this problem is one where DAEMON level access is allowed to all execute nodes but WRITE level access is not. When the problem happens, the *condor_shadow* log will contain a line like the following:

```
DaemonCore: PERMISSION DENIED to unknown user from host ...  
for command 61001 (FILETRANS_DOWNLOAD), access level WRITE
```

The workaround for this problem is to allow WRITE access from the execute nodes. If the existing configuration requires WRITE access to be authenticated, then simply add WRITE access by the authenticated condor identities associated with all execute nodes. If WRITE access is not currently required to be authenticated, then allow unauthenticated WRITE access from all worker nodes. Note that this does *not* imply that execute nodes will be able to modify the job queue without authenticating. Remote commands that modify the job queue (for example, *condor_submit* or *condor_qedit*) always require that the user be authenticated, no matter what configuration options are used; if no method of remote authentication can succeed in the pool for WRITE operations, then commands that modify the job queue can only run on the submit machine.

Additions and Changes to the Manual:

- None.

Version 7.0.3

Release Notes:

- This is a bug fix release. A bug in Condor version 7.0.2 sometimes caused the *condor_schedd* to become unresponsive for 20 seconds when starting the *condor_shadow* to run a job. Therefore, anyone running 7.0.2 is strongly encouraged to upgrade.

New Features:

- None.

Configuration Variable Additions and Changes:

- The configuration variable `VALID_SPOOL_FILES` now automatically includes `SCHEDD.lock`, the lock file used for high availability *condor_schedd* fail over. Other high availability lock files are not currently included.

Bugs Fixed:

- Fixed a problem sometimes causing minutes or more of lag between the time of job suspension or unsuspension and the corresponding entries in the job user log.
- Fixed a problem in *condor_q* **-better-analyze** handling requirements expressions containing the expression `= != UNDEFINED`.
- Configuration variable `GRIDMANAGER_GAHP_CALL_TIMEOUT` is now recognized for non-dugrid grid universe jobs.
- Fixed a bug that could cause the *condor_schedd* daemon to abort and restart some time after a graceful restart, when jobs to which the *condor_schedd* daemon reconnected were preempted.
- Fixed a bug causing failure to reconnect to jobs which use `$$([expression])` in their ClassAds. The jobs would go on hold with the hold reason: "Cannot expand `$$([expression])`."
- Fixed a bug in Condor version 7.0.2 that sometimes caused the *condor_schedd* daemon to become unresponsive for 20 seconds when starting the *condor_shadow* daemon to run a job.

Known Bugs:

- None.

Additions and Changes to the Manual:

- See section 4.5.1 for documentation on finding the port number the *condor_schedd* daemon is listening on for use with the web service API.

Version 7.0.2**Release Notes:**

- On Unix, Condor no longer requires its `EXECUTE` directory to be world-writable, as long as it is not on a root-squashed NFS mount and is owned by the user given in the `CONDOR_IDS` setting (or by Condor's real UID, if not started as `root`). Condor will automatically remove world-writability from existing `EXECUTE` directories where possible. Note: The `EXECUTE` directory has never been required to be world-writable on Windows.
- With this release, a binary package for IA64 SUSE Linux Enterprise 8 will no longer be made available.

New Features:

- A clipped port to FreeBSD 7.0 x86 and x86_64 is available, but at this time, it is not available for download as a binary package.
- Previously, *condor_q -better-analyze* was supported on most but not all versions of Linux. It is now supported on all Unix platforms, but not yet on Windows platforms.

Configuration Variable Additions and Changes:

- The new configuration variable `GRIDMANAGER_MAX_WS_DESTROY_PER_RESOURCE` limits the number of simultaneous WS destroy commands issued to a given server for grid universe jobs of type `gt4`. The default value is 5.

Bugs Fixed:

- Fixed a bug in the standard universe where if a Linux machine was configured to use the Network Service Cache Daemon (`nscd`), taking a checkpoint would be deferred indefinitely.
- Fixed a bug that caused the Quill daemon to crash.
- Fixed bug that prevented Quill, when running on a Windows host, from successfully updating the database.
- Fixed a bug that prevented Quill's *condor_dbmsd* daemon from proper shutting down upon request when running on Windows platforms.
- Fixed a bug that caused Stork to be completely broken.
- As a back port from Condor versions 7.1, the Windows Installer is now completely internationalized: it will no longer fail to install because of a missing "Users" group; instead, it will use the regionally appropriate group.
- As a back port from Condor versions 7.1, interoperability with Samba (as a PDC) has been improved. Condor uses a fast form of login during credential validation. Unfortunately, this login procedure fails under Samba, even if the credentials are valid. The new behavior is to attempt the fast login, and on failure, fall back to the slower form.
- As a back port from Condor versions 7.1, Windows slot users no longer have the Batch Privilege added, nor does Condor first attempt a Batch login for slot users. This was causing permission problems on hardened versions of Windows, such as Windows Server 2003, in that not interactive users lacked the permission to run batch files (via the *cmd.exe* tool). This affected any user submitting jobs that used batch files as the executable.
- Fixed a bug that could sometimes cause the *condor_schedd* to either EXCEPT or crash shortly after a user issues a *condor_rm* command with the **-forcex** option.
- *condor_history* in a Quill environment, when given the **-constraint** option, would ignore attributes from the vertical schema. This has been fixed.

- In Unix, when started as `root`, the *condor_master* now changes the effective user id back to `root` (instead of `condor`) when restarting itself. This occurs for example due to the command *condor_restart*. This makes no difference unless the *condor_master* is wrapped with a script, and the script expects to be run as `root` not only on initial start up, but on restart as well.
- The dedicated scheduler would sometimes take two negotiation cycles to acquire all the machines it needed to run a job. This has been now fixed.
- *condor_dagman* no longer prints "Argument added" and "Retry Abort Value" diagnostic messages at the default verbosity, to reduce the size of the `dagman.out` file and the start up time for very large DAGs.
- *condor_dagman* now prints a few fatal parse errors at lower verbosity settings than it did previously.
- *condor_preen* no longer deletes *MyProxy* password files in the Condor spool directory.
- When using TCP updates (UDP updates are the default), the *condor_collector* would sometimes freeze for 20 seconds when receiving an invalidation notice. The notice is received when Condor is being turned off on a machine in the pool.
- Fixed a case in which the *condor_schedd*'s job queue log file could get corrupted when encountering errors writing to the disk such as 'out of space'. This type of corruption was detected by the *condor_schedd* the next time it restarted and read the file to restore the job queue, so you would only have been affected by this problem if your *condor_schedd* refused to start up until you fixed or removed the job queue log file. This bug has existed in all versions of Condor, but it became more likely to occur in 6.9.4.
- The configuration setting `JAVA` may now contain spaces. Previously, this did not work.
- Fixed a problem that caused occasional failure to detect hung Condor daemons.
- Fixed a file descriptor leak in the negotiator. The leak happened whenever the negotiator failed to initiate the `NEGOTIATE` command to a *condor_schedd*, for example if security negotiation failed with the *condor_schedd*. Under Unix, this would eventually cause the *condor_negotiator* to run out of file descriptors, exit, and restart. This bug affected all previous versions of Condor.
- Fixed several bugs in the user log reader that caused it to generate an invalid persisted state if no events had been read in. When read back in, this persisted state would cause the reader to segfault during initialization.
- Fixed a bug causing communication problems if different portions of a Condor pool were configured with different values of `SEC_DEFAULT_SESSION_DURATION`. This bug affects all previous versions of Condor. The client side of the connection was always using its own security session duration, even if the server's duration was shorter. Among other potential problems, this was observed to cause file transfer failures when the starter was configured with a longer session duration than the shadow.
- Fixed a bug in the user log writer that was causing the writing of events to the global event log fail in some conditions.

- In the grid universe, submission of nordugrid jobs is now properly throttled by configuration parameters `GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE` and `GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE`.
- The NorduGrid GAHP server can now properly extract job execution information from newer NorduGrid servers. Previously, the GAHP could crash when talking to newer servers.
- Fixed a bug that caused *condor_config_val* `-set` or `-rset` to fail if security negotiation was turned off. This happens, for example, if `SEC_DEFAULT_NEGOTIATION = NEVER`. This bug was introduced in Condor 7.0.0.
- Fixed a bug that could cause incorrect IP addresses to be advertised when the *condor_collector* was on a multi-homed host.
- Fixed a problem where unexpected ownership and permissions on files inside a job's working directory could cause the *condor_starter* to EXCEPT.
- Improved the speed at which the *condor_startd* can handle claim requests, particularly when the *condor_startd* manages a large number of slots.
- Fixed an error in the way the *condor_procd* calculates image size for jobs that involve multiple processes. Previously the maximum image size for any single process was being used. Now the image size sum across all processes is used.
- The *condor_procd* no longer truncates its log file on start up. Enabling a log file for the *condor_procd* is only recommended for debugging, since it is not rotated to conserve disk space.
- Fixed a problem present in Condor 7.0.1 and 7.1.0 where the *condor_startd* will crash upon deactivating or releasing a COD claim.
- Condor on Windows can now correctly handle job image size when processes are created that allocate more than 2GB of address space.
- The `JOB_INHERITS_STARTER_ENVIRONMENT` setting now works when the `GLEEXEC_STARTER` feature is in use.
- Fixed a problem causing *condor_schedd* to perform poorly when handling large job queues in which there are any idle local or scheduler universe jobs (for example, Condor cron jobs).
- Sped up *condor_schedd* graceful shutdown when disconnecting from running jobs that have job leases. Previously, it would only disconnect from one such job at a time, so if there were a lot of jobs running, *condor_schedd* could take so long to shut down that job leases expire before it has a chance to restart and reconnect to the jobs.
- Fixed a bug that could cause incorrect IP addresses to be advertised when the *condor_collector* was on a multi-homed host.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 7.0.1

Release Notes:

- Fixed a bug in Condor's authorization policy reader. The bug affects cases where the policy (ALLOW/DENY and HOSTALLOW/HOSTDENY settings) mixes host-based authorizations with authorizations that refer to the authenticated user name. In some cases, this bug would result in host-based settings not being applied to authenticated users.

New Features:

- Support for Backfill Jobs is now available on Windows platforms. For more information on this, please see section 3.12.9 on page 418.
- Condor has been ported to Red Hat Enterprise Linux 5.0 running on the 32-bit x86 architecture and on the 64-bit x86_64 architecture.
- The command **email_attributes** in a job submit description file defines a set of job ClassAd attributes whose values should be included in the e-mail notification of job completion.
- The configuration variable `CONDOR_VIEW_HOST` may now contain a port number, and may refer to a *condor_collector* daemon running on the same host as the *condor_collector* that is forwarding ClassAds. It is also now possible to use the forwarded ClassAds for matchmaking purposes. For example, several *condor_collector* daemons could forward ClassAds to a single aggregating *condor_collector* daemon which a *condor_negotiator* then uses as its source of information for matchmaking.
- *condor_configure* and *condor_install* now detect missing shared libraries (such as `libstdc++.so.5` on Linux), and print messages and exit if missing libraries are detected. The new command line option **-ignore-missing-libs** causes it not to exit after the messages have been printed, and to proceed with the installation.
- Added a **-force** command line option to *condor_configure* (and *condor_install*) which will turn on **-overwrite** and **-ignore-missing-libs**.
- *condor_configure* now writes simple sh and csh shell scripts which can be sourced by their respective shells to set the user's `PATH` and `CONDOR_CONFIG` environment variables. By default, these are created in the root of the Condor installation, but this can be changed via the **-env-scripts-dir** command line option. Also, the creation of these scripts can be disabled with the **-no-env-scripts** command line option.

Configuration Variable Additions and Changes:

- The new configuration variables `PREEMPTION_REQUIREMENTS_STABLE` and `PREEMPTION_RANK_STABLE` are boolean values to identify whether or not attributes used within the definition of `PREEMPTION_REQUIREMENTS` and `PREEMPTION_RANK` remain unchanged during a negotiation cycle. See section 3.3.17 on page 213 for complete definitions.
- The configuration variable `STARTER_UPLOAD_TIMEOUT` changed its default value to 300 seconds.
- The new configuration variable `CKPT_PROBE` specifies an internal to Condor executable which determines information about how a process is laid out in memory, in addition to other information. This executable is not yet available on Windows platforms.
- The new configuration variable `CKPT_SERVER_CHECK_PARENT_INTERVAL` sets an interval of time between checks by the checkpoint server to see if its parent, the *condor_master* daemon, has gone away unexpectedly. The checkpoint server shuts itself down if this happens. The default interval for checking is 120 seconds. Setting this parameter to 0 disables the check.

Bugs Fixed:

- Upgrade from PCRE v5.0 to PCRE v7.6, due to security vulnerabilities found in PCRE v5.0.
- Fixed file descriptor leak in the *condor_schedd* when using the SOAP interface.
- Fixed a bug that primarily affected pools with `MaxJobRetirementTime` (0 by default) set larger than `REQUEST_CLAIM_TIMEOUT` (30 minutes by default). Since 6.9.3, when the *condor_schedd* timed out requesting a claim to a slot, the *condor_startd* was not made aware of the canceled request. This resulted in some wasted time (up to `ALIVE_INTERVAL`) in which the *condor_startd* would wait for a job to run.
- A problem with *condor_history* in a Quill environment incorrectly interpreting the **-name** option has been fixed.
- A memory leak that prevented *condor_load_history* from running with large history files has been fixed.
- A bug in *condor_history* when running in a quill environment has been fixed. This bug would cause the command to crash in some situations.
- The job ClassAd attribute `EmailAttributes` now works for grid universe jobs.
- On 32-bit Linux platforms, the job queue database file may now exceed 2GB. Previously, the *condor_schedd* would halt with an error when trying to write past the 2GB mark.
- On 32-bit Linux platforms, *condor_history* can now read from history files larger than 2GB *except* when using the **-backwards** option.

- Local universe jobs are now scheduled to run more promptly. Previously, new local universe jobs would sometimes take up to `SCHEDD_INTERVAL` (default 5 minutes) to be considered for running.
- The memory usage of the *condor_collector* used to grow over time if daemons with new names kept joining and then leaving the pool (for example, in a *Glidein* pool). This was due to statistics on dropped updates that accumulated for all daemons that ever advertised themselves to the *condor_collector*. These statistics are now periodically purged of information about daemons which have not reported in a long time. How long is controlled by `COLLECTOR_STATS_SWEEP`, which defaults to 2 days.
- Condor daemons would die when trying to send ClassAd advertisements to a host name that could not be resolved by DNS.
- Since 6.9.5, file transfer errors for vanilla, java, or parallel jobs would sometimes not result in the job going on hold as it should. This was most likely for very small files that failed to be written for some reason.
- The `ImageSize` reported for jobs on AIX was too big by a factor of 1024.
- Since 6.9.5, *condor_glidein* failed in the set up stage, due to the change in syntax of quoting rules in the Condor submit description file for `gt2` argument strings.
- Fixed a bug in the *condor_gridmanager* that could prevent refreshed X509 proxies from being forwarded to the remote machine for grid universe jobs of type `gt4`.
- Fixed a bug in Condor's authorization policy reader. The bug affects cases where the policy (`ALLOW/DENY` and `HOSTALLOW/HOSTDENY` settings) mixes host-based authorizations with authorizations that refer to the authenticated user name. In some cases, this bug would result in host-based settings not being applied to authenticated users.
- Fixed a bug in *condor_history* which causes a crash when *condor_quill* is enabled.
- Fixed a problem affecting the GSI and SSL authentication methods. When these methods successfully authenticated the user but failed to find a mapping of the X509 name to a condor user id, they were setting the authenticated name to `gsi` and `ssl` respectively. However, these names contain no domain, so they could not be referred to in the authorization policy. Now these anonymous mappings are `gsi@unmappeduser` and `ssl@unmappeduser`. Therefore, configuration to deny access by users who are not explicitly mapped in the map file appears as:

```
DENY_READ = *@unmappeduser
DENY_WRITE = *@unmappeduser
```

Known Bugs:

- When using *condor_compile* with the RHEL5 x86 port of Condor to produce a standard universe executable, one will see a warning message about how linking with dynamic libraries is not portable. This warning is erroneous and should be ignored. It will be fixed in a future version of Condor.

Additions and Changes to the Manual:

- The existing configuration variables `SYSTEM_PERIODIC_HOLD`, `SYSTEM_PERIODIC_RELEASE`, and `SYSTEM_PERIODIC_REMOVE` have documented definitions. See section 3.3.11 for definitions.
- A manual page for *condor_load_history* has been added.

Version 7.0.0

Release Notes:

- PVM support has been dropped.
- The time zone for the *PostgreSQL* 8.2 database used with Quill on Windows machines must be explicitly set to use an abbreviation. This Windows environment variable is `TZ`. Proper abbreviations for the value of this variable may be found within the *PostgreSQL* installation in a file, `share/timzonesets/<continent>.txt`, where `<continent>` is replaced by the continent of the desired time zone.

New Features:

- The Windows MSI installer now supports VM Universe.
- Eliminated the “tarball in a tarball” in our distribution. The contents of `release.tar` from the distribution tarball (for example, `condor-6.9.6-linux-x86-centos45-dynamic.tar.gz`) is now included in the distribution tarball.
- Updated *condor_configure* to match the above change. The `--install` option now takes a directory path as its parameter, for example `--install=/path/to/release`. It previously took the path to the `release.tar` tarball.
- Added *condor_install*, which is a symlink to *condor_configure*. Invoking

```
condor_install
```

is identical to running

```
condor_configure --install=.
```

- Added the option `--prefix=dir` to *condor_configure* and *condor_install*. This is an alias for `--install-dir=dir`.

- Added the option **-backup** option to *condor_configure* and *condor_install*. This option re-names the target *sbin* directory, if the *condor_master* daemon exits while in the target *sbin* directory. Previous versions of *condor_configure* did this by default.
- Changed the default behavior of *condor_install* to exit with a warning if the target *sbin* directory exists, the *condor_master* daemon is in the *sbin* directory, and neither the **-backup** nor **-overwrite** options are specified. This prevents *condor_install* from improperly moving an *sbin* directory out of the way. For example,

```
condor_install --prefix=/usr
```

will not move */usr/sbin* out of the way unless the **-backup** option is also specified.

- Updated the usage summary of *condor_configure* and *condor_install* to be much more readable.

Configuration Variable Additions and Changes:

- The new configuration variable `DEAD_COLLECTOR_MAX_AVOIDANCE_TIME` defines the maximum time in seconds that a daemon will fail over from a primary *condor_collector* to a secondary *condor_collector*. See section 3.3.3 on page 156 for a complete definition.

Bugs Fixed:

- Fixed a memory leak in the *condor_procd* daemon on Windows.
- Fixed a problem that could cause Condor daemons to crash if a failure occurred when communicating with the *condor_procd*.
- Fixed a couple of problems that were preventing the *condor_startd* from properly removing per-job directories when running with PrivSep.
- The *condor_startd* will no longer fail to initialize, claiming the `EXECUTE` directory has improper permissions, when PrivSep is enabled.
- Look ups of ClassAd attribute `CurrentTime` are now case-insensitive, just like all other attributes.
- Fixed problems causing the following error message in the log file:

```
ERROR: receiving new UDP message but found a short message still waiting to be closed (consumed=1).
```

- The existence of the executable given in the submit file is now enforced (when transferring the executable and not using VM universe).
- The copy of *condor_dagman* that ships with Condor is now automatically added to the list of trusted programs in the Windows Firewall.

- Removed **remove_kill_sig** from the submission file generated by *condor_submit_dag* on Windows.
- Fixed the algorithm in the *condor_negotiator* daemon, which with large numbers of machine ClassAds (for example, 10,000) was causing long delays at the beginning of each negotiation cycle.
- Use of MAX_CONCURRENT_UPLOADS was resulting in a connection attempt from the *condor_shadow* to the *condor_schedd* with a fixed 10 second timeout, which is sometimes too small. This timeout has been increased to be the same as other connection timeouts between the *condor_shadow* and the *condor_schedd*, and it now respects SHADOW_TIMEOUT_MULTIPLIER, so it can be adjusted if necessary.
- Fixed a problem with MAX_CONCURRENT_UPLOADS and MAX_CONCURRENT_DOWNLOADS, which was sometimes allowing more than the configured number of concurrent transfers to happen.
- Fixed a bug in the *condor_schedd* that could cause it to crash due to file descriptor exhaustion when trying to send messages to hundreds of *condor_startds* simultaneously.
- Fixed a 6.9.4 bug in the *condor_startd* that would cause it to crash when a BOINC backfill job exited.
- Since 6.9.4, when using glExec, configuring SLOTx_EXECUTE would cause *condor_starter* to fail when starting the job.
- Fixed a bug from 6.9.5 which caused authentication failure for the pool password authentication method.
- Fixed a bug that caused Condor daemons to crash when encountering some types of invalid ClassAd expressions.
- Fixed a bug under Linux that could cause multi-process daemons lacking a log lock file to crash while rotating logs that have reached their maximum configured size.
- Fixed a bug under Windows that sometimes caused connection attempts between Condor daemons to fail with Windows error number 10056.
- Fixed a problem in which there are multiple *condor_collector* daemons in a pool for fault tolerance. If the primary *condor_collector* failed, the *condor_negotiator* would fail over to the secondary *condor_collector* indefinitely (or until the secondary *condor_collector* also failed or the administrator ran *condor_reconfig*). This was a problem for users flocking jobs to the pool, because flocking currently only works with the primary *condor_collector*. Now, the *condor_negotiator* will fail over for a restricted amount of time, up to DEAD_COLLECTOR_MAX_AVOIDANCE_TIME seconds. The default is one hour, but if querying the dead primary *condor_collector* takes very little time to fail, the *condor_negotiator* may retry more frequently in order to remain responsive to flocked users.
- Fixed a problem preventing the use of *condor_q -analyze* with the **-pool** option.

- Fixed a problem in the *condor_negotiator* in which machines go unassigned when user priorities result in the machines getting split into shares that are rounded down to 0. For example if there are 10 machines and 100 equal priority submitters, then each submitter was getting 0.1 machines, which got rounded down to 0, so no machines were assigned to anybody. The message in the *condor_negotiator* log in this case was this:

```
Over submitter resource limit (0) ... only consider startd ranks
```

- Fixed a problem introduced in 6.9.3 that would cause daemons to run out of file descriptors if they create sub-processes and are configured to use a lock file for the debug log.
- Standard universe jobs now work properly when using PrivSep.
- Fixed problem with PrivSep mode where a job that dumps core would not get the core file transferred back to the submit host if the **transfer_output_files** submit option were used.
- Fixed a bug that caused the *condor_starter* to crash if a job called *condor_chirp* with the `get_job_attr` option.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

8.6 Development Release Series 6.9

This is the development release series of Condor. The details of each version are described below.

Version 6.9.5

Release Notes:

- There are some known vulnerabilities in the Virtual Machine (VM) universe support included in this release that can allow VM universe jobs to read arbitrary files on the host machine. VM universe is disabled by default, so this potential vulnerability does not affect most sites. However, VM universe should not be enabled unless the job policy only allows users that you trust completely to run jobs on the machine.
- Condor is now licensed under the terms of the Apache License version 2.0.

- Dropped support for the following platforms:
 - Red Hat Linux 7.x systems on the x86 processor.
 - Digital Unix systems on the Alpha processor.
 - Yellow Dog Linux 3.0 systems on the PPC processor.
 - MacOS 10.3 systems on the PPC processor.

These ports are still supported in the 6.8 series of Condor.

- Dropped support for OGSA GRAM (grid-type gt3) in the grid universe. This version of GRAM is not included in recent versions of the Globus Toolkit. This does not affect Condor's support for pre-WS GRAM (grid-type gt2) or WS GRAM (grid-type gt4).
- The suggested configuration value for `SHADOW_RENICE_INCREMENT` has been changed from 10 to 0. If using the value 10 in an existing configuration file, we recommend changing it. This improves the performance of Condor on busy submit nodes where other processes would cause low priority *condor_shadow* daemons to become starved for CPU time.
- For grid-type gt2 grid universe jobs, job arguments are now handled as they are for all other job types. Previously, the user would have to manually escape characters that had special meaning to GRAM's RSL language.
- *condor_version* now includes a specific build identification number for official builds of Condor.

New Features:

- *condor_q*, when Quill is enabled, now displays the last time Quill updated the database. This permits seeing how fresh the database information is.
- *condor_history*, when Quill is enabled, will now query the database for historical items even when the **-constraint** option is given. Previously, it would go to the history file in that case.
- *condor_submit* can now write the ClassAds it generates to a file instead of sending them to the *condor_schedd* daemon.
- When the *condor_master* daemon sends an obituary e-mail, it prints the last few lines of the log file for that daemon, and the name of the file. This e-mail now contains the full path name of that log file, not just the file name. This is more convenient for sites which run multiple instances of the same daemon on one machine.
- Added new policy for parallel universe jobs to control how they exit. If the attribute `ParallelShutdownPolicy` is set to the string "WAIT_FOR_ALL", then Condor will wait until every node in the parallel job has completed to consider the job finished. If this attribute is not set, or is set to any other string, the default policy is in effect. This policy is needed for MPI jobs: when the first node exits, the whole job is considered done, and condor kills all other running nodes in that parallel job.

- Added new Windows specific ClassAd attributes:

- WindowsMajorVersion
- WindowsMinorVersion
- WindowsBuildNumber

For definitions, please see the unnumbered subsection labeled Machine ClassAd Attributes on page 886.

- Added new authorization levels to allow fine-grained control over the security settings that are used by the collector when receiving ClassAd updates by different types of daemons: `ADVERTISE_MASTER`, `ADVERTISE_STARTD`, and `ADVERTISE_SCHEDD`. An example of what you can do with this is to require that all *condor_startds* that join the pool be authenticated with a pool password and exist within a restricted set of IP addresses, while schedds may join the pool from a broader set of IP addresses and must authenticate with X509 credentials.
- Added ability to throttle in Condor's file transfer mechanism the maximum number of simultaneous stage-outs and stage-ins for jobs submitted from the same *condor_schedd*. The configuration variables are `MAX_CONCURRENT_DOWNLOADS` and `MAX_CONCURRENT_UPLOADS`. The default is 10 simultaneous uploads of input files and 10 simultaneous downloads of output files. These limits currently do not apply to grid universe jobs or standard universe jobs.
- Added `SCHEDD_QUERY_WORKERS`, which is 3 by default in Unix, and which is ignored in Windows. This specifies the maximum number of concurrent sub-processes that the *condor_schedd* will spawn to handle queries.
- Condor-C now uses a more efficient protocol when querying the status of jobs from Condor 6.9.5 and newer *condor_schedd* daemons.
- Added 4 new counters to the job ClassAd:
 - NumJobStarts
 - NumJobReconnects
 - NumShadowExceptions
 - NumShadowStarts

For more information, please see their descriptions in section 9 on page 879.

- Added a new attribute, `GridJobStatus`, to the ClassAds of **grid** universe jobs. This string shows the job's status as reported by the remote job management system.
- *condor_q -analyze* now shows the full hold reason for jobs that are on hold.
- Increased efficiency of *condor_preen* when there are large numbers of jobs in the job queue. Without this, the *condor_schedd* would become unresponsive for a long time (e.g. 10 minutes with 20,000 jobs in the queue) whenever *condor_preen* was activated.
- A 6.9.5 *condor_q* can now query an older *condor_quill* daemon directly for job information.
- Reduced memory requirements of *condor_shadow*.

- Added the ability to *condor_submit* to list unused or unexpanded variables in submission file.
- Added the capability to assign priorities to DAG nodes. Ready nodes within a DAG are submitted in priority order by *condor_dagman*.
- Added the capability to assign categories to DAG nodes, and throttle submission of node jobs by category.
- `USE_CLONE_TO_CREATE_PROCESSES` (which defaults to True) is now supported on ppc64, SUSE 9. This also fixes a bug in which the Yellow Dog Linux version of Condor installed on a ppc64 SUSE 9 machine would fail to start jobs.
- When the *condor_preen* sends email about old files being found, it now includes the name of the machine and the number of files found in the subject of the message.
- The user log reading code is now able to handle global event log rotations correctly. The API is backwards compatible, but with several new methods, it is able to invisibly handle rotated event log files.
- The user log writer code now generates a header record (as a “generic” event) with some meta information to the event log. This header is not written to the “user log”, only to the global event log. Some of the information stored in this header is used by the enhanced log reader (see above) to more reliably detect rotated log files.
- The Grid Monitor now refrains from polling the status of jobs that it has learned are done.
- For grid-type condor jobs, the *condor_gridmanager* is now more efficient when querying the status of jobs on the remote *condor_schedd* when there are jobs with different X509 subjects.
- URLs can now be given for the input and output files of grid-type nordugrid jobs in the grid universe. URLs are forwarded to the NorduGrid server, which performs the transfers.
- The length of the paths to the job’s initial working directory, user log, and input/output files are no longer limited to 255 characters. Previously, *condor_submit* would refuse to accept jobs exceeding this `POSIX_PATH_MAX` limit. Now the only limit is whatever limit the operating system enforces on the system where the files are accessed.
- The *condor_startd* now receives updates to the job ClassAd from the *condor_starter*. The primary benefit of this is that `DiskUsage` is updated and can therefore be used in policy expressions, such as `PREEMPT`. The frequency of updates is determined by `STARTER_UPDATE_INTERVAL`.
- Several improvements have been made to Condor’s ability to run using privilege separation on the execute side. See section 3.6.12 for details.
- Added support on Linux systems for reliably tracking all of a job’s processes using a dedicated supplementary group ID. This has the advantage of working regardless of whether a job runs using a dedicated user account. See section 3.12.10 for details.

Configuration Variable Additions and Changes:

- The new variables `MAX_CONCURRENT_DOWNLOADS` and `MAX_CONCURRENT_UPLOADS` limit the number of simultaneous file transfers that may take place through Condor's file transfer mechanism for jobs submitted to the same *condor_schedd*. The default is 10 simultaneous uploads of input files and 10 simultaneous downloads of output files. These limits currently do not apply to grid universe jobs or standard universe jobs. See page 196 for more information.
- The default for `JOB_START_DELAY` has been changed from 2 seconds to 0. This means the *condor_schedd* will not limit the rate at which it starts up *condor_shadow* processes by default. The delay between startup of jobs may now be controlled individually for jobs using the job attribute `NextJobStartDelay`, which defaults to 0 seconds and is at most `MAX_NEXT_JOB_START_DELAY`, which defaults to 10 minutes.
- The new variable `SCHEDD_QUERY_WORKERS` specifies the maximum number of concurrent sub-processes that the *condor_schedd* will spawn to handle queries. This is ignored in Windows. In Unix, the default is 3. See page 196 for more details.
- The new variable `WANT_UDP_COMMAND_SOCKET` controls if Condor daemons should create a UDP command socket in addition to the TCP command socket (which is required). The default is `True`, but it is now possible to completely disable UDP command sockets by defining this to `False`. See section 3.3.3 on page 157 for more information.
- The new variable `NEGOTIATOR_INFORM_STARTD` controls if the *condor_negotiator* should inform the *condor_startd* when it has been matched with a job. The default is `True`. See section 3.3.17 on page 212 for more information.
- The new variable `SHADOW_LAZY_QUEUE_UPDATE` controls if the *condor_shadow* should immediately update the job queue for certain attributes (for example, the new `NumJobStarts` and `NumJobReconnects` counters) or if it should wait and only update the job queue on the next periodic update. The default is `True` to do lazy, periodic updates. See section 3.3.12 on page 202 for more information.
- The new variable `WARN_ON_UNUSED_SUBMIT_FILE_MACROS` controls if *condor_submit* should warn when there are unused or unexpanded variables in a submit file. The default is `True` to list unused or unexpanded variables.
- `SCHEDD_ROUND_ATTR_XXXX` can now take a value that is a percentage, such as 25%. This causes the value for the attribute `<XXXX>` to be rounded up to the specified percentage of its closest order of magnitude. For example, a setting of 25% will cause a value near 100 to be rounded up to the next multiple of 25 and a value near 1000 will be rounded up to the next multiple of 250. The purpose of this rounding is to be able to better group similar jobs together for negotiation purposes. The configuration variables `SCHEDD_ROUND_ATTR_ImageSize`, `SCHEDD_ROUND_ATTR_ExecutableSize`, and `SCHEDD_ROUND_ATTR_DiskUsage` now have a default value of 25% rather than 4. The result is that instead of rounding to 10MB multiples, the rounding scales at roughly 25% of the number being rounded.
- The default for `STARTER_UPDATE_INTERVAL` has been changed from 20 minutes to 5 minutes.

- The new parameters `PRIVSEP_ENABLED` and `PRIVSEP_SWITCHBOARD` are required when setting up execute-side Condor to use privilege separation. See section 3.6.12 for details.
- The new parameters `USE_GID_PROCESS_TRACKING`, `MIN_TRACKING_GID`, and `MAX_TRACKING_GID` are used when setting up a Linux machine to use process tracking based on dedicated supplementary group IDs. See section 3.12.10 for details.

Bugs Fixed:

- Updated the SOAP API's enum `UniverseType` to include all the supported universes.
- Missing files that Quill needed to function now appear in the downloadable release of Condor.
- When a *condor_starter* discovered a missing username in the process of discovering the owner of an executing job, a cryptic and misleading error message was emitted to the daemon log. The error text has been cleaned up to be more meaningful.
- On Windows daylight saving is handled incorrectly by `stat()` and `fstat()`. According to the MSDN, they both return the UTC time of a file; however, if daylight saving is detected, the time is adjusted by one hour, which results in the *condor_master* thinking that a different version of it has been installed. In which case it recycles itself, and it's child process twice a year: not exactly what one would expect given that UTC time is not intended to pay attention to these regional changes.
- When the master starts a collector on a central manager, the master now pauses for a short time before starting any other daemons. This helps the other daemons to appear in the collector more quickly.
- Patched the parallel universe scripts `lamscrip` and `mp1scrip` so that they work with newer versions of the GNU textutils.
- Fixed a bad bug in the standard universe—introduced in Condor 6.9.4, which would cause corruption of any binary data being written to any fd the application opened. If an application only writes ASCII data to an fd, the application will not encounter this bug.
- Condor daemons will now print an error message in the logs when `<SUBSYS>_ATTRS` contains attributes that are not valid `ClassAd` values. Previously, this was a silent error. The most common reason for this problem is an unquoted string value.
- The *condor_negotiator* now prints out the value of the configuration parameter `PREEMPTION_REQUIREMENTS` if it is set. Previously, it always logged that it was unset, even when it was.
- Fixed bug in the master that occurred if the collector was configured to use an ephemeral command port (i.e. by explicitly setting the port to 0). The collector is now more reliable in this situation.
- Standard universe jobs are no longer restricted in the length of file names that may be passed to system calls within the program. Previously, file names approaching 255 characters or more could cause the program to crash or behave incorrectly.

- Fixed a long-standing bug causing *condor_submit* to fail when given a requirements expression longer than 2048 characters.
- Fixed a bug introduced in Condor 6.9.4 that caused grid universe jobs of type *gt4* to not work when the Condor daemons were started as root and any file transfer was associated with the job.
- Fixed a bug introduced in Condor 6.9.4 that caused the *condor_gridmanager* to exit immediately on startup when the Condor daemons were started as root and a condor username didn't exist.
- Removed race condition that was causing the *condor_schedd* to core dump on Windows when the Condor service was stopped.
- When grid universe jobs of grid-type *condor*, *lsf*, or *pbs* are running, *condor_q* will now show the correct accumulated runtime.
- When removing grid universe jobs of type *gt2* that have just finished executing, the chance of encountering Globus GRAM error 31 (the job manager failed to cancel the job as requested) is now much reduced.
- Fixed a problem introduced in 6.9.4: the *condor_schedd* would hang when given a constraint with *condor_hold* that included jobs that the user did not have permission to modify.
- Fixed a problem from 6.9.4 in which the *schedd* would not relinquish a claimed *startd* after reconnecting to a disconnected job. After the job finished, the *startd* would remain in the claimed idle state until the claim lease expired (20 minutes by default).
- Applied the *QUERY_TIMEOUT* to fix problem where the *schedd* would block for a long time when doing negotiation with *flocked* or *HAD* negotiator, and one of the collectors was not routable (for instance, when the machine is powered off). Previously, there was no time-out and would result in the *schedd* waiting for the connection attempt to fail, which may take a long time.
- If both *EXECUTE_LOGIN_IS_DEDICATED* and *DEDICATED_EXECUTE_ACCOUNT_REGEX* are defined, the latter now takes precedence, whereas previously the reverse was true.
- Fixed a problem where if *STARTD_RESOURCE_PREFIX* was set to anything besides *slot* (the default), all jobs would run using the “*condor-reuse-slot1*” (Windows) or *SLOT1_USER* account, regardless of the actual slot used for execution. This problem existed in versions 6.9.3 and 6.9.4 of Condor.
- Undocumented “DAGMan helper” functionality has been removed due to disuse
- Reworked Condor's detection of CPUs and “Hyper Threads” under Linux. It now correctly detects these on all machines that we've been able to test against. No configuration changes are involved in this fix.
- When a standard universe job becomes held due to user job policy or a version mismatch, a hold reason is now set in the job ad.

- Invalid `QUILL_DB_TYPE` settings could result in a segmentation fault in *condor_q*. Condor now ignores invalid settings and assumes *PostgreSQL*.
- In rare cases, *condor_reconfig* could cause *condor_master* and one of its children to become deadlocked. This problem was only possible with security negotiation enabled, and it has therefore existed in all versions of Condor since security negotiation was added.
- Fixed a potential crash in the *condor_starter* if it's told to shutdown while it's disconnected from the *condor_shadow*.
- Fixed the global event log rotation code. Previously, if two or more processes were concurrently writing to the event log, they didn't correctly detect that another writer process had rotated the file, and would do their own rotation, resulting in data loss.
- Fixed a bug in the *condor_schedd* that could cause it to not match any jobs for long periods of time.
- Fixed potential crash when GCB was turned on.
- Removed spurious attempts to open file `/home/condor/execute/dir_####/userdir/externals/install/globus-4.0.5/cert.pem` when SSL authentication is enabled.
- Fixed problem where local universe jobs could leave stray processes behind after termination.
- Fixed a memory leak that affected all daemons receiving ClassAds via the network if encryption were enabled. This bug existed in versions 6.9.3 and 6.9.4.
- On Windows, fixed a problem that could cause spurious failures with Condor-C or with streaming a job's standard output or error.

Known Bugs:

- Condor on MacOSX 10.4 on the PowerPC architecture. will report zero image size and resident set size for jobs. This is due to bugs in the MacOSX 10.4 kernel on the PowerPC.
- There are some known vulnerabilities in the Virtual Machine (VM) universe support included in this release that can allow VM universe jobs to read arbitrary files on the host machine. VM universe is disabled by default, so this potential vulnerability does not affect most sites. However, VM universe should not be enabled unless the job policy only allows users that you trust completely to run jobs on the machine.
- The *condor_startd* will crash if `ENABLE_BACKFILL` is set to True. This was also the case in 6.9.4.
- The pool password authentication method fails to authenticate (and in fact will cause the client to crash).
- If *condor_dagman* cannot execute a PRE or POST script (for example, if the script name is specified incorrectly), *condor_dagman* will hang indefinitely. (Note that if the script is executed and fails, *condor_dagman* deals with this correctly.) (This bug was fixed in version 6.8.7.)

Version 6.9.4

Release Notes:

- The default in standard universe for **copy_to_spool** is now `true`. In 6.9.3, it was changed to `false` for all universes for performance reasons, but this is deemed too risky for standard universe, because any modification of the executable is likely to make it impossible to resume execution using checkpoint files made from the original version of the executable.
- Version 1.5.0 of the Generic Connection Broker (GCB) is now used for building Condor. This version of GCB fixes a few critical bugs.
 - GCB was unable to pass information about sockets registered at a GCB broker to child processes due to a bug in the way a special environment variable was being set.
 - All sockets for outbound connections were being registered at the GCB broker, which was putting severe strain on the GCB broker even under relatively low load. Now, only sockets that are listening for inbound connections are registered at the broker.
 - The `USE_CLONE_TO_CREATE_PROCESSES` setting was causing havoc for applications linked with GCB. This configuration setting is now always disabled if GCB is enabled.
 - Fixed a race condition in `GCB_connect()` that would frequently cause `connect()` attempts to fail, especially non-blocking connections.
 - Fixed bugs in `GCB_select()` when GCB changes the direction of a connection from active to passive (for example, so that a Condor daemon running behind a firewall will use an outbound connection to communicate with a public client that had attempted to initiate contact via the GCB broker).
 - Also improved logging at the GCB broker.

Additionally, there was a bug in how Condor was publishing the classified ads for GCB-enabled daemons. Condor used to be re-writing any attributes containing an IP address when a classified ad was sent over a network connection (in an effort to provide correct behavior for multi-homed machines). Now, this re-writing is disabled whenever GCB is enabled, since GCB already has logic to determine the correct IP addresses to advertise.

For more information about GCB, see section 3.7.3 on page 332.

- The owner of the log file for the *condor_gridmanager* has changed to the `condor` user. In Condor 6.9.3 and previous versions, it was owned by the user submitting the job. Therefore, the owner of and permissions on an existing log file are likely to be incorrect. Condor issues an error if the *condor_gridmanager* is unable to read and write the existing file. To correct the problem, an administrator may modify file permissions such that the `condor` user may read and write the log file. Alternatively, an administrator may delete the file, and Condor will create a new file with the expected owner and permissions. In addition, the definition for `GRIDMANAGER_LOG` in the `condor-config.generic` file has changed for Condor 6.9.4.

New Features:

- Condor has been ported to Yellow Dog 5.0 Linux on the PPC architecture. This port of Condor will also run on the Sony Playstation 3 running said distribution of Linux.
- Enhanced the standard universe to utilize Condor's privilege separation mechanism.
- Implemented a completely new version of Quill. Quill can now record information about all the daemons into a relational database. See section 3.11 for details on Quill.
- Jobs in the mpi universe now can have \$\$ expanded in their ads in the same way as other universes.
- Added the **vm** universe, to facilitate running jobs under Xen or VMware virtual machines.
- Added the **-subsystem** command to *condor_status* that queries all ClassAds of a given type.
- Improved the speed at which the *condor_schedd* writes to its database file *job_queue.log* and the job history file. In benchmark tests, this roughly doubles the maximum throughput rate to approximately 20 jobs per second, although actual performance depends on the specific hardware used.
- The *condor_startd* now records historical statistics about the total time (in seconds) that it spends in every state/activity pair. If a given slot spent more than 0 seconds in any of the possible pairs, the specifically-named ClassAd attribute for that pair is defined in the slot's ClassAd. The list of possible new machine attributes (alphabetically):

```
TotalTimeBackfillBusy
TotalTimeBackfillIdle
TotalTimeBackfillKilling
TotalTimeClaimedBusy
TotalTimeClaimedIdle
TotalTimeClaimedRetiring
TotalTimeClaimedSuspended
TotalTimeMatchedIdle
TotalTimeOwnerIdle
TotalTimePreemptingKilling
TotalTimePreemptingVacating
TotalTimeUnclaimedBenchmarking
TotalTimeUnclaimedIdle
```

- The *condor_shadow* now waits and retries after failing to commit the final update to the job ClassAd in the *condor_schedd*'s job queue, rather than immediately aborting and causing the job to be requeued to run again. See page 203 for the related configuration options.
- If the *condor_starter* fails with a core dump on Unix, the core dump file is now put in the LOG directory. Previously, it was deleted by the *condor_startd*.

- Added a small amount of randomization to the default values of `PERIODIC_CHECKPOINT` (in the example config file) and `PASSWD_CACHE_REFRESH` (in Condor's internal default) in order to decrease the chances of synchronized timing across many processes causing overloading of servers.
- Added the new submit command **`cron_window`**. It is an alias to **`deferral_window`**.
- Optimized the submission of grid-type `gt4` grid universe jobs to the remote resource. Submission now takes one operation instead of three.
- Added new functionality for multi-homed machines (those with multiple network interfaces) to allow Condor to handle private networks in some cases without having to use the Generic Connection Broker (GCB). See the entries below that describe the new `PRIVATE_NETWORK_NAME` and `PRIVATE_NETWORK_INTERFACE` configuration variables.

Configuration Variable Additions and Changes:

- Added `SLOTx_EXECUTE`. This allows the execute directory to be configured independently for each batch slot. You could use this, for example, to have jobs on a multi-CPU machine using scratch space on different disks so that there is less chance of them interfering with each other. See page 152 for more details.
- The semantics of `SLOT_TYPE-<N>` has changed slightly. Previously, any resource shares left undefined would default to a fractional share equal to $1/\text{NUM_CPUS}$. Now, the default is `auto`, which causes all remaining resources to be evenly divided. This is more convenient in cases where some slots are configured to take more or less than their "fair" share and the rest are desired to evenly split the remainder. The underlying reason for this change was to be able to better support the specification of disk partition shares in all the possible cases: The `auto` share takes into account how many slots are sharing the same disk partition.
- When set to `True`, the new configuration variable `LOGS_USE_TIMESTAMP` will cause Condor to print all daemon log messages using a Unix timestamp instead of a formatted date string. This feature is useful for debugging Condor Glideins that may be executing in different timezones. It should be noted that this does not affect job user logs. The default is `False`.
- The existing configuration variable `LOG_ON_NFS_IS_ERROR` has changed behavior. When set to `False`, *condor_submit* does not emit a warning about user logs files being on NFS.
- The existing configuration variables `DAEMON_LIST`, `DC_DAEMON_LIST`, and `MASTER_HA_LIST` have changed behavior. Trailing commas are now ignored. Previously, trailing commas could cause the *condor_master* to misbehave, including exiting with an error.
- The `<SUBSYS>_DAEMON_AD_FILE` was defined for the *condor_schedd*. This setting was first made available in Condor 6.9.1 but was not used for any daemon. It appears in the configuration file as `SCHEDD_DAEMON_AD_FILE` and is set to the file `.schedd.classad` in the

LOG directory. This setting is not necessary unless you are using the Quill functionality, and pools may upgrade to 6.9.4 without setting it if they are not using Quill.

- Added new configuration variables `DBMSD`, `DBMSD_ARGS`, and `DBMSD_LOG`, which define the location of the *condor_dbmsd* daemon, the command line arguments to that daemon, and the location of the daemon's log. Default values are

```
DBMSD = $(SBIN)/condor_dbmsd
DBMSD_ARGS = -f
DBMSD_LOG = $(LOG)/DbmsdLog
```

These configuration variables are only necessary when using Quill, and then must be defined on only one machine in the Condor pool.

- Added new configuration variables `PRIVATE_NETWORK_NAME` and `PRIVATE_NETWORK_INTERFACE`, which allow Condor daemons to function more properly on multi-homed machines and in certain network configurations that involve private networks. There are no default values, both must be explicitly defined to have any effect. See section 3.3.6 on page 165 for more information about these two new settings.
- Added new configuration variables `EVENT_LOG`, `MAX_EVENT_LOG`, `EVENT_LOG_USE_XML`, and `EVENT_LOG_JOB_AD_INFORMATION_ATTRS` to specify the new event log which logs job user log events, but across all users. See section 3.3.4 on page 161 for definitions of these configuration variables.

Bugs Fixed:

- Trailing commas in lists of items in submit files and configuration files are now ignored. Previously, Condor would treat trailing commas in various surprising ways.
- Numerous bugs in GCB and the interaction between Condor and GCB. See the release notes above for details.
- The submit file entry “coresize” was not being honored properly on many universe. It is now honored on all universes except *pvm* and the grid universes (except where the grid type is Condor). For the *java* universe, it controls the core file size for the JVM itself.
- The *condor_configure* installation script now allows Condor to be installed on hosts without a fully-qualified domain name.
- Fixed a bug in *condor_dagman*: if a DAG run with a per-DAG configuration file specification generated a rescue DAG, the rescue DAG file did not contain the appropriate DAG configuration file line. (This bug was introduced when the per-DAG configuration file option was added in version 6.9.2.)
- Fixed a bug introduced in 6.9.3 when handling local universe jobs. The starter ignored failures in contacting the *condor_schedd* in the final update to the job queue.

- When the *condor_schedd* is issued a graceful shutdown command, any jobs that running with a job lease are allowed to keep running. When the *condor_schedd* starts back up at a later time, it will spawn *condor_shadow* to reconnect to the jobs if they are still executing. This mimics the same behavior as a fast shutdown. This also fixes a bug in 6.9.3 in which the *condor_schedd* would fail to reconnect to jobs that were left running during a graceful shutdown.
- When the *condor_starter* is gracefully shutting down and if it has become disconnected from the *condor_shadow*, it will wait for the job lease time to expire before giving up on telling the *condor_shadow* that the job was evicted. Previously, the *condor_starter* would exit as soon as it was done evicting the job.
- Job ad attribute `HoldReasonCode` is now properly set when *condor_hold* is called and when jobs are submitted on hold.
- If a job specified a job lease duration, and the *condor_schedd* was killed or crashed, the *condor_shadow* used to notice when the *condor_schedd* was gone, and gracefully shutdown the job (evicting the job at the remote site). Now, the *condor_shadow* honors the job lease duration, and if the lease has not yet expired, it simply exists without evicting the job, in the hopes that the *condor_schedd* will be restarted in time to reconnect to the still-running job and resume computation.
- Fixed a bug from 6.9.3 in which *condor_q -format* no longer worked when given an expression (as opposed to simple attribute reference). The expression was always treated as being undefined.
- When a condor daemon such as the *condor_schedd* or *condor_negotiator* tried to establish many new security sessions for UDP messages in a short span of time, it was possible for the daemon to run out of file descriptors, causing it to abort execution and be restarted by the *condor_master*. A problem was found and fixed in the mechanism that protects against this.
- Improved error descriptions when Condor-C encounters failures when sending input files to the remote schedd.
- Rare failure conditions during stage in would cause Condor-C to put the state of the job in the remote schedd into an invalid state in which it would run but later fail during stage out. This now results in the job on the submit side going on hold with a staging failure.
- Fixed a bug which could cause *condor_store_cred* to crash during common use.
- Fixed a bug where the vanilla universe *condor_starter* would possibly crash when running a job not as the owner of the job.
- Fixed a bug which would cause a *condor_starter* being used for the local universe to core dump.
- Fixed a bug which caused the *condor_schedd* to core dump while processing a job's crontab entries in the submit description file.
- Fixed a privilege separation bug in the standard universe *condor_starter*.

Known Bugs:

- **standard** universe jobs do not work when writing binary data. The behavior exhibited in this case may include the job crashing, or corrupt binary written.
- **grid** universe jobs for the `gt4` grid type do not work, if Condor daemons are started as root and there is file transfer associated with or specified by the job. These jobs are placed on hold.
- The `STARTD_RESOURCE_PREFIX` setting on Windows results in broken behavior on both Condor 6.9.3 and 6.9.4. Specifically, when this setting is given a value other than its default (“slot”), all jobs will run using the “condor-reuse-slot1” user account, regardless of the actual slot used for execution.

Additions and Changes to the Manual:

- New documentation for the new **vm** universe in the User’s Manual, section 2.11. Definitions of configuration variables for the **vm** universe are in section 3.3.28.
- New RDBMS schema tables added for Quill in section 3.11.4.
- ClassAd attribute definitions reside in a new appendix. In addition to machine and job attributes, DaemonMaster and Scheduler attributes are included.

Version 6.9.3

Release Notes:

- As of version 6.9.3, the entire Condor system has undergone a major terminology change. For almost 10 years, Condor has used the term *virtual machine* or *vm* to refer to each distinct resource that could run a Condor job (for example, each of the CPUs on an SMP machine). Back when we chose this terminology, it made sense, since each of these resource was like an independent machine in a pool, with its own state, ClassAd, claims, and so on. However, in recent years, the term *virtual machine* is now almost universally associated with the kinds of virtual machines created using tools such as VMware and Xen. Entire operating systems run inside a given process, usually emulating the underlying hardware on a host machine. So, to avoid confusion with these other kinds of virtual machines, the old *virtual machine* terminology has been replaced by the term *slot*.

Numerous configuration settings, command-line arguments to Condor tools, ClassAd attribute names, and so on, have all been modified to reflect the new *slot* terminology. In general, the old settings and options will still work, but are now retired and may disappear in the future.

- The `condor_install` installation script has been removed. All sites should use `condor_configure` when setting up a new Condor installation.

- The `SECONDARY_COLLECTOR_LIST` configuration variable has been removed. Sites relying on this variable should instead use the configuration variable `COLLECTOR_HOST`. It may be used to define a list of *condor_collector* daemon hosts.
- Cleaned up and improved help information for *condor_history*.

New Features:

- Numerous scalability and performance improvements. Given enough memory, the *schedd* can now handle much larger job queues (e.g. 10s of thousands) without the severe degradation in performance that used to be the case.
- Added the `START_LOCAL_UNIVERSE` and `START_SCHEDULER_UNIVERSE` parameters for the *condor_schedd*. This allows administrators to control whether a Local/Scheduler universe job will be started. This expression is evaluated against the job's ClassAd before the `Requirements` expression.
- All Local and Scheduler universe jobs now have their `Requirements` expressions evaluated before execution. If the expression evaluates to false, the job will not be allowed to begin running. In previous versions of Condor, Local and Scheduler universe jobs could begin execution without the *condor_schedd* checking the validity of the `Requirements`.
- Added `SCHEDD_INTERVAL_TIMESLICE` and `PERIODIC_EXPR_TIMESLICE`. These indicate the maximum fraction of time that the *schedd* will spend on the respective activities. Previously, these activities were done on a fixed interval, so with very large job queue sizes, the fraction of time spent was increasing to unreasonable levels.
- Under Intel Linux, added `USE_CLONE_TO_CREATE_PROCESSES`. This defaults to true and results in scalability improvements for processes using large amounts of memory (e.g. a *schedd* with a lot of jobs in the queue).
- Jobs in the parallel universe now can have \$\$ expanded in their ads in the same way as other universes.
- Local universe jobs now support policy expression evaluation, which includes the `ON_EXIT_REMOVE`, `ON_EXIT_HOLD`, `PERIODIC_REMOVE`, `PERIODIC_HOLD`, and `PERIODIC_RELEASE` attributes. The periodic expressions are evaluated at intervals determined by the `PERIODIC_EXPR_INTERVAL` configuration macro.
- Jobs can be scheduled to executed periodically, similar to the `crontab` functionality found in Unix systems. The *condor_schedd* calculates the next runtime for a job based on the new `CRON_MINUTE`, `CRON_HOUR`, `CRON_DAY_OF_MONTH`, `CRON_MONTH`, and `CRON_DAY_OF_WEEK` attributes. A preparation time defined by the `CRON_PREP_TIME` attribute allows a job to be submitted to the execution machine before the actual time the job is to begin execution. Jobs that would like to be run repeatedly will need to define the `ON_EXIT_REMOVE` attribute properly so that they are re-queued after executing each time.

- Condor now looks for its configuration file in `/usr/local/etc` if the `CONDOR_CONFIG` environment variable is not set and there is no `condor_config` file located in `/etc/condor`. This allows a default Condor installation to be more compatible with Free BSD.
- If a user job requests streaming input or output in the submit file, the job can now run with job leases and the job will continue to run for the lease duration should the submit machine crash. Previously, jobs with streaming i/o would be evicted if the submit machine crashed. While the submit machine is down, if the job tried to issue a streaming read or write, the job will block until the submit machine returns or the job lease expires.
- Ever since version 6.7.19, *condor_submit* has added a default job lease duration of 20 minutes to all jobs that support these leases. However, there was no way to disable this functionality if a user did not want job lease semantics. Now, a user can place `job_lease_duration = 0` in their submit file to manually disable the job lease.
- Added new configuration knob `STARTER_UPLOAD_TIMEOUT` which sets the timeout for the *condor_starter* to upload output files to the *condor_shadow* on job exit. The default value is 200 seconds, which should be sufficient for serial jobs. For parallel jobs, this may need to be increased if many large output files are sent back to the shadow on job exit.
- *condor_dagman* now aborts the DAG on “scary” submit events. These are submit events in which the Condor ID of the event does not match the expected value. Previously, *condor_dagman* printed a warning, but continued. To restore Condor to the previous behavior, set the new `DAGMAN_ABORT_ON_SCARY_SUBMIT` configuration variable to `False`.
- When the *condor_master* detects that its GCB broker is unavailable and there is a list of alternative brokers, it will restart immediately if `MASTER_WAITS_FOR_GCB_BROKER` is set to `False` instead of waiting for another broker to become available. *condor_glidein* now sets `MASTER_WAITS_FOR_GCB_BROKER` to `False` in its configuration file.
- When using GCB and a list of brokers is available, the *condor_master* will now pick a random broker rather than the least-loaded one.
- All Condor daemons now evaluate some ClassAd expressions whenever they are about to publish an update to the *condor_collector*. Currently, the two supported expressions are:

DAEMON_SHUTDOWN If `True`, the daemon will gracefully shut itself down and will not be restarted by the *condor_master* (as if it sent itself a *condor_off* command).

DAEMON_SHUTDOWNFAST If `True`, the daemon will quickly shut itself down and will not be restarted by the *condor_master* (as if it sent itself a *condor_off* command using the **-fast** option).

For more information about these expressions, see section 3.3.5 on page 164.

- When the *condor_master* sends email announcing that another daemon has died, exited, or been killed, it now notes the name of the machine, the daemon’s name, and a summary of the situation in the Subject line.

- Anyplace in a Condor configuration or submit description file where wild cards may be used, you can now place wild cards at both the beginning and end of the string pattern (i.e. match strings that contain the text between the wild cards anywhere in the string). Previously, only one wild card could appear in the string pattern.
- Added optional configuration setting `NEGOTIATOR_MATCH_EXPRS`. This allows the negotiator to insert expressions into the matched ClassAd. See page 214 for more information.
- Increased speed of ClassAd parsing.
- Added `DEDICATED_EXECUTE_ACCOUNT_REGEX` and deprecated the boolean setting `EXECUTE_LOGIN_IS_DEDICATED`, because the latter could not handle a policy where some jobs run as the job owner and some run as dedicated execution accounts. Also added support for `STARTER_ALLOW_RUNAS_OWNER` under Unix. See Section 3.3.7 and Section 3.6.11 for more information.
- All Condor daemons now publish a `MyCurrentTime` attribute which is the current local time at the time the update was generated and sent to the *condor_collector*. This is in addition to the `LastHeardFrom` attribute which is inserted by the *condor_collector* (the current local time at the collector when the update is received).
- *condor_history* now accepts partial command line arguments. For example, `-constraint` can be abbreviated `-const`. This brings *condor_history* in line with other Condor command line tools.
- *condor_history* can now emit ClassAds formatted as XML with the new `-xml` option. This brings *condor_history* more in line *condor_q*.
- The `$$` substitution macro syntax now supports the insertion of literal `$$` characters through the use of `$$ (DOLLARDOLLAR)`. Also, `$$` expansion is no longer recursive, so if the value being substituted in place of a `$$` macro itself contains `$$` characters, these are no longer interpreted as substitution macros but are instead inserted literally.
- When started as root on a Linux 64-bit x86 machine, Condor daemons will now leave core files in the log directory when they crash. This matches Condor's behavior on most other Unix-like operating systems, including 32-bit x86 versions of Linux.
- The `_CONDOR_SLOT` variable is now placed into the environment for jobs of all universes. This variable indicates what slot a given job is running on, and will have the same value as the `SlotID` from the machine classified ad where the job is running. The `_CONDOR_SLOT` variable replaces the deprecated `CONDOR_VM` environment variable, which was only defined for standard universe jobs.
- Added a `USE_PROCD` configuration parameter. If this parameter is set to true for a given daemon, the daemon will use the *condor-procd* program to monitor process families. If set to false, the daemon will execute process family monitoring logic on its own. The *condor-procd* is more scalable and is also an essential piece in the ongoing privilege separation effort. The disadvantage of using the ProcD is that it is newer, less-hardened code.

Configuration Variable Additions and Changes:

- The `SECONDARY_COLLECTOR_LIST` configuration variable has been removed. Sites relying on this variable should instead use the configuration variable `COLLECTOR_HOST` to define a list of *condor_collector* daemon hosts.
- Added new configuration variables `START_LOCAL_UNIVERSE` and `START_SCHEDULER_UNIVERSE` for the *condor_schedd* daemon. These boolean expressions default to `True`. `START_LOCAL_UNIVERSE` is relevant only to local universe jobs. `START_SCHEDULER_UNIVERSE` is relevant only to scheduler universe jobs. These new variables allow an administrator to define a `START` expression specific to these jobs. The expression is evaluated against the job's ClassAd before the `Requirements` expression.
- Added new configuration variables `SCHEDD_INTERVAL_TIMESLICE` and `PERIODIC_EXPR_TIMESLICE`. These configuration variables address a scalability issue for very large job queues. Previously, the *condor_schedd* daemon handled an activity related to counting jobs, as well as the activity related to evaluating periodic expressions for jobs at the fixed time interval of 5 minutes. With large job queues, the fraction of the *condor_schedd* daemon execution time devoted to these two activities became excessive, such that it could be doing little else. The fixed time interval is now gone, and Condor calculates the amount of time spent on the two activities, using these new configuration variables to calculate an appropriate time interval.

Each is a floating point value within the range (noninclusive) 0.0 to 1.0. Each determines the maximum fraction of the time interval that the *condor_schedd* daemon will spend on the respective activity. `SCHEDD_INTERVAL_TIMESLICE` defaults to the value 0.05, such that the calculated time interval will be 20 * the amount of time spent on the counting jobs activity. `PERIODIC_EXPR_TIMESLICE` defaults to the value 0.01, such that the calculated time interval will be 100 * the amount of time spent on the periodic expression evaluation activity.
- Added new configuration variable `USE_CLONE_TO_CREATE_PROCESSES`, relevant only to the Intel Linux platform. This boolean value defaults to `True`, and it results in scalability improvements for Condor processes using large amounts of memory. These processes may clone themselves instead of forking themselves. An example of the improvement occurs for a *condor_schedd* daemon with a lot of jobs in the queue.
- Added new configuration variable `STARTER_UPLOAD_TIMEOUT`, which allows a configurable time (in seconds) for a timeout used by the *condor_starter*. The default value of 200 seconds replaces the previously hard coded value of 20 seconds. This timeout before job failure is to upload output files to the *condor_shadow* upon job exit. The default value should be sufficient for serial jobs. For parallel jobs, it may need to be increased if there are many large output files.
- Added new configuration variable `DAGMAN_ABORT_ON_SCARY_SUBMIT`. This boolean variable defaults to `True`, and causes *condor_dagman* to abort the DAG on “scary” submit events. These are submit events in which the Condor ID of the event does not match the expected value. Previously, *condor_dagman* printed a warning, but continued. To restore Condor to the previous behavior, set `DAGMAN_ABORT_ON_SCARY_SUBMIT` to `False`.
- Added new configuration variable `NEGOTIATOR_MATCH_EXPRS`. It causes the *condor_negotiator* to insert expressions into the matched ClassAd. See page 214 for details.

- Added new configuration variable `DEDICATED_EXECUTE_ACCOUNT_REGEX` to replace the retired `EXECUTE_LOGIN_IS_DEDICATED`, because `EXECUTE_LOGIN_IS_DEDICATED` could not handle a policy where some jobs run as the job owner and others run as dedicated execution accounts. Also added support for the existing configuration variable `STARTER_ALLOW_RUNAS_OWNER` under Unix. See Section 3.3.7 and Section 3.6.11 for more information.
- Added new configuration variable `USE_PROCD`. This boolean variable defaults to `False` for the *condor_master*, and `True` for all other daemons. When `True`, the daemon will use the *condor_procd* program to monitor process families. When `False`, a daemon will execute process family monitoring logic on its own. The *condor_procd* is more scalable and is also an essential piece in the ongoing privilege separation effort. The disadvantage of using the *condor_procd* is that it is newer, less-hardened code.

Bugs Fixed:

- On Unix systems, Condor can now handle file descriptors larger than `FD_SETSIZE` when using the `select` system call. Previously, file descriptors larger than `FD_SETSIZE` would cause memory corruption and crashes.
- When an update to the *condor_collector* from the *condor_startd* is lost, it is possible for multiple claims to the same resource to be handed out by the *condor_negotiator*. This is still true. What is fixed is that these multiple claims will not result in mutual annihilation of the various attempts to use the resource. Instead, the first claim to be successfully requested will proceed and the others will be rejected.
- *condor_glidein* was setting `PREEN_INTERVAL=0` in the default configuration, but this is no longer a legal value, as of 6.9.2.
- *condor_glidein* was not setting necessary configuration parameters for *condor_procd* in the default glidein configuration.
- In 6.9.2, Condor daemons crashed after failing to authenticate a network connection.
- *condor_status* will now accurately report the `ActvtyTime` (activity time) value in Condor pools where not all machines are in the same timezone, or if there is clock-skew between the hosts.
- Fixed the known issue in Condor 6.9.2 where using the `EXECUTE_LOGIN_IS_DEDICATED` setting on UNIX platforms would cause the *condor_procd* to crash.
- Failure when activating a COD claim no longer will result in an opportunistic job running on the same *condor_startd* being left suspended. This problem was most likely to be seen when using the `GLEEXEC_STARTER` feature.
- In Condor 6.9.2 for Tru64 UNIX, the *condor_master* would immediately fail if started as root. This problem has been fixed.

- Condor 6.9.2 introduced a problem where the *condor_master* would fail if started as root with the UID part of the `CONDOR_IDS` parameter set to 0 (root). This issue has been fixed.

Known Bugs:

- The 6.9.3 *condor_schedd* daemon incorrectly handles jobs with leases (true by default for vanilla, java, and parallel universe jobs) when shutting down gracefully. These jobs are allowed to continue running, but when the *condor_schedd* daemon is started back up, it fails to reconnect to them. The result is that the orphaned jobs are left running for the duration of the job's lease time (a default time of 20 minutes). The state of the jobs in the restarted queue is independent of any orphaned running jobs, so these queued jobs may begin running on another machine while orphans are still running.
- *condor_q -format* in 6.9.3 does not work with expressions. It behaves as if the expression evaluates to an undefined result.

Version 6.9.2

Release Notes:

- As part of ongoing security enhancements, Condor now has a new, required daemon: *condor_procd*. This daemon is automatically started by the *condor_master*, you do not need to add it to `DAEMON_LIST`. However, you must be certain to update the *condor_master* if you update any of the other Condor daemons.
- Some configuration settings that previously accepted 0 no longer do so. Instead the daemon using the setting will exit with an error message listing the acceptable range to its log. For these settings 0 was equivalent to requesting the default. As this was undocumented and confusing behavior it is no longer present. To request a setting use its default, either comment it out, or set it to nothing (`"EXAMPLE_SETTING="`). Settings impacted include but are not limited to: `MASTER_BACKOFF_CONSTANT`, `MASTER_BACKOFF_CEILING`, `MASTER_RECOVER_FACTOR`, `MASTER_UPDATE_INTERVAL`, `MASTER_NEW_BINARY_DELAY`, `PREEN_INTERVAL`, `SHUTDOWN_FAST_TIMEOUT`, `SHUTDOWN_GRACEFUL_TIMEOUT`, `MASTER_<name>_BACKOFF_CONSTANT`, `MASTER_<name>_BACKOFF_CEILING`,
- Version 1.4.1 of the Generic Connection Broker (GCB) is now used for building Condor. This version of GCB fixes a timing bug where a client may incorrectly think a network connection has been established, and also guards against an unresponsive client from causing a denial of service by the broker. For more information about GCB, see section 3.7.3 on page 332.

New Features:

- On UNIX, an execute-side Condor installation can run without root privileges and still execute jobs as different users, properly clean up when a job exits, and correctly enforce policies specified by the Condor administrator and resource owners. Privileged functionality has been separated into a well-defined set of functions provided by a `setuid` helper program. This feature currently does not work for the standard or PVM universes.
- Added support for `EmailAttributes` in the parallel universe. Previously, it was only valid in the vanilla and standard universes.
- Added configuration parameter `DEDICATED_SCHEDULER_USE_FIFO` which defaults to true. When false, the dedicated scheduler will use a best-fit algorithm to schedule parallel jobs. This setting is not recommended, as it can cause starvation. When true, the dedicated scheduler will schedule jobs in a first-in, first-out manner.
- Added **-dump** to `condor_config_val` which will print out all of the macros defined in any of the configuration files found by the program. `condor_config_val -dump -v` will augment the output with exactly what line and in what file each configuration variable was found. **NOTE:** : The output format of the **-dump** option will most likely change in a future revision of Condor.
- Node names in `condor_dagman` DAG files can now be DAG keywords, except for `PARENT` and `CHILD`.
- Improved the log message when `OnExitRemove` or `OnExitHold` evaluates to `UNDEFINED`.
- Added the `DAGMAN_ON_EXIT_REMOVE` configuration macro, which allows customization of the `OnExitRemove` expression generated by `condor_submit_dag`.
- When using GCB, Condor can now be told to choose from a list of brokers. `NET_REMAP_INAGENT` is now a space and comma separated list of brokers. On start up, the `condor_master` will query all of the brokers and pick the least-used one for it and its children to use. If none of the brokers are operational, then the `condor_master` will wait until one is working. This waiting can be disabled by setting `MASTER_WAITS_FOR_GCB_BROKER` to `FALSE` in the configuration file. If the chosen broker fails and recovery is not possible or another broker is available, the `condor_master` will restart all of the daemons.
- When using GCB, communications between parent and child Condor daemons on the same host no longer use the GCB broker. This improves scalability and also allows a single host to continue functioning if the GCB broker is unavailable.
- The `condor_schedd` now uses non-blocking methods to send the “alive” message to the `condor_startd` when renewing the job lease. This prevents the `condor_schedd` from blocking for 20 seconds while trying to connect to a machine that has become disconnected from the network.
- `condor_advertise` can read the classad to be advertised from standard input.
- Unix Condor daemons now reinitialize their DNS configuration (e.g. IP addresses of the name servers) on reconfig.

- A configuration file for *condor_dagman* can now be specified in a DAG file or on the *condor_submit_dag* command line.
- Added *condor_cod* option **-lease** for creation of COD claims with a limited duration lease. This provides automatic cleanup of COD claims that are not renewed by the user. The default lease is infinitely long, so existing behavior is unchanged unless **-lease** is explicitly specified.
- Added *condor_cod* command **delegate-proxy** which will delegate an x509 proxy to the requested COD claim. This is primarily useful for sites wishing to use glexec to spawn the *condor_starter* used for COD jobs. The new command optionally takes an **-x509proxy** argument to specify the proxy file. If this argument is not present, *condor_cod* will search for the proxy using the same logic as *condor_submit* does.
- `STARTD_DEBUG` can now be empty, indicating a default, minimal log level. It now defaults to empty. Previously it had to be non-empty and defaulted to include `D_COMMAND`.
- The addition of the *condor_procd* daemon means that all process family monitoring and control logic is no longer replicated in each Condor daemon that needs it. This improves Condor's scalability, particularly on machines with many processes.

Bugs Fixed:

- Under various circumstances, condor 6.9.1 daemons would abort with the message, "ERROR: Unexpected pending status for fake message delivery." A specific example is when `OnExitRemove` or `OnExitHold` evaluated to `UNDEFINED`. This caused the *condor_schedd* to abort.
- In Condor 6.9.1, the *condor_schedd* would die during startup when trying to reconnect to running jobs for which the *condor_schedd* could not find a startd `ClassAd`. This would happen shortly after logging the following message: "Could not find machine `ClassAds` for one or more jobs. May be flocking, or machine may be down. Attempting to reconnect anyway."
- Improved Condor's validity checking of configuration values. For example, in some cases where Condor was expecting an integer but was given an expression such as `12*60`, it would silently interpret this as 12. Such cases now result in the condor daemon exiting after issuing an error message into the log file.
- When sending a `WM_CLOSE` message to a process on Windows, Condor daemons now invoke the helper program *condor_softkill* to do so. This prevents the daemon from needing to temporarily switch away from its dedicated service Window Station and Desktop. It also fixes a bug where daemons would leak Window Station and Desktop handles. This was mainly a problem in the *condor_schedd* when running many scheduler universe jobs.

Known Bugs:

- *condor_glidein* generates a default config file that sets `PREEN_INTERVAL` to an invalid value (0). To fix this, remove the setting of `PREEN_INTERVAL`.

- There are a couple of known issues with Condor's `GLEEXEC_STARTER` feature when used in conjunction with COD. First, the *condor_cod* tool invoked with the **delegate_proxy** option will sometimes incorrectly report that the operation has failed. In addition, the `GLEEXEC_STARTER` feature will not work properly with COD unless the UID that the each COD job runs as is different than the UID of the opportunistic job or any other COD jobs that are running on the execute machine when the COD claim is activated.
- The `EXECUTE_LOGIN_IS_DEDICATED` feature has been found to be broken on UNIX platforms. Its use will cause the *condor_procdd* to crash, bringing down the other Condor daemons with it.

Version 6.9.1

Release Notes:

- The 6.9.1 release contains all of the bug fixes and enhancements from the 6.8.x series up to and including version 6.8.3.
- Version 1.4.0 of the Generic Connection Broker (GCB) library is now used for building Condor, and it is the 1.4.0 versions of the *gcb_broker* and *gcb_relay_server* programs that are included in this release. This version of GCB includes enhancements used by Condor along with a new GCB-related command-line tool: *gcb_broker_query*. Condor 6.9.1 will not work properly with older versions of the *gcb_broker* or *gcb_relay_server*. For more information about GCB, see section 3.7.3 on page 332.

New Features:

- Improved the performance of the ClassAd matching algorithm, which speeds up the *condor_schedd* and other daemons.
- Improved the scalability of the algorithm used by the *condor_schedd* daemon to find runnable jobs. This makes a noticeable difference in *condor_schedd* daemon performance, when there are on the order of thousands of jobs in the queue.
- the `D_COMMAND` debugging level has been enhanced to log many more messages.
- Updated the version of DRMAA, which contains several great improvements regarding scalability and race conditions.
- Added the `DAGMAN_SUBMIT_DEPTH_FIRST` configuration macro, which causes *condor_dagman* to submit ready nodes in more-or-less depth-first order, if set to `True`. The default behavior is to submit the ready nodes in breadth-first order.
- Added configuration parameter `USE_PROCESS_GROUPS`. If it is set to `False`, then Condor daemons on Unix machines will not create new sessions or process groups. This is intended for use with Glidein, as we have had reports that some batch systems cannot properly track jobs that create new process groups. The default value is `True`.

- The default value for the submit file command **copy_to_spool** has been changed to `False`, because copying the executable to the spool directory for each job (or job cluster) is almost never desired. Previously, the default was `True` in all cases, except for grid universe jobs and remote submissions.
- More types of file transfer errors now result in the job going on hold, with a specific error message about what went wrong. The new cases involve failures to write output files to disk on the submit side (for example, when the disk is full). As always, the specific error number is recorded in `HoldReasonSubCode`, so you can enforce an automated error handling policy using **periodic_release** or **periodic_remove**.
- Added the `<SUBSYS>_DAEMON_AD_FILE` configuration variable, which is similar to the `<SUBSYS>_ADDRESS_FILE`. This new variable will be used in future versions of Condor, but is not necessary for 6.9.1.

Bugs Fixed:

- Fixed a bug in the *condor_master* so that it will now send obituary e-mails when it kills child processes that it considers hung.
- *condor_configure* used to always make a personal Condor with **-install** even when **-type** called for only execute or submit types. Now, *condor_configure* honors the **-type** argument, even when using **-install**. If **-type** is not specified, the default is to still install a full personal Condor with the following daemons: *condor_master*, *condor_collector*, *condor_negotiator*, *condor_schedd*, *condor_startd*.
- While removing, putting on hold, or vacating a large number of jobs, it was possible for the *condor_schedd* and the *condor_shadow* to temporarily deadlock with each other. This has been fixed under Unix, but not yet under Windows.
- Communication from a *condor_schedd* to a *condor_startd* now occurs in a nonblocking manner. This fixes the problem of the *condor_schedd* blocking when the claimed machine running the *condor_startd* cannot be reached, for example because the machine is turned off.

Known Bugs:

- Under various circumstances, condor 6.9.1 daemons abort with the message, “ERROR: Unexpected pending status for fake message delivery.” A specific example is when `OnExitRemove` or `OnExitHold` evaluated to `UNDEFINED`, which causes the *condor_schedd* to abort.
- In Condor 6.9.1, the *condor_schedd* will die during startup when trying to reconnect to running jobs for which the *condor_schedd* can not find a *startd* ClassAd. This happens shortly after logging the following message: “Could not find machine ClassAds for one or more jobs. May be flocking, or machine may be down. Attempting to reconnect anyway.”

Version 6.9.0

Release Notes:

- The 6.9.0 release contains all of the bug fixes and enhancements from the 6.8.x series up to and including version 6.8.2.

New Features:

- Preliminary support for using *glexec* on execute machines has been added. This feature causes the *condor_startd* to spawn the *condor_starter* as the user that *glexec* determines based on the user's GSI credential.
- A "per-job history files" feature has been added to the *condor_schedd*. When enabled, this will cause the *condor_schedd* to write out a copy of each job's ClassAd when it leaves the job queue. The directory to place these files in is determined by the parameter `PER_JOB_HISTORY_DIR`. It is the responsibility of whatever external entity (for example, an accounting or monitoring system) is using these files to remove them as it completes its processing.
- *condor_chirp* command now supports writing messages to the user log.
- *condor_chirp* `getattr` and `putattr` now send all classad `getattr` and `putattr` commands to the `proc 0` classad, which allows multiple `proc` parallel jobs to use `proc 0` as a scratch pad.
- Parallel jobs now support an `AllRemoteHosts` attribute, which lists all the hosts across all `procs` in a cluster.
- The `DAGMAN_ABORT_DUPLICATES` configuration macro (which causes *condor_dagman* to abort itself if it detects another *condor_dagman* running on the same DAG) now defaults to `True` instead of `False`.

Bugs Fixed:

- None.

Known Bugs:

- None.

8.7 Stable Release Series 6.8

This is a stable release series of Condor. It is based on the 6.7 development series. All new features added or bugs fixed in the 6.7 series are available in the 6.8 series. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 6.8.x releases. New features will be added in the forthcoming 6.9.x development series.

The 6.8.x series supports a different set of platforms than 6.6.x. Please see the updated table of available platforms in section 1.5 on page 5.

The details of each version are described below.

Version 6.8.8

Release Notes:

- This release fixes a security vulnerability that affects those who rely upon Condor's network message integrity checking (where the configuration is set to `SEC_DEFAULT_INTEGRITY = REQUIRED`). Not all of Condor's network communications are vulnerable to the integrity checking bug, so based on the scope of the affected parts, we consider the level of threat to be modest. A denial of service attack could be launched against Condor by an attacker who tampers with Condor's network communications. All previous releases of Condor are affected by this bug. For users of the 6.9 development series, a fix for this problem will be released as part of the new 7.0.0 stable series release, which is planned to happen near the end of 2007.

New Features:

- None.

Bugs Fixed:

- Fixed a named pipe collision on Windows: streaming error and output would not work on more than one slot (Condor version 6.8.8 terminology: Condor vm) at a time.
- Fixed a bug in Condor's network message integrity checking.
- Fixed a forward-compatibility problem when a 6.8 *condor_startd* runs jobs for a 6.9 or later *condor_schedd* and the communication between them is configured to use integrity checking or encryption. The problem caused the *condor_startd* to crash.
- Fixed a problem that sometimes caused corruption of ClassAd data that is forwarded from one *condor_collector* daemon to another via `CONDOR_VIEW_HOST`.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 6.8.7

Release Notes:

- None.

New Features:

- None.

Bugs Fixed:

- On Windows, fixed a problem that could cause spurious failures with Condor-C or with streaming a job's standard output or error.
- A claim in the state Claimed/Idle could not be preempted until it transitioned into Busy or went away of its own accord. This bug was introduced in 6.7.1.
- The user-based authorization parameters in the configuration file (for example, `ALLOW_READ`) now properly recognize values where the user name contains a wild card (for example, `*@cs.wisc.edu/bird.cs.wisc.edu`).
- A rare threading problem in the Windows version of Condor has been fixed. The problem could cause memory corruption in the *condor_starter* while receiving input files and in the *condor_schedd* while transferring input/output files for a remotely submitted job or a spooled job.
- Increased the verbosity of some error messages (related to reading log files) in *condor_dagman*.
- Fixed a bug in *condor_dagman* that would cause it to hang if it was unable to successfully spawn a PRE or POST script. This case is now dealt with as a PRE or POST script failure.

Known Bugs:

- None.

Additions and Changes to the Manual:

- None.

Version 6.8.6

Release Notes:

- Condor is now officially supported on Microsoft Vista.
- Condor is now officially supported on MacOS running natively on Intel CPUs. (and Condor binaries for Intel MacOS are now available for download).
- Condor now uses Globus 4.0.5 for GSI, pre-WS GRAM, and GridFTP support.

New Features:

- On all Unix ports of Condor except MacOSX, AIX, and Tru64, separate debug symbol files are now supported. This allows meaningful debugging of core files in addition to attaching to stripped executables during runtime.
- *condor_dagman* now prints reports of pending nodes to the *dagman.out*, if it has been waiting more than `DAGMAN_PENDING_REPORT_INTERVAL` seconds without seeing any node job events. This is to help diagnose the problem if *condor_dagman* gets "stuck".
- Optimized the submission of grid-type gt4 grid universe jobs to the remote resource. Submission now takes one operation instead of three.
- The *condor_shadow* will obtain a session key to the *condor_schedd* at the start of the job instead of potentially waiting until the job completes. This reduces the chances of re-running already completed jobs in the event of authentication failures (for instance, if a Kerberos KDC is down or overloaded).

Bugs Fixed:

- On MacOS, Condor is more robust about how it monitors characteristics (such as image size) of a running job. Fixed several issues that would cause the *condor_startd* to exit on MacOS 10.4 running on Intel processors.
- Fixed bug in the local universe where the local universe execute directory was not removed when the job could not start. The most common case was an incorrectly named executable file.

- Fixed a bug that prevented dollar dollar expansions with a default argument that contained forward slashes from expanding properly. An example that now works correctly, but exhibited the incorrect behavior:

```
$$ (SomeVariable:/path/to/file)
```

- The Windows installer now works on MS Vista. Also, it does not pop up any command windows.
- The *condor_ckpt_server* was fixed to honor HIGHPORT and LOWPORT. While the well-known ports for the checkpoint data server have not changed, the helper processes that perform the store and restore (which communicate directly with the standard universe job) now bind to ports within specified ranges. Note that this will limit the number of simultaneous store/restore requests to the number of available ports.
- Fixed a bug in *condor_dagman* that caused recovery/bootstrap mode to be very slow on large DAGs (i.e., ones with hundreds of thousands of nodes).
- Fixed a bug that caused *condor_dagman* to incorrectly deal with VARS lines specifying more than one macro name (this bug was introduced in version 6.8.5).
- Fixed a bug in the configuration macro RANDOM_INTEGER when used as part of a larger expression. The entire configuration value containing the reference to RANDOM_INTEGER was being replaced by the chosen random integer, rather than just having RANDOM_INTEGER() itself be replaced.
- Fixed a bug in the GSI configuration parameters. If GSI_DAEMON_DIRECTORY was set and GRIDMAP was not set, then Condor would look in the wrong location for the GSI private key and mapfile.
- *condor_q* would produce garbage output in its error message when failing to contact the collector specified via **-pool**.
- In Unix only, fixed a file descriptor leak that could cause the *condor_schedd* daemon to crash.
- File transfer failures for spooled jobs no longer result in *condor_schedd* child processes hanging around for 8 hours before finally exiting. Too many such processes occasionally resulted in memory exhaustion.
- Fixed a bug in *condor_dagman*: DIR and ABORT-DAG-ON specifications were not propagated to rescue DAGs.
- Added a workaround for a bug in the Globus GRAM JobManager (http://bugzilla.mcs.anl.gov/globus/show_bug.cgi?id=5467) that can cause very short jobs' standard output and error to be lost.
- Disable GSI authorization callouts for the GridFTP server that Condor starts to perform file transfers for grid-type gt4 grid universe jobs.

Known Bugs:

- Grid universe type GT4 (web services GRAM) does not work properly on Itanium-based machines, because it requires Java 1.5, which is not available on the Itanium (ia_64).

Additions and Changes to the Manual:

- Several updates to the DAGMan documentation (section 2.10).
- Improved the group quota documentation.

Version 6.8.5

Release Notes:

- This release is not fully compatible with the 6.6 series (or anything earlier than that). Specifically, a 6.6 schedd will be rejected when it tries to contact a 6.8.5 startd to make use of a claim.
- The Globus libraries used by Condor now include the following advisory packages:
 - globus_gss_assist-3.23
 - globus_xio-0.35
 - globus_gram_protocol-6.5
 - globus_gass_transfer-2.12

See <http://www.globus.org/toolkit/advisories.html> for details on the bugs fixed by these updated packages. The patch given in Globus Bugzilla 5091 (http://bugzilla.mcs.anl.gov/globus/show_bug.cgi?id=5091) is also included.

New Features:

- A clipped port to x86 Debian 4.0 has been added.
- The functionality embodied in *condor_q -better-analyze* is now available for X86_64 native ports of Condor.
- We now supply distinct, native ports for Mac OS X 10.3 and 10.4.
- There is a new configuration macro COLLECTOR_REQUIREMENTS that may be used to filter out unwanted ClassAd updates. For more information, see section 3.3.16.
- Added a **-f** option to *condor_store_cred*, which generates a pool password file that can be used for the PASSWORD authentication method on Unix Condor installations.

Bugs Fixed:

- The config file entry `HOSTALLOW_DAEMON` is now looked at in addition to `ALLOW_DAEMON`.
- Fixed a bug where under certain conditions Condor's file logging codes would perform a segmentation fault.
- Removed periodic re-indexing of the quill history_vertical table. This should not be needed with the current schema, and it should speed up database re-indexing operations.
- Fixed a bug that would cause the dedicated scheduler to crash, if the *condor_schedd* was suspended or blocked for more than approximately 10 minutes. The most likely cause of a suspension is a *condor_schedd* executable mounted from a remote NFS file system.
- Fixed a bug where if `-lc` was specified multiple times for the compiler when using *condor_compile* (some tools like *pgf90* do this), *condor_compile* would fail to link the application and emit a multiply defined symbol error for many symbols.
- Fixed a bug where Condor erroneously indicates that a scheduler universe's job executable is missing or not executable. This occurred if the scheduler universe job had been submitted with **CopyToSpool = false** in the submit description file, and the user had a umask which prevented the user named `condor` from following the search path to the user-owned executable.
- Fixed a bug that could cause the *condor_schedd* to crash if it received too many matches in one negotiation cycle (more than 1000 on a Linux platform).
- Fixed a bug in which *condor_history* did not honor the **-format** flag properly when Quill is in use.
- Fixed a bug in which a java property that includes surrounding double quote marks caused the detection of a java virtual machine to go awry. The fix, which may change in the future, changes any extra double quotes within a property value to single quotes.
- Fixed a bug in which the *condor_quill* daemon crashed occasionally when the Postgres database server was unavailable.
- The Solaris 9 Condor package can be used under Solaris 10 again. Changes in 6.7.20 broke this compatibility.
- *condor_dagman* now does a better job, especially in recovery mode, of detecting potentially incorrect submit events. Those have Condor IDs not matching what is expected.
- *condor_dagman* now truncates existing node job user log files to zero length, rather than deleting the log files. This prevents breaking the link if a user log file is set up as a link.
- When starting a GridFTP server to handle file transfers for gt4 grid jobs, the *condor_gridmanager* now properly sets the `GLOBUS_TCP_PORT_RANGE` and `GLOBUS_TCP_SOURCE_RANGE` environment variables if appropriate.
- Fixed a bug that could cause a security session to get deleted by the server (for example, the *condor_schedd*) before the client (for example, the *condor_shadow*) was done using it. This bug can be observed as communication failure the next time the client tried to connect to the

server. In some cases, this caused jobs to be re-queued to be run again, because the final update of the job queue failed.

- If a grid job becomes held while it's still submitted to the remote resource and is then removed, the *condor_gridmanager* will now attempt to remove the job from the remote resource before letting it leave the local job queue.
- Fixed a bug in the *condor_c-gahp* that caused it to not use the user's credential for authentication with the remote schedd on some connections.
- The *condor_c-gahp* now properly lists all of the commands it supports in response to the COMMANDS command.
- Fix a bug in how the *condor_c-gahp* updates configuration parameter GSI_DAEMON_NAME to include the job's credential if it has one.
- Removed the 5096-character restriction on the length of DAG macro values (and names) in *condor_dagman*.
- Condor-G will now notice when jobs are missing from the status reports sent by the Grid Monitor. Jobs can disappear for short periods of time under normal circumstances, but a prolonged absence is often a sign of problems on the remote machine. The amount of time that a job can go missing from the Grid Monitor status reports before the *condor_gridmanager* reacts can be set by the new configuration parameter GRID_MONITOR_NO_STATUS_TIMEOUT. The default is 15 minutes.
- *condor_q-analyze* will now print a warning if a job being analyzed is already completed or if a grid universe job being analyzed has already been matched.
- In *condor_shadow*, when forwarding an updated X509 proxy to an executing job, the logic for whether to delegate or copy the proxy (determined by configuration parameter DELEGATE_JOB_GSI_CREDENTIALS) was reversed. The authentication logic for this operation was also incorrect, causing the operation to fail in many instances.
- Made a small improvement to the reliability of Condor's process ancestry tracking under Linux. However, jobs that create children with more than 4096 bytes of environment are still problematic, due to a Linux kernel limitation that prevents reading more than 4k from /proc/*pid*/environ. The only truly reliable way to ensure that Condor is aware of all processes spawned by a Unix job is to use VMX.USER.
- *condor_glidein* option **-run.here** no longer fails when the current working directory is not in PATH.
- *condor_glidein* option **-runtime** would cause runtime errors at startup under some batch systems. The problematic parentheses characters are no longer generated as part of the environment value that is set by this option.
- On rare occasions, the *condor_startd* will compute a negative MIPS rating when performing benchmarks on the machine. This caused the MIPS attribute to disappear from the machine ad. Now, the *condor_startd* ignores these bogus results. The cause of the negative MIPS ratings is still unknown.

- Fixed a bug that caused *condor_dagman* to hang if it processed, in recovery mode, a node for which all submit attempts failed and a POST script was run.
- Fixed a bug that would cause the *condor_negotiator*'s memory usage to grow over time when job or machine ClassAds made use of ClassAd functions that do regular expression matching operations.
- Fixed a bug that was preventing Condor daemons from caching DNS information for hosts authenticated via HOSTALLOW settings (i.e. no strong authentication). The collector, in particular, should spend much less time on IP to host name lookups.
- When a job has an X509 proxy file (as indicated by the X509UserProxy attribute in the job ad), the *condor_starter* now always sets X509_USER_PROXY in the job's environment to point to a copy of that proxy file.
- Fixed several bugs that could cause the *condor_c-gahp* to time out when talking to the *condor_schedd* and falsely report that commands completed successfully. A common result is grid type condor grid universe jobs being placed on hold because the *condor_gridmanager* mistakenly thinks they disappeared from the remote *condor_schedd*'s queue.
- Fixed a bug in Stork which was causing it to write the output and error log files as the wrong user, and read the input file as the wrong user.
- Fixed a bug in Stork which was causing it to kill hung jobs as the wrong user.
- Fixed some possible static buffer overflows related to the transferring of a job's data files.
- Jobs with standard output and error going to the same file should not lose data in the common case.
- Heavily loaded condor daemons (e.g. *condor_schedd*) had a problem when they got behind processing the exit status of child process (e.g. *condor_shadow*). The problem was that the daemon would continue to expect status updates from its child, even after the child had exited, and when the daemon decided that the lack of status updates meant that the child was hung, the daemon would try to kill any process that happened to have the same pid as the child which had already exited. In the case of the schedd, this would also result in the job run attempt being marked as a failure and the job would remain in the queue to run again. Condor no longer activates the "hung child" procedure for jobs which have exited but which have not yet had their exit status processed internally by the daemon.
- For grid-type condor jobs, made the *condor_gridmanager* more tolerant of unexpected responses from the remote *condor_schedd*.
- On HP/UX and AIX, fixed a bug that could cause Condor's process family tracking logic to lose track of processes.
- Fixed a memory error that would cause *condor_q* to sometimes crash when using Quill.
- Fixed a problem where the Windows *condor_credd* would be inaccessible to other Condor components if CREDD_HOST were set to a DNS alias and not the canonical DNS name.

- Fixed a bug in the *condor_shadow* on Windows where it would fail to correctly perform the PASSWORD authentication method.
- The Windows *condor_credd* now uses the configuration parameter CREDD_HOST, if defined, to set its name when advertising itself to the *condor_collector*. Thus, if CREDD_HOST is set to something other than then *condor_credd*'s host name, clients can still locate the daemon.
- Fixed a bug in the *condor_c_gahp* that could cause it to not perform hold, release, or remove commands on jobs in the remote *condor_schedd*.
- Fixed the default value of configuration parameter STARTD_AD_REEVAL_EXPR.

Known Bugs:

- *condor_dagman* incorrectly parses DAG file VARS lines specifying more than one macroname/value pair. You can work around this problem by specifying each macroname/value pair on a separate line. (This bug was introduced in version 6.8.5.)

Version 6.8.4

Release Notes:

- None.

New Features:

- Added new tool *condor_dump_history* which will enable schema migration to future Quill schema versions.
- Quill can now automatically rebuild the indexes on the *PostgreSQL* database tables. Some sites reported that even with auto vacuuming turned on, the indexes on the tables were growing without bounds. Rebuilding the indexes fixes that problem. Rebuilding is disabled by setting the parameter QUILL_SHOULD_REINDEX to False. Re-indexing happens immediately after the history file is purged of old data. So, if Quill is configured to never delete history data, the tables are never re-indexed. Also, *condor_quill* was changed so that the history deletion also happens at start time. This ensures that old history rows are deleted if Quill crashes before the scheduled deletion time.
- Added more information to StarterLog for an error message involved in file transfers:

Download acknowledgment missing attribute: Result.

The extra information is a full dump of the ClassAd that was received, in order to help determine why the expected attribute was not found.

- Added output to the `dagman.out` file documenting when `condor_dagman` shortcuts node retries because of `condor_submit` failures or a helper command failure.

Bugs Fixed:

- Fixed a bug in `condor_q` that only happened when running with a Quill database and using the long `(-l)` option. The bug was introduced in 6.8.3. The bug truncated the output of `condor_q`, and only displayed some of the job attributes.
- Fixed a bug in `condor_submit` that caused standard universe jobs to be unable to open their standard output or standard error, if `should_transfer_files` is **YES** or **IF_NEEDED** in the submit description file.
- Fixed a bug in `condor_glidein` that could cause it to request the queue unknown when submitting its setup job to GRAM, leading to failures.
- The `OnExitRemove` expression generated for DAGMan by `condor_submit_dag` evaluated to `UNDEFINED` for some values of `ExitCode`, causing `condor_dagman` to go on hold.
- Fixed a bug in which garbage values (random bits from memory) were sometimes written to the pool history file in the field representing the backfill state.
- `condor_submit_dag` now generates a submit file (`.condor.sub`) for `condor_dagman` that sends `stdout` and `stderr` to separate files. This has always been recommended, and recent versions of Condor cause `stdout` and `stderr` to overwrite each other if they are directed to the same file.
- Fixed several bugs for grid type **nordugrid** jobs. The `condor_gridmanager` would create an invalid RSL for these jobs and save their output to the wrong location in some cases.
- `condor_glidein` now properly escapes glidein tarball URLs that contain characters that have special meaning to GRAM RSL. It also turns on TCP updates to the `condor_collector`, if they are enabled on the submit machine.
- When using the submit file option `getenv=true`, environment variables containing a newline in their value are no longer inserted into the job's environment. The `condor_schedd` daemon does not allow newlines within ClassAd values, so the attempt to insert such values resulted in failure of job submission and caused the `condor_schedd` daemon to abort.
- Fixed a bug that caused `condor_dagman` to hang if a node with a POST script and retries initially runs but fails, and then has all `condor_submit` attempts fail on the retry.
- Fixed a problem in the Windows installer where the `DAEMON_LIST` parameter would be incorrectly set if the "Join an existing Condor pool" option was selected or the "Submit jobs to Condor pool" option was unchecked. In the first case, a `condor_collector` and `condor_negotiator` would incorrectly be run on the machine. In the second case, a `condor_schedd` would incorrectly be run. The problem exists in all previous 6.8 and 6.9 series releases.

- Fixed a bug in the handling of local universe jobs for a very busy *condor_schedd* daemon. When a local universe job completed, the *condor_starter* might not be able to connect to the *condor_schedd* daemon to update final information about the job, such as the exit status. Under this circumstance, the *condor_starter* would hang indefinitely. The bug is fixed by having the *condor_starter* attempt to retry a few times (with a delay in between each attempt) before exiting with a fatal error. The fatal error causes the job to restart.

Known Bugs:

- Setting `DAGMAN_DELETE_OLD_LOGS` to false can cause *condor_dagman* to have problems (including hanging), especially when running a rescue DAG. If you want to keep your old user log files, the best thing to do is to rename them before each *condor_dagman* run. If you do run with `DAGMAN_DELETE_OLD_LOGS` set to false, check your `dagman.out` file for error messages about submit event Condor IDs not matching the expected value. If you get such an error, you will probably have to *condor_rm* the *condor_dagman* job, remove or rename the old user log file(s) and run the rescue DAG. (Note: this bug also applies to earlier versions of *condor_dagman*.)

Version 6.8.3

Release Notes:

- In this release, the command *condor_q -long* does not work when querying the Quill database. Instead, use the command *condor_q -direct quilld -long*, or use a previous version of *condor_q*.
- Performed a security audit of all places where Condor opens files, to make certain files are opened with a reasonable permission mode and with the `O_EXCL` flag whenever possible.

New Features:

- Added the `JOB_INHERITS_STARTER_ENVIRONMENT` configuration macro. When set to `True`, jobs inherit all environment variables from the *condor_starter*. This is useful for glidein jobs that need to access environment variables from the batch system running the glidein daemons. The default for this configuration macro is `False`, so existing behavior is unchanged. This feature does not apply to standard and pvm universe jobs.

- Changed the default UDP receive buffer for the *condor_collector* from 1M to 10M. This value can be configured with the (existing) `COLLECTOR_SOCKET_BUFSIZE` macro.

NOTE: For some Linux distributions, it may be necessary to configure a larger value than the default; this parameter is `/proc/sys/net/core/rmem_max`. You can see the values that the *condor_collector* actually used by enabling `D_FULLDEBUG` for the *condor_collector* and looking at the log line that looks like this:

Reset OS socket buffer size to 2048k (UDP), 255k (TCP).

- Added a new configuration macro to control the size of the TCP send buffers for the *condor_collector*. This macro used to be the same as `COLLECTOR_SOCKET_BUFSIZE`. The new macro is `COLLECTOR_TCP_SOCKET_BUFSIZE`, and it defaults to 128K.
- Added a clipped port for SuSE Linux Enterprise Server 9 running on the PowerPC architecture. Note the known bug below.
- The *condor_schedd* now maintains a birth date for the job queue. Nothing in Condor currently uses this feature, but future versions of *condor_quill* may require it.
- There is a new configuration file macro `RANDOM_INTEGER(min,max[,step])`. It produces a pseudo-random integer within the range `min` and `max`, inclusive at configuration time.

Bugs Fixed:

- Fixed a deadlock situation between the *condor_schedd* and the *condor_startd* that can significantly impact the *condor_schedd*'s performance. The likelihood of the deadlock increased based upon the number of VMs advertised by the *condor_startd*.
- Fixed a bug reading the user job log on Windows that caused occasional DAGMan confusion. Thanks to Fairview Software, Inc. for both finding the bug and writing a patch.
- Fixed a denial of service problem: Condor daemons no longer freeze for 20 seconds when a client connects to them and then sends no data. This behavior is common with port scanners.
- Fixed a race condition with *condor_quill* caused by *PostgreSQL*'s default transaction isolation level being "read committed". This bug would cause truncated *condor_q* reads when using Quill.
- Fixed a bug where the *condor_ckpt_server* would segfault when turned off with *condor_off -fast*.
- Fixed a bug in the *condor_startd* where it could die with SIGABRT when a *condor_starter* exited under certain rare circumstances. The bug seems to have been most likely to appear on x86_64 Linux machines, but could potentially affect all platforms.
- Fixed a problem with *condor_history* when running with Quill enabled, which caused it to allocate an unbounded amount of memory.
- Fixed a problem with *condor_q* when running with Quill, which caused it to silently truncate the printing of the job queue.
- Fixed a bug in the *condor_gridmanager* that caused the following configuration files parameters to be ignored for grid types *condor* and *nordugrid* jobs: `GRIDMANAGER_RESOURCE_PROBE_INTERVAL`, `GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE`, and `GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE`.
- Fixed a bug in *condor_run* that caused it to abort on non-fatal warnings from *condor_submit* and print incorrect error messages.

- Fixed a bug in the *condor_gridmanager* dealing with grid type gt4 grid universe jobs. If the job's standard output or error was not specified in the job ClassAd, the *condor_gridmanager* would create an improper GRAM RSL string, causing the job to fail.
- Fixed a bug in the *condor_gridmanager* that could cause it to delegate the wrong credential when refreshing the credentials for a grid type gt4 grid universe job.
- The *condor_gridmanager* could get into a state where it would no longer start up Globus jobmanagers for grid type gt2 grid universe jobs, if previous requests failed due to connection errors. This bug has been fixed.
- The *condor_c-gahp* now properly exits when the pipe to its parent goes away. Before, it would fill its log with large amounts of useless messages, before exiting several minutes later.
- Fixed a bug where a problem opening standard input, output, or error, the standard universe might generate an incorrect warning in the *condor_shadow*'s log.
- The *condor_gridmanager* now recovers properly when a proxy refresh fails for a gt2 grid universe job in the stage-out state. Before, the job would become held with a hold reason of "Globus error 3: an I/O operation failed".
- A number of fixes to minor typos and incorrect formatting in Condor's log files.
- When REQUEST_CLAIM_TIMEOUT was reached and the *condor_schedd* failed to contact the *condor_startd* to release the claim, the *condor_schedd* would periodically try releasing the claim indefinitely, possibly resulting in a lengthy communication delay each time.
- Under Windows, Condor daemons such as the *condor_schedd* were sometimes limiting their use of pending connect operations more than they should have. This would result in the message, "file descriptor safety level exceeded".
- *condor_fetchlog* no longer allows or documents the -dagman option. The option's appearance was an error. The option never worked.
- The *condor_schedd* ensures that the initial job queue log file contains a sequence number for use by Quill. This fixes a case in which no sequence number was inserted, because the initial rotation of this (empty) file failed. Quill also now reports exactly what the problem is if it reads a job queue log in this state, rather than simply crashing. This problem has so far only been observed under Windows.
- Fixed a problem on Windows where, when submitting a job with a sandbox (for example, using the -s or -r option to *condor_submit*), an erroneous file permissions check in the *condor_schedd* would result in a failed submission.
- The *condor_startd* would crash shortly after start up if the RANK expression contained any use of the unary minus operator. This patch should also fix any other cases where Condor daemons crashed due to the use of the unary minus operator in ClassAd expressions.
- Stork now writes a terminated event to the user log when it removes a transfer job from its queue because of failures to invoke a transfer module. Without this event, DAGMan would not notice that these jobs had left the queue.

- Fixed a problem where the *condor_schedd* on Windows would incorrectly reject a job if the client provided an `Owner` attribute that was correct but differed in case from the authenticated name. This bug was thought to have been fixed in Condor 6.8.0.
- Fixed problems with *condor_store_cred* behaving strangely when storing or removing a user name that is some initial substring of “condor.pool”. Specifying such a user name would be incorrectly interpreted as equivalent to specifying the `-c` option.
- Fixed a problem with *condor_glidein* spewing lots of text to the screen when checking the status of a job it submitted.
- A new version of the GT4 GAHP is included, with the following changes:
 - A new *axis.jar* from Globus fixes a thread safety bug that can cause lockups in subscriptions for WS notifications. See Globus Bugzilla 4858 (http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=4858).
 - Fixed bugs that caused memory related to destroyed jobs to not be reclaimed in both the client and the server.
 - Removed redundant usage of Secure Message, Secure Conversation, and Transport Security when talking to a WS GRAM service. Now, only Transport Security is used.
- Fixed memory leaks in *condor_quill*.
- Fixed a bug that might have caused *condor_startd* problems launching the *condor_starter* for the standard universe on 64-bit systems.
- Improved Condor’s file transfer. If you request that Condor automatically transfer back your output, it now detects changes better. Previously, it would only transfer back files that had a more recent timestamp than the spool date. Now, it will transfer back any file that has changed in date (including being dated in the past) or changed in size.

Known Bugs:

- SuSE Linux Enterprise Server 9 on PowerPC only: The default Java interpreter on SuSE Linux Enterprise Server 9 running on the PowerPC architecture has compatibility problems with this release of Condor. The problem exhibits itself as the *condor_startd* hanging, never reporting itself to the *condor_collector*. The workaround is to either disable the Java universe (set `JAVA` to an empty string), or disable just-in-time compilation when running in the Java universe with the following configuration setting:

```
JAVA_EXTRA_ARGUMENTS = -Djava.compiler=NONE
```

Version 6.8.2

Release Notes:

- Condor now uses Globus 4.0.3 for GSI, GRAM, and GridFTP support. This includes a patch for the OpenSSL vulnerability detailed in CVE-2006-4339 and http://www.openssl.org/news/secadv_20060905.txt. It also includes fixes for Globus Bugzilla 4689 (http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=4689) and a bug that can cause duplicate UUIDs to be generated for WS GRAM jobs.
- The *condor_schedd* daemon no longer forks separate processes to change ownership of job directories in the spool. Previously on Unix-like systems, this would create a new process before a job started running and after it finished running. Some sites with very busy *condor_schedd* daemons were encountering scaling problems.

New Features:

- Because, by default, the *condor_startd* daemon references the job ClassAd attribute *NumCkpts*, Condor's default configuration will now round up the value of *NumCkpts*, in order to improve matchmaking performance. See the entry on *SCHEDD_ROUND_ATTR* in section 3.3.11.
- Enhanced the RHEL3 x86_64 port of Condor to include the standard universe.
- *condor_submit_dag -f* no longer deletes the *dagman.out* file. *condor_submit_dag* without the *-f* option will now submit a DAGMan run even if the *dagman.out* file exists. In this case, the file will be appended to.
- Added a property to the Windows installer program to determine whether the Condor service will be started after installation. The property name is *STARTSERVICE*, and the default value is "Y".

Bugs Fixed:

- A bug caused the *condor_master* daemon to kill only immediate children within the process tree, upon an abnormal exit of the *condor_master* daemon. The *condor_master* daemon now kills all descendant processes.
- Fixed a bug where if the file system was full, the debugging log files (for example *SchedLog*) would silently lose messages. Now, if the disk is full, the Condor daemons will exit.
- Fixed a bug in the *condor_schedd* daemon that caused it to stop negotiating for grid universe jobs in the case that it decided it could not spawn any new *condor_shadow* processes.
- Added the *ProcessId* class (which more uniquely identifies a process than a PID does) to the *condor_dagman* abort duplicate runs feature. This makes it less likely that a given instance of *condor_dagman* will mistakenly conclude that another instance of *condor_dagman* is already running on the same DAG. Also fixed an unrelated bug in the abort duplicate runs feature that could cause a *condor_dagman* to not abort itself when it should.
- Condor daemons leaked memory (consuming more and more memory over time) when parsing ClassAds that use functions with arguments.

- Fixed a bug in the *condor_starter* daemon, which caused it to look in the wrong place for the job's executable, if `TransferExecutable` was set to `True` in the job ClassAd.
- *condor_history* no longer crashes if `HISTORY` is not defined in the Condor configuration file.
- Fixed an unintentional change to the value of **-Condorlog** in a *condor_dagman* submit description file: it is once again the log file of the first node job.
- Fixed a bug in *condor_q* that would cause *condor_q -hold* or *condor_q -run* to exit with an error on some platforms.
- Fixed a bug on Unix platforms, in which a misconfiguration of `MAIL` would cause the *condor_master* daemon to restart all of its child daemons whenever it tried (and failed) to send e-mail to the administrator.
- Network related error messages have been improved to make debugging easier. For example, when timing out on a read or write operation, the peer's address is now included in the error message.
- An invalid value for `UPDATE_INTERVAL` now causes the *condor_startd* daemon to abort. Previously, it would continue running, but some invalid values (for example, 0) could cause it to stop sending periodic ClassAd updates to the *condor_collector*, even after being reconfigured with a valid value. Only a complete restart of the *condor_startd* daemon was sufficient to get it out of this state.
- Fixed a bug that caused X.509 limited proxies to be delegated as impersonation (i.e. non-limited) proxies. Any authentication attempted with the resulting proxies would fail.
- Fixed a couple bugs that would cause Condor to lose track of some Condor-related processes and subsequently fail to clean up (kill) these processes.
- Fixed a bug that would cause *condor_history* to crash when dealing with rotated history files. Note that history file rotation is turned on by default. (See Section 3.3.3 for descriptions of `ENABLE_HISTORY_ROTATION` and `MAX_HISTORY_ROTATIONS`.)

Known Bugs:

- None.

Version 6.8.1

Release Notes:

- Version 6.8.1 fixes important bugs, some of which have security implications. All users are encouraged to upgrade, and full disclosure of the vulnerabilities will be given at the end of October 2006.

- Condor is now linked against GSI from Globus 4.0.2. This includes a patch for Globus Security Advisories 2006-01 (http://www.globus.org/mail_archive/security-announce/2006/08/msg00000.html) and 2006-02 (http://www.globus.org/mail_archive/security-announce/2006/08/msg00001.html). It also includes a patch for the OpenSSL vulnerability detailed in CVE-2006-4339 and http://www.openssl.org/news/secadv_20060905.txt.
- The PCRE (Perl Compatible Regular Expressions) library used by Condor is now dynamically linked and shipped as a DLL with Condor for Windows, rather than being statically linked.

New Features:

- Added an optional argument to the *condor_dagman* ABORT-DAG-ON command that allows the DAGMan exit code to be specified separately from the node value that causes the abort; also, a DAG can now be aborted on a zero exit code from a node.
- Added the `ALLOW_FORCE_RM` configuration variable. If this expression evaluates to `True`, then an *condor_rm -f* attempt is allowed. If it evaluated to `False`, the attempt is disallowed. The expression is evaluated in the context of the job ClassAd. If not defined, the value defaults to `True`, matching the behavior of previous Condor releases.
- *condor_dagman* will now reject DAGs for which any of the nodes' user job log files are on NFS (because of the unreliability of NFS file locking, this can cause DAGs to fail). This feature can be turned off by setting the `DAGMAN_LOG_ON_NFS_IS_ERROR` configuration macro to `False` (the default is `True`).
- *condor_submit* can now be configured to reject jobs for which the log file is on NFS. To do this, set the `LOG_ON_NFS_IS_ERROR` configuration macro to `True`. The default is that *condor_submit* will issue a warning for a log file on NFS.
- Added the `DAGMAN_ABORT_DUPLICATES` configuration macro, which causes *condor_dagman* to attempt to detect at startup whether another *condor_dagman* is already running on the same DAG; if so, the second *condor_dagman* will abort itself.
- The new configuration variable `NETWORK_MAX_PENDING_CONNECTS` may be used to limit the maximum number of simultaneous network connection attempts. This is primarily relevant to the *condor_schedd* daemon, which may try to connect to large numbers of *condor_startd* daemons when claiming them. The *condor_negotiator* may also connect to large numbers of *condor_startd* daemons when initiating security sessions used for sending MATCH messages. On Unix, the default is to allow up to eighty percent of the process file descriptor limit. On Windows, the default is 1600.
- Added some more debug output to *condor_dagman* to clarify fatal errors.
- The `-format` argument to *condor_q* and *condor_status* can now take an expression in addition to a simple attribute name.
- DRMAA is now available on most Linux platforms, Windows and PPC MacOS.

Bugs Fixed:

- When a large number of jobs (roughly 200 or more) are running from a single *condor_schedd* daemon, and those jobs are using job leases (the default in 6.8), it is possible for the *condor_schedd* daemon to enter a state where it crashes on startup until all of the job leases expire.
- Condor jobs submitted with the `NiceUser` priority were not being matched if the `NEGOTIATOR_MATCHLIST_CACHING` setting was `TRUE` (which is enabled by default).
- Fixed a Quill bug that prevented it from running on Windows. The symptom showed with errors in the QuillLog such as

`POLLING RESULT: ERROR`
- Fixed a bug in Quill where it would cause errors such as

`duplicate key violates unique constraint "history_vertical_pkey"`

in the QuillLog and the *PostgreSQL* log file. These errors triggered a significant slowdown in the performance of Quill and the database. This would only happen when a job attribute changed type from a string type to a numeric type, or vice versa.
- In those unusual cases where Condor is unable to create a new process, it shuts down cleanly, eliminating a small possibility of data corruption.
- Fixed a bug with the `gt4` and `nordugrid` grid universe jobs that caused the `stdout` and `stderr` of a job to not be transferred correctly, if the given file names had absolute paths.
- *condor_dagman* now echos warnings from *condor_submit* and *stork_submit* to the `dagman.out` file.
- Fixed a bug introduced in 6.7.20, causing the *condor_ckpt_server* to exit immediately after starting up, unless Condor's security negotiation was disabled.
- `MAX_<SUBSYS>_LOG` defaults to one Megabyte, even if the setting is missing from the configuration. Previously it was 64 Kilobytes.
- Fixed a bug related to non-blocking connect that could occasionally cause Condor daemons to crash.
- Fixed a rare bug where an exceptionally large query to the *condor_collector* could cause it to crash. The most common cause was a single *condor_schedd* daemon restarting, and trying to recover a large number of job leases at once. More than approximately 250 running jobs on a single *condor_schedd* daemon would be necessary to trigger this bug.
- When using the `JOB_PROXY_OVERRIDE_FILE` configuration parameter, the X.509 proxy will now be properly forwarded for Condor-C jobs.

- Greatly reduced the chance that a Condor-C job in the REMOVED state will be HELD due to an expired proxy or failure to talk to the remote *condor_schedd*.
- Fixed error and debug messages added in Condor version 6.7.20 that incorrectly reported IP and port numbers. These messages were intended to report the peer's address, but they were instead reporting the local address of the network socket.
- Fixed a bug introduced in Condor version 6.7.20 which could cause Condor daemons to die with the message

```
PANIC -- OUT OF FILE DESCRIPTORS
```

The conditions causing this related to failed attempts to send updated status to the *condor_collector* daemon, with both non-blocking updates and security negotiation enabled (the defaults).

- Also fixed a bug in the negotiator with the same effect as above, except it only happened with the configuration setting `NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT=False`.
- Fixed a bug in *condor_schedd* under Solaris that could also cause file descriptors to become exhausted over time when many machines were claimed in a short spans of time (e.g. over 100) and the *condor_schedd* process file descriptor limit was near 256.
- Fixed a bug in *condor_schedd* under Windows that could cause network sockets to be allocated and never released back to the system. The circumstances that could cause this were very rare. The error message in the logs indicating that this problem was happening is

```
ERROR: DuplicateHandle() failed in Sock::set_inheritable
```

In cases where this error message is displayed, the network socket is closed.

- Under some conditions, when making TCP connections, Condor was still trying to connect for the full duration of the operation timeout (often 10 or 20 seconds), even if the connection attempt was refused (for example, because the port being accessed is not accepting connections). Now, the connect operation finishes immediately after the first such failure, allowing the Condor process to continue with other tasks.
- Fixed the problems relating to credential cache problems in the Kerberos authentication mechanism. The current version of Kerberos is 1.4.3.
- Fixed bugs in the SSL authentication mechanism that caused the *condor_schedd* to crash when submitting a job (on Unix) and caused all tools and daemons to crash on Windows when using SSL.
- Some of the binaries required to use Condor-C on Windows were mistakenly not included in previous releases of Condor. This has been fixed.
- Fixed a problem on Windows where the *condor_startd* could fail to include some attributes in its ClassAd. This would result in some jobs incorrectly not being matched to that machine. This only happened if `CREDD_HOST` was defined and Condor daemons on the execute machine were unable to authenticate with the *condor_credd*.

- Fixed a *condor_dagman* bug which had prevented the `$(DAGManJobId)` attribute from being expanded in job submit files (for example, when used as the value to define the **Priority** command).
- Fixed a bug in *condor_submit* that caused parallel universe jobs submitted via Condor-C to become mpi universe jobs.
- Fixed a bug which could cause Condor daemons to hang if they try to write to the standard error stream (`stderr`) on some platforms. In general, this should never happen, but can, due to third party libraries (beyond our control) trying to write error or other messages.
- Fixed *condor_status* to report error messages.
- Fixed a bug in which setting the configuration variable

```
NEGOTIATOR_CONSIDER_PREEMPTION = False
```

caused an incorrect calculation. The fraction of the pool already being claimed by a user was calculated using the wrong total number of *condor_startd* daemons. This could cause some *condor_startd* daemons to remain unclaimed, even when there were jobs available to run on them.

- Fixed a security vulnerability in Condor's FS and FS_REMOTE authentication methods. The vulnerability allowed an attacker to impersonate another user on the system, potentially allowing submission of jobs as a different user. This may allow escalation to root privilege if the Condor binaries and configuration files have improper permissions. The fix is not backwards compatible, which means all daemons and tools using FS authentication must be running Condor 6.8.1 or greater. The same applies to FS_REMOTE; All daemons and tools using FS_REMOTE must be using Condor 6.8.1 or greater. In practice, this means that for FS, all Condor binaries on one host must be version 6.8.1 or greater, but versions can be different from host to host. For FS_REMOTE it means all binaries across all hosts must be 6.8.1 or greater.
- Fixed a couple race conditions in stork and the credd where credential files were possibly created with improper permissions before being set to owner permissions.
- Fixed a bug in the *condor_gridmanager* that caused it to delegate 12-hour proxies for grid-type gt4 jobs and then not refresh them.
- Fixed a bug in the *condor_gridmanager* that caused a directory needed for staging-in of grid-type gt4 job files to be removed when the *condor_Gridmanager* exited, causing the stage-in to fail.
- Fixed a bug that caused the *checkpoint server* to restart because of (ostensibly) getting an unexpected `errno` from `select()`.
- Fixed a bug on Windows where setting **output** or **error** to a relative or absolute path (as opposed to a simple file name without path information) would not work properly.

- History file rotation did not previously work on Windows because the name of a rotated files would contain an ISO 8601 extended format timestamp, which contains colon characters. The naming convention for rotated files has been modified to use ISO 8601 basic format, avoiding this problem.
- The CLAIMTOBE authentication method (which is inherently insecure and should only be used for testing or other special circumstances) previously would authenticate without providing the “domain” portion of the user name. As an example, a user would be authenticated as simply “user” rather than “user@cs.wisc.edu”. This problem has been fixed, but the new protocol is not backwards compatible so the fix is turned off by default. Correct behavior can be enabled by setting the SEC_CLAIMTOBE_INCLUDE_DOMAIN parameter to True.
- Fixed a bug with the NEGOTIATOR_MATCHLIST_CACHING that would cause very low-priority jobs (like jobs submitted with `nice_user=True`) to not match even if resources were available.
- Fixed a buffer overflow that could crash the *condor_negotiator*.
- SCHEDD_ROUND_ATTR_<xxxx> preserves the value being rounded up when it is a multiple of the power of 10 specified for rounding. Previously, the value would be incremented; now it remains the same. For example, if SCHEDD_ROUND_ATTR_<xxxx>=2 and the value being rounded up is 100, it now remains 100, rather than being incremented to 200.
- Fixed *condor_updates_stats* to report it’s version number correctly.

Known Bugs:

- The **-completedsince** option to *condor_history* works when Quill is enabled. The behavior of *condor_history* **-completedsince** is undefined when Quill is *not* enabled.

Version 6.8.0

Release Notes:

- The default configuration for Condor now requires that `HOSTALLOW_WRITE` be explicitly set. Condor will refuse to start if the default configuration is used unmodified. Existing installations should not need to change anything. For those who desire the earlier default, you can set it to “*”, but note that this is potentially a security hole allowing anyone to submit jobs or machines to your pool.
- Most Linux distributions are now supported using dynamically linked binaries built on a RedHat Enterprise Linux 3 machine. Recent security patches to a number of Linux distributions have rendered the binaries built on RedHat 9 machines ineffective. The download pages have been changed to reflect this, but Linux users should be aware of this change. The recommended download for most x86 Linux users is now: `condor-6.8.0-linux-x86-rhel3-dynamic.tar.gz`.

- Some log messages have been clarified or moved to different debugging levels. For example, certain messages that looked like errors were printed to `D_ALWAYS`, even though nothing was wrong and the system was behaving as expected.
- The new features and bugs fixed in the rest of this section only refer to changes made since the 6.7.20 release, not the last stable release (6.6.11). For a complete list of changes since 6.6.11, read the 6.7 version history in section ?? on page ??.

New Features:

- Version 1.4 of the Condor DRMAA libraries are now included with the Condor release. For more information about DRMAA, see section 4.5.2 on page 480.
- Version 1.0.15 of the Condor GAHP is now used for Condor-G and Condor-C.
- Added the **-outfile.dir** command-line argument to *condor_submit_dag*. This allows you to change the directory in which *condor_dagman* writes the `dagman.out` file.
- Added a new **-summary** (also **-s**) option to the *condor_update_stats* tool. If enabled, this prevents it from displaying the entire history for each machine and only displays the summary info.

Bugs Fixed:

- Fixed a number of potential static buffer overflows in various Condor daemons and libraries.
- Fixed some small memory leaks in the *condor_startd*, *condor_schedd*, and a potential leak that effected all Condor daemons.
- Fixed a bug in Quill which caused it to crash when certain long attributes appeared in a job ad.
- The startd would crash after a reconfig if the address of a collector had not been resolved since the previous reconfig (e.g. because DNS was down during that time).
- Once a Condor daemon failed to lookup the IP address of the collector (e.g. because DNS was down), it would fail to contact the collector from that time until the next reconfig. Now, each time Condor tries to contact the collector, it generates a fresh DNS query if the previous attempt failed.
- When using Condor-C or the **-s** or **-r** command-line options to *condor_submit*, the job's standard output and error would be placed in the job's initial working directory, even if the job ad said to place them in a different directory.
- Greatly sped up the parsing of large DAGs (by a factor of 50 or so) by using a hash table instead of linear search to find DAG nodes.
- Fixed a bug in *condor_dagman* that caused an `EXECUTABLE_ERROR` event from a node job to abort the DAG instead of just marking the relevant node as failed.

- Fixed a bug in *condor_collector* that caused it to discard machine ads that don't have an IP address field (either *StartdIpAddr* or *STARTD_IP_ADDR*). The *condor_startd* will always produce a *StartdIpAddr* field, but machine ads published through *condor_advertise* may not.
- When using *BIND_ALL_INTERFACES* on a dual-homed machine, a bug introduced in 6.7.18 was causing Condor daemons to sometimes incorrectly report their IP addresses, which could cause jobs to fail to start running.
- Made the event checking in *condor_dagman* less strict: added the new "allow duplicate events" value to the *DAGMAN_ALLOW_EVENTS* macro (this value is part of the default); 16 value now also allows terminate event before submit; changed "allow all events" to "allow almost all events" (all except "run after terminal event"), so it is more useful.
- *condor_dagman* and *condor_submit_dag* now report **-NoEventChecks** as ignored rather than deprecated.
- Fixed a bug in the *condor_dagman* **-maxidle** feature: a shadow exception event now puts the corresponding job into the idle state in *condor_dagman*'s internal count.
- Fixed a problem on Windows where daemons would sometimes crash when dealing with UNC path names.
- Fixed a problem where the *condor_schedd* on Windows would incorrectly reject a job if the client provided an *Owner* attribute that was correct but differed in case from the authenticated name.
- Fixed a *condor_startd* crash introduced in version 6.7.20. This crash would appear if an execute machine was matched for preemption but then not claimed in time by the appropriate *condor_schedd*.
- Resolved an issue where the *condor_startd* was unable to clean up jobs' execute directories on Windows when the *condor_master* was started from the command line rather than as a service.
- Added more patches to Condor's DRMAA interface to make it more compatible with Sun Grid Engine's DRMAA interface.
- Removed the unused *D_UPDOWN* debug level and added the *D_CONFIG* debug level.
- Fixed a bug that caused *condor_q* with the **-l** or **-xml** arguments to print out duplicate attributes when using Quill.
- Fixed a bug that prevented Condor-C jobs (universe grid jobs of type condor) from submitting correctly if *QUEUE_ALL_USERS_TRUSTED* is set to True.
- Fixed a bug that could cause the *condor_negotiator* to crash if the pool contains several different versions of the *condor_schedd* and in the config file *NEGOTIATOR_MATCHLIST_CACHING* is set to True.
- Changed the default value for config file entry *NEGOTIATOR_MATCHLIST_CACHING* from False to True. When set to True, this will instruct the negotiator to safely cache data in order to improve matchmaking performance.

- The *condor_master* now recognizes *condor_quill* as a valid Condor daemon without any manual configuration on the part of site administrators. This simplifies the configuration changes required to enable Quill.
- Fixed a rare bug in the *condor_starter* where if there was a failure transferring job output files back to the submitting host, it could hang indefinitely, and the job appeared as if it was continuing to run.

Known Bugs:

- The **-completedsince** option to *condor_history* works when Quill is enabled. The behavior of *condor_history -completedsince* is undefined when Quill is *not* enabled.

CHAPTER

NINE

Command Reference Manual (man pages)

cleanup_release

uninstall a previously installed software release installed by *install_release*

Synopsis

cleanup_release [-help]

cleanup_release *install-log-name*

Description

cleanup_release uninstalls a previously installed software release installed by *install_release*. The program works through the install log in reverse order, removing files as it goes. Each delete is logged in the install log to allow recovery from a crash. The install log name is provided as the *install-log-name* argument to this program.

Options

-help Display brief usage information and exit.

Exit Status

cleanup_release will exit with a status of 0 (zero) upon success, and non-zero otherwise.

See Also

install_release (on page 856).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_advertise

Send a ClassAd to the *condor_collector* daemon

Synopsis

condor_advertise [-help | -version] [-pool *centralmanagerhostname[:portname]*] [-debug] [-tcp]
update-command [*classad-filename*]

Description

condor_advertise sends a ClassAd to the *condor_collector* daemon on the central manager machine. The required argument *update-command* says what daemon type's ClassAd is to be updated. The optional argument *classad-filename* is the file from which the ClassAd should be read. If *classad-filename* is omitted or is "-", then the ClassAd is read from standard input.

The *update-command* may be one of the following strings:

UPDATE_STARTD_AD

UPDATE_SCHEDD_AD

UPDATE_MASTER_AD

UPDATE_GATEWAY_AD

UPDATE_CKPT_SRVR_AD

UPDATE_NEGOTIATOR_AD

UPDATE_HAD_AD

UPDATE_AD_GENERIC

UPDATE_SUBMITTOR_AD

UPDATE_COLLECTOR_AD

UPDATE_LICENSE_AD

UPDATE_STORAGE_AD

condor_advertise can also be used to invalidate and delete ClassAds currently held by the *condor_collector* daemon. In this case the *update-command* will be one of the following strings:

INVALIDATE_STARTD_ADS

INVALIDATE_SCHEDD_ADS
INVALIDATE_MASTER_ADS
INVALIDATE_GATEWAY_ADS
INVALIDATE_CKPT_SRVRS_ADS
INVALIDATE_NEGOTIATOR_ADS
INVALIDATE_HAD_ADS
INVALIDATE_ADS_GENERIC
INVALIDATE_SUBMITTOR_ADS
INVALIDATE_COLLECTOR_ADS
INVALIDATE_LICENSE_ADS
INVALIDATE_STORAGE_ADS

For any of these INVALIDATE commands, the ClassAd in the required file consists of three entries. The file contents will be similar to:

```
MyType = "Query"  
TargetType = "Machine"  
Requirements = Name == "condor.example.com"
```

The definition for MyType is always Query. TargetType is set to the MyType of the ad to be deleted. This MyType is DaemonMaster for the *condor_master* ClassAd, Machine for the *condor_startd* ClassAd, Scheduler for the *condor_schedd* ClassAd, and Negotiator for the *condor_negotiator* ClassAd. Requirements is an expression evaluated within the context of ads of TargetType. When Requirements evaluates to True, the matching ad is invalidated. A full example is given below.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portname]* Specify a pool by giving the central manager's host name and an optional port number. The default is the COLLECTOR_HOST specified in the configuration file.

-tcp Use TCP for communication. Without this option, UDP is used.

-debug Print debugging information as the command executes.

General Remarks

The job and machine ClassAds are regularly updated. Therefore, the result of *condor_advertise* is likely to be overwritten in a very short time. It is unlikely that either Condor users (those who submit jobs) or administrators will ever have a use for this command. If it is desired to update or set a ClassAd attribute, the *condor_config_val* command is the proper command to use.

For those administrators who do need *condor_advertise*, you can optionally include these attributes:

DaemonStartTime - The time the service you are advertising started running. Measured in seconds since the Unix epoch.

UpdateSequenceNumber - An integer that begins at 0 and increments by one each time you re-advertise the same ad.

If both of the above are included, the *condor_collector* will automatically include the following attributes:

UpdatesTotal - The actual number of advertisements for this daemon that the *condor_collector* has seen.

UpdatesLost - The number of advertisements that for this daemon that the *condor_collector* expected to see, but did not.

UpdatesSequenced - The total of UpdatesTotal and UpdatesLost.

UpdatesHistory - See COLLECTOR_DAEMON_HISTORY_SIZE in section 3.3.16.

Examples

Assume that a machine called condor.example.com is turned off, yet its *condor_startd* ClassAd does not expire for another 20 minutes. To avoid this machine being matched, an administrator chooses to delete the machine's *condor_startd* ClassAd. Create a file (called *remove_file* in this example) with the three required attributes:

```
MyType = "Query"
TargetType = "Machine"
Requirements = Name == "condor.example.com"
```

This file is used with the command:

```
% condor_advertise INVALIDATE_STARTD_ADS remove_file
```

Exit Status

condor_advertise will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_check_userlogs

Check user log files for errors

Synopsis

condor_check_userlogs *UserLogFile1* [*UserLogFile2* ... *UserLogFileN*]

Description

condor_check_userlogs is a program for checking a user log or set of users logs for errors. Output includes an indication that no errors were found within a log file, or a list of errors such as an execute or terminate event without a corresponding submit event, or multiple terminated events for the same job.

condor_check_userlogs is especially useful for debugging *condor_dagman* problems. If *condor_dagman* reports an error it is often useful to run *condor_check_userlogs* on the relevant log files.

Exit Status

condor_check_userlogs will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_checkpoint

send a checkpoint command to jobs running on specified hosts

Synopsis

condor_checkpoint [-help | -version]

condor_checkpoint [-debug] [-name *name* | *name* | -addr "<*a.b.c.d:port*>" | "<*a.b.c.d:port*>" ...] [-all]

condor_checkpoint [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *name*] [-addr "<*a.b.c.d:port*>"] ... [| -all]

Description

condor_checkpoint sends a checkpoint command to a set of machines within a single pool. This causes the startd daemon on each of the specified machines to take a checkpoint of any running job that is executing under the standard universe. The job is temporarily stopped, a checkpoint is taken, and then the job continues. If no machine is specified, then the command is sent to the machine that issued the *condor_checkpoint* command.

The command sent is a periodic checkpoint. The job will take a checkpoint, but then the job will immediately continue running after the checkpoint is completed. *condor_vacate*, on the other hand, will result in the job exiting (vacating) after it produces a checkpoint.

If the job being checkpointed is running under the standard universe, the job produces a checkpoint and then continues running on the same machine. If the job is running under another universe, or if there is currently no Condor job running on that host, then *condor_checkpoint* has no effect.

There is generally no need for the user or administrator to explicitly run *condor_checkpoint*. Taking checkpoints of running Condor jobs is handled automatically following the policies stated in the configuration files.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "<*a.b.c.d:port*>" Send the command to a machine's master located at "<*a.b.c.d:port*>"

"<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-all Send the command to all machines in the pool

Exit Status

`condor_checkpoint` will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To send a `condor_checkpoint` command to two named machines:

```
% condor_checkpoint robin cardinal
```

To send the `condor_checkpoint` command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_checkpoint -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_chirp

Access files or job ClassAd from an executing job

Synopsis

condor_chirp [-help]

condor_chirp fetch *RemoteFileName LocalFileName*

condor_chirp put [-mode *mode*] [-perm *UnixPerm*] *LocalFileName RemoteFileName*

condor_chirp remove *RemoteFileName*

condor_chirp get_job_attr *JobAttributeName*

condor_chirp set_job_attr *JobAttributeName AttributeValue*

condor_chirp ulog *Message*

Description

condor_chirp is run from a user job while executing. It accesses files or job ClassAd attributes on the submit machine. Files can be read, written or removed. Job attributes can be read, and most attributes can be updated.

Descriptions using the terms *local* and *remote* are given from the point of view of the executing program.

If the input file name for **put** is a dash, *condor_chirp* uses standard input as the source. If the output file name for **fetch** is a dash, *condor_chirp* writes to standard output instead of a local file.

Jobs that use *condor_chirp* must have the attribute `WantIOProxy` set to `True` in the job ad. To do this, place

```
+WantIOProxy = true
```

in the submit description file for the job.

condor_chirp only works for jobs run in the vanilla, mpi, parallel and java universes.

The optional **-mode** *mode* argument is one or more of the following characters describing the *RemoteFileName* file.

- w: open for writing

- a: force all writes to append
- t: truncate before use
- c: create the file, if it does not exist
- x: fail if 'c' is given, and the file already exists

The optional **-perm** *UnixPerm* argument describes the file access permissions in a Unix format (for example, 660).

Options

-help Display usage information and exit.

fetch Copy the *RemoteFileName* from the submit machine to the execute machine.

remove Remove the *RemoteFileName* file from the submit machine.

put Copy the *LocalFileName* from the execute machine to the submit machine. Perm is the unix permission to open the file with.

get_job_attr Prints the named job ClassAd attribute to standard output.

set_job_attr Sets the named job ClassAd attribute with the given attribute value.

ulog Appends a message to the job's user log.

Examples

To copy a file from the submit machine to the execute machine while the user job is running, run

```
% condor_chirp fetch remotefile localfile
```

To print to standard output the value of the `Requirements` expression from within a running job, run

```
% condor_chirp get_job_attr Requirements
```

Note that the remote (submit-side) directory path is relative to the submit directory, and the local (execute-side) directory is relative to the current directory of the running program.

To append the word "foo" to a file on the submit machine, run

```
% echo foo | condor_chirp put -mode wat - RemoteFile
```

To append the message "Hello World" to the user log, run

```
% condor_chirp ulog "Hello World"
```

Exit Status

condor_chirp will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_cod

manage COD machines and jobs

Synopsis

condor_cod [-help | -version]

condor_cod *request* [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
 | [-addr "<a.b.c.d:port>"] [[-help | -version] | [-debug | -timeout N | -classad file]
][-requirements *expr*] [-lease N]

condor_cod *release* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]][-fast]

condor_cod *activate* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]
][-keyword *string* | -jobad *filename* | -cluster N | -proc N | -requirements *expr*]

condor_cod *deactivate* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]
][-fast]

condor_cod *suspend* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]]

condor_cod *renew* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]]

condor_cod *resume* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]]

condor_cod *delegate_proxy* -id *ClaimID* [[-help | -version] | [-debug | -timeout N | -classad file]
][-x509proxy *ProxyFile*]

Description

condor_cod issues commands that manage and use COD claims on machines, given proper authorization.

Instead of specifying an argument of *request*, *release*, *activate*, *deactivate*, *suspend*, *renew*, or *resume*, the user may invoke the *condor_cod* tool by appending an underscore followed by one of these arguments. As an example, the following two commands are equivalent:

```
condor_cod release -id "<128.105.121.21:49973>#1073352104#4"
```

```
condor_cod_release -id "<128.105.121.21:49973>#1073352104#4"
```

To make these extended-name commands work, hard link the extended name to the *condor_cod* executable. For example on a Unix machine:

```
ln condor_cod_request condor_cod
```

The *request* argument gives a claim ID, and the other commands (*release*, *activate*, *deactivate*, *suspend*, and *resume*) use the claim ID. The claim ID is given as the last line of output for a *request*, and the output appears of the form:

```
ID of new claim is: "<a.b.c.d:portnumber>#x#y"
```

An actual example of this line of output is

```
ID of new claim is: "<128.105.121.21:49973>#1073352104#4"
```

Also see section 4.3 for more a complete description of COD.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr "<a.b.c.d:port>" Send the command to a machine located at "<a.b.c.d:port>"

-lease *N* For the **request** of a new claim, automatically release the claim after *N* seconds.

request Create a new COD claim

release Relinquish a claim and kill any running job

activate Start a job on a given claim

deactivate Kill the current job, but keep the claim

suspend Suspend the job on a given claim

renew Renew the lease to the COD claim

resume Resume the job on a given claim

delegate_proxy Delegate an X509 proxy for the given claim

General Remarks

Examples

Exit Status

condor_cod will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_cold_start

install and start Condor on this machine

Synopsis

condor_cold_start -help

condor_cold_start [-basedir *directory*] [-force] [-setuponly | -runonly] [-arch *architecture*] [-site *repository*] [-localdir *directory*] [-runlocalconfig *file*] [-logarchive *archive*] [-spoolarchive *archive*] [-execarchive *archive*] [-filelock] [-pid] [-artifact *filename*] [-wget] [-globuslocation *directory*] [-configfile *file*]

Description

condor_cold_start installs and starts Condor on this machine, setting up or using a predefined configuration. In addition, it has the functionality to determine the local architecture if one is not specified. Additionally, this program can install pre-made log, execute, and/or spool directories by specifying the archived versions.

Options

-arch *architecturestr* Use the given *architecturestr* to fetch the installation package. The string is in the format:

<condor_version>-<machine_arch>-<os_name>-<os_version>

(for example 6.6.7-i686-Linux-2.4). The portion of this string <condor_version> may be replaced with the string "latest" (for example, latest-i686-Linux-2.4) to substitute the most recent version of Condor.

-artifact *filename* Use *filename* for name of the artifact file used to determine whether the *condor_master* daemon is still alive.

-basedir *directory* The directory to install or find the Condor executables and libraries. When not specified, the current working directory is assumed.

-execarchive *archive* Create the Condor execute directory from the given *archive* file.

-filelock Specifies that this program should use a POSIX file lock midwife program to create an artifact of the birth of a *condor_master* daemon. A file lock undertaker can later be used to

determine whether the *condor_master* daemon has exited. This is the preferred option when the user wants to check the status of the *condor_master* daemon from another machine that shares a distributed file system that supports POSIX file locking, for example, AFS.

-force Overwrite previously installed files, if necessary.

-globuslocation *directory* The location of the globus installation on this machine. When not specified `/opt/globus` is the directory used. This option is only necessary when other options of the form **-*archive** are specified.

-help Display brief usage information and exit.

-localdir *directory* The directory where the Condor `log`, `spool`, and `execute` directories will be installed. Each running instance of Condor must have its own local directory.

-logarchive *archive* Create the Condor log directory from the given *archive* file.

-pid This program is to use a unique process id midwife program to create an artifact of the birth of a *condor_master* daemon. A unique pid undertaker can later be used to determine whether the *condor_master* daemon has exited. This is the default option and the preferred method to check the status of the *condor_master* daemon from the same machine it was started on.

-runlocalconfig *file* A special local configuration file bound into the Condor configuration at runtime. This file only affects the instance of Condor started by this command. No other Condor instance sharing the same global configuration file will be affected.

-runonly Run Condor from the specified installation directory without installing it. It is possible to run several instantiations of Condor from a single installation.

-setuponly Install Condor without running it.

-site *repository* The ftp, http, gsiftp, or mounted file system directory where the installation packages can be found (for example, `www.cs.example.edu/packages/coldstart`).

-spoolarchive *archive* Create the Condor spool directory from the given *archive* file.

-wget Use *wget* to fetch the `log`, `spool`, and `execute` directories, if other options of the form **-*archive** are specified. *wget* must be installed on the machine and in the user's path.

-configfile *file* A required option to specify the Condor configuration file to use for this installation. This file can be located on an http, ftp, or gsiftp site, or alternatively on a mounted file system.

Exit Status

condor_cold_start will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

Examples

To start a Condor installation on the current machine, using <http://www.example.com/Condor/deployment> as the installation site:

```
% condor_cold_start \  
-configfile http://www.example.com/Condor/deployment/condor_config.mobile \  
-site http://www.example.com/Condor/deployment
```

Optionally if this instance of Condor requires a local configuration file *condor_config.local*:

```
% condor_cold_start \  
-configfile http://www.example.com/Condor/deployment/condor_config.mobile \  
-site http://www.example.com/Condor/deployment \  
-runlocalconfig condor_config.local
```

See Also

condor_cold_stop (on page 692), *filelock_midwife* (on page 852), *uniq_pid_midwife* (on page 874).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_cold_stop

reliably shut down and uninstall a running Condor instance

Synopsis

condor_cold_stop -help

condor_cold_stop [-force] [-basedir *directory*] [-localdir *directory*] [-runlocalconfig *file*]
 [-cleaninstall] [-cleanlocal] [-stop] [-logarchive *archive*] [-spoolarchive *archive*]
 [-execarchive *archive*] [-filelock] [-pid] [-artifact *file*] [-nogurl] [-globuslocation *directory*]
 -configfile *file*

Description

condor_cold_stop reliably shuts down and uninstall a running Condor instance. This program first uses *condor_local_stop* to reliably shut down the running Condor instance. It then uses *condor_cleanup_local* to create and store archives of the log, spool, and execute directories. Its last task is to uninstall the Condor binaries and libraries using *cleanup_release*.

Options

- artifact *file*** Uses *file* as the artifact file to determine whether the *condor_master* daemon is still alive.
- basedir *directory*** Directory where the Condor installation can be found. When not specified, the current working directory is assumed.
- cleaninstall** Remove the Condor installation. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.
- cleanlocal** The program will remove the log, spool, exec directories for this Condor instance. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.
- configfile *file*** The same configuration file path given to *condor_cold_start*. This program assumes the file is in the installation directory or the current working directory.

- execarchive *archive*** The program will create a tar'ed and gzip'ed archive of the `execute` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.
- filelock** Determine whether the `condor_master` daemon has exited using a file lock undertaker. This option must match the corresponding option given to `condor_cold_start`.
- force** Ignore the status of the `condor_schedd` daemon (whether it has jobs in the queue or not) when shutting down Condor.
- globuslocation *directory*** The directory containing the Globus installation. This option is required if any of the options of the form **-*archive** are used, and Globus is not installed in `/opt/globus`.
- localdir *directory*** Directory where the `log`, `spool`, and `execute` directories are stored for this running instance of Condor. Required if the **-cleanlocal** option is specified.
- logarchive *archive*** The program will create a tar'ed and gzip'ed archive of the `log` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.
- nogurl** Do not use `globus-url-copy` to store the archives. This implies that the archives can only be stored on mounted file systems.
- pid** Determine whether the `condor_master` daemon has exited using a unique process id undertaker. This option must match the corresponding option given to `condor_cold_start`.
- runlocalconfig *file*** Bind *file* into the configuration used by this instance of Condor. This option should be the one provided to `condor_cold_start`.
- spoolarchive *archive*** The program will create a tar'ed and gzip'ed archive of the `spool` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.
- stop** The program will shut down this running instance of Condor. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

Exit Status

`condor_cold_stop` will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

Examples

To shut down a Condor instance on the target machine:

```
% condor_cold_stop -configfile condor_config.mobile
```

To shutdown a Condor instance and archive the log directory:

```
% condor_cold_stop -configfile condor_config.mobile \  
-logarchive /tmp/log.tar.gz
```

See Also

condor_cold_start (on page 689), *filelock_undertaker* (on page 854), *uniq_pid_undertaker* (on page 876).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_compile

create a relinked executable for submission to the Standard Universe

Synopsis

condor_compile *cc* | *CC* | *gcc* | *f77* | *g++* | *ld* | *make* | ...

Description

Use *condor_compile* to relink a program with the Condor libraries for submission into Condor's Standard Universe. The Condor libraries provide the program with additional support, such as the capability to checkpoint, which is required in Condor's Standard Universe mode of operation. *condor_compile* requires access to the source or object code of the program to be submitted; if source or object code for the program is not available (i.e. only an executable binary, or if it is a shell script), then the program must be submitted into Condor's Vanilla Universe. See the reference page for *condor_submit* and/or consult the "Condor Users and Administrators Manual" for further information.

To use *condor_compile*, simply enter "condor_compile" followed by whatever you would normally enter to compile or link your application. Any resulting executables will have the Condor libraries linked in. For example:

```
condor_compile cc -O -o myprogram.condor file1.c file2.c ...
```

will produce a binary "myprogram.condor" which is relinked for Condor, capable of checkpoint/migration/remote-system-calls, and ready to submit to the Standard Universe.

If the Condor administrator has opted to fully install *condor_compile*, then *condor_compile* can be followed by practically any command or program, including *make* or shell-script programs. For example, the following would all work:

```
condor_compile make

condor_compile make install

condor_compile f77 -O mysolver.f

condor_compile /bin/csh compile-me-shellscript
```

If the Condor administrator has opted to only do a partial install of *condor_compile*, then you are restricted to following *condor_compile* with one of these programs:

```
cc (the system C compiler)

acc (ANSI C compiler, on Sun systems)

c89 (POSIX compliant C compiler, on some systems)

CC (the system C++ compiler)

f77 (the system FORTRAN compiler)

gcc (the GNU C compiler)

g++ (the GNU C++ compiler)

g77 (the GNU FORTRAN compiler)

ld (the system linker)

f90 (the system FORTRAN 90 compiler)
```

NOTE: If you use explicitly call “ld” when you normally create your binary, simply use:

```
condor_compile ld <ld arguments and options>
```

instead.

NOTE: f90 (FORTRAN 90) is only supported on Solaris and Digital Unix.

Exit Status

condor_compile is a script that executes specified compilers and/or linkers. If an error is encountered before calling these other programs, *condor_compile* will exit with a status value of 1 (one). Otherwise, the exit status will be that given by the executed program.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_config_bind

bind together a set of configuration files

Synopsis

condor_config_bind -help

condor_config_bind -o outputfile configfile1 configfile2 [configfile3...]

Description

condor_config_bind dynamically binds two or more Condor configuration files through the use of a new configuration file. The purpose of this tool is to allow the user to dynamically bind a local configuration file into an already created, and possible immutable, configuration file. This is particularly useful when the user wants to modify a configuration but cannot actually make any changes to the global configuration file (even to change the list of local configuration files). This program does not modify the given configuration files. Rather, it creates a new configuration file that specifies the given configuration files as local configuration files.

Condor evaluates each of the configuration files in the given command-line order (left to right). A value defined in two or more of the configuration files results in the last one evaluated defining the value. It overrides any others. To bind a new local configuration into a global configuration, specify the local configuration second within the command-line ordering.

Options

configfile1 First configuration file to bind.

configfile2 Second configuration file to bind.

configfile3... An optional list of other configuration files to bind.

-help Display brief usage information and exit

-o *output_file* Specifies the file name where this program should output the binding configuration.

Exit Status

condor_config_bind will exit with a status value of 0 (zero) upon success, and non-zero on error.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_config_val

Query or set a given Condor configuration variable

Synopsis

condor_config_val [*options*] *variable*...

condor_config_val [*options*] **-set** *string*...

condor_config_val [*options*] **-rset** *string*...

condor_config_val [*options*] **-unset** *variable*...

condor_config_val [*options*] **-runset** *variable*...

condor_config_val [*options*] **-tilde**

condor_config_val [*options*] **-owner**

condor_config_val [*options*] **-config**

condor_config_val [*options*] **-dump**

condor_config_val [*options*] **-verbose** *variable*...

Description

condor_config_val can be used to quickly see what the current Condor configuration is on any given machine. Given a list of variables, *condor_config_val* will report what each of these variables is currently set to. If a given variable is not defined, *condor_config_val* will halt on that variable, and report that it is not defined. By default, *condor_config_val* looks in the local machine's configuration files in order to evaluate the variables.

condor_config_val can also be used to quickly set configuration variables for a specific daemon on a given machine. Each daemon remembers settings made by *condor_config_val*. The configuration file is not modified by this command. Persistent settings remain when the daemon is restarted. Runtime settings are lost when the daemon is restarted. In general, modifying a host's configuration with *condor_config_val* requires the CONFIG access level, which is disabled on all hosts by default. Administrators have more fine-grained control over which access levels can modify which settings. See section 3.6.1 on page 282 for more details.

NOTE: The changes will not take effect until you perform a *condor_reconfig*.

NOTE: It is generally wise to test a new configuration on a single machine to ensure you have no syntax or other errors in the configuration before you reconfigure many machines. Having bad syntax or invalid configuration settings is a fatal error for Condor daemons, and they will exit. Far

better to discover such a problem on a single machine than to cause all the Condor daemons in your pool to exit.

Options

-name *machine_name* Query the specified machine's *condor_master* daemon for its configuration.

-pool *centralmanagerhostname[:portnumber]* Use the given central manager and an optional port number to find daemons.

-address *<ip:port>* Connect to the given ip/port.

-master | -schedd | -startd | -collector | -negotiator The daemon to query (if not specified, master is default).

-set *string* . . . Set a persistent config file entry. The string must be a single argument, so you should enclose it in double quotes. The string must be of the form "variable = value".

-rset *string* . . . Set a runtime config file entry. See the description for **-set** for details about the string to use.

-unset *variable* . . . Unset a persistent config file variable.

-runset *variable* . . . Unset a runtime config file variable.

-tilde Return the path to the Condor home directory.

-owner Return the owner of the *condor_config_val* process.

-config Print the current configuration files in use.

-dump Returns a list of all of the defined macros in the configuration files found by *condor_config_val*, along with their values. If the **-verbose** is supplied as well, then the specific configuration file which defined each macro, along with the line number of its definition is also printed. NOTE: The output of this argument is likely to change in a future revision of Condor.

-verbose variable... Returns the configuration file name and line number where a configuration variable is defined.

variable... The variables to query.

Exit Status

condor_config_val will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To request the schedd daemon on host perdita to give the value of the MAX_JOBS_RUNNING configuration variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

To request the schedd daemon on host perdita to set the value of the MAX_JOBS_RUNNING configuration variable to the value 10.

```
% condor_config_val -name perdita -schedd -set "MAX_JOBS_RUNNING = 10"
Successfully set configuration "MAX_JOBS_RUNNING = 10" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects the change implemented:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
10
```

To set the configuration variable MAX_JOBS_RUNNING back to what it was before the command to set it to 10:

```
% condor_config_val -name perdita -schedd -unset MAX_JOBS_RUNNING
Successfully unset configuration "MAX_JOBS_RUNNING" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita  
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects that variable has gone back to its value before initial set of the variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING  
500
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_configure

Configure or install Condor

Synopsis

condor_configure or *condor_install* [**--help**]

condor_configure or *condor_install* [**--install**[=<path/to/release>]] [**--install-dir**=<path>] [**--prefix**=<path>] [**--local-dir**=<path>] [**--make-personal-condor**] [**--type** = < submit, execute, manager >] [**--central-manager** = < hostname>] [**--owner** = < ownername >] [**--make-personal-stork**] [**--overwrite**] [**--ignore-missing-libs**] [**--force**] [**--no-env-scripts**] [**--env-scripts-dir** = < directory >] [**--backup**] [**--stork**] [**--credd**] [**--verbose**]

Description

condor_configure and *condor_install* refer to a single script that installs and/or configures Condor on Unix machines. As the names imply, *condor_install* is intended to perform a Condor installation, and *condor_configure* is intended to configure (or reconfigure) an existing installation. Both will run with Perl 5.6.0 or more recent versions.

condor_configure (and *condor_install*) are designed to be run more than one time where required. It can install Condor when invoked with a correct configuration via

```
condor_install
```

or

```
condor_configure --install
```

or, it can change the configuration files when invoked via

```
condor_configure
```

Note that changes in the configuration files do not result in changes while Condor is running. To effect changes while Condor is running, it is necessary to further use the *condor_reconfig* or *condor_restart* command. *condor_reconfig* is required where the currently executing daemons need to be informed of configuration changes. *condor_restart* is required where the options **--make-personal-condor** or **--type** are used, since these affect which daemons are running.

Running *condor_configure* or *condor_install* with no options results in a usage screen being printed. The **--help** option can be used to display a full help screen.

Within the options given below, the phrase *release directories* is the list of directories that are released with Condor. This list includes: `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src`.

Options

- help** Print help screen and exit

- install** Perform installation, assuming that the current working directory contains the release directories. Without further options, the configuration is that of a Personal Condor, a complete one-machine pool. If used as an upgrade within an existing installation directory, existing configuration files and local directory are preserved. This is the default behavior of *condor_install*.

- install-dir=<path>** Specifies the path where Condor should be installed or the path where it already is installed. The default is the current working directory.

- prefix=<path>** This is an alias for **—install-dir**.

- local-dir=<path>** Specifies the location of the local directory, which is the directory that generally contains the local (machine-specific) configuration file as well as the directories where Condor daemons write their run-time information (`spool`, `log`, `execute`). This location is indicated by the `LOCAL_DIR` variable in the configuration file. When installing (that is, if **—install** is specified), *condor_configure* will properly create the local directory in the location specified. If none is specified, the default value is given by the evaluation of `$(RELEASE_DIR)/local.$(HOSTNAME)`.

During subsequent invocations of *condor_configure* (that is, without the **—install** option), if the **—local-dir** option is specified, the new directory will be created and the `log`, `spool` and `execute` directories will be moved there from their current location.

- make-personal-condor** Installs and configures for Personal Condor, a fully-functional, one-machine pool.

- type= < submit, execute, manager >** One or more of the types may be listed. This determines the roles that a machine may play in a pool. In general, any machine can be a submit and/or execute machine, and there is one central manager per pool. In the case of a Personal Condor, the machine fulfills all three of these roles.

- central-manager=<hostname>** Instructs the current Condor installation to use the specified machine as the central manager. This modifies the configuration variables

COLLECTOR_HOST and NEGOTIATOR_HOST to point to the given host name). The central manager machine's Condor configuration needs to be independently configured to act as a manager using the option **-type=manager**.

—owner=<ownername> Set configuration such that Condor daemons will be executed as the given owner. This modifies the ownership on the log, spool and execute directories and sets the CONDOR_IDS value in the configuration file, to ensure that Condor daemons start up as the specified effective user. See section 3.6.11 on UIDs in Condor on page 315 for details. This is only applicable when *condor_configure* is run by root. If not run as root, the owner is the user running the *condor_configure* command.

—overwrite Always overwrite the contents of the sbin directory in the installation directory. By default, *condor_install* will not install if it finds an existing sbin directory with Condor programs in it. In this case, *condor_install* will exit with an error message. Specify **—overwrite** or **—backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an sbin directory out of the way that it should not move. This is particularly useful when trying to install Condor in a location used by other things (/usr, /usr/local, etc.) For example: *condor_install* **—prefix=/usr** will not move /usr/sbin out of the way unless you specify the **—backup** option.

The **—backup** behavior is used to prevent *condor_install* from overwriting running daemons – Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

—backup Always backup the sbin directory in the installation directory. By default, *condor_install* will not install if it finds an existing sbin directory with Condor programs in it. In this case, *condor_install* will exit with an error message. You must specify **—overwrite** or **—backup** to tell *condor_install* what to do.

This prevents *condor_install* from moving an sbin directory out of the way that it should not move. This is particularly useful if you're trying to install Condor in a location used by other things (/usr, /usr/local, etc.) For example: *condor_install* **—prefix=/usr** will not move /usr/sbin out of the way unless you specify the **—backup** option.

The **—backup** behavior is used to prevent *condor_install* from overwriting running daemons – Unix semantics will keep the existing binaries running, even if they have been moved to a new directory.

—ignore-missing-libs Ignore missing shared libraries that are detected by *condor_install*. By default, *condor_install* will detect missing shared libraries such as libstdc++.so.5 on Linux; it will print messages and exit if missing libraries are detected. The **—ignore-missing-libs** will cause *condor_install* to not exit, and to proceed with the installation if missing libraries are detected.

- force** This is equivalent to enabling both the —**overwrite** and —**ignore-missing-libs** command line options.
- no-env-scripts** By default, *condor_configure* writes simple *sh* and *csh* shell scripts which can be sourced by their respective shells to set the user's `PATH` and `CONDOR_CONFIG` environment variables. This option prevents *condor_configure* from generating these scripts.
- env-scripts-dir=<directory>** By default, the simple *sh* and *csh* shell scripts (see —**no-env-scripts** for details) are created in the root directory of the Condor installation. This option causes *condor_configure* to generate these scripts in the specified directory.
- make-personal-stork** Creates a Personal Stork, using the *condor_credd* daemon.
- stork** Configures the Stork data placement server. Use this option with the —**credd** option.
- credd** Configure the the *condor_credd* daemon (credential manager daemon).
- verbose** Print information about changes to configuration variables as they occur.

Exit Status

condor_configure will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

Examples

Install Condor on the machine (`machine1@cs.wisc.edu`) to be the pool's central manager. On `machine1`, within the directory that contains the unzipped Condor distribution directories:

```
% condor_install --type=submit,execute,manager
```

This will allow the machine to submit and execute Condor jobs, in addition to being the central manager of the pool.

To change the configuration such that `machine2@cs.wisc.edu` is an execute-only machine (that is, a dedicated computing node) within a pool with central manager on `machine1@cs.wisc.edu`, issue the command on that `machine2@cs.wisc.edu` from within the directory where Condor is installed:

```
% condor_configure --central-manager=machine1@cs.wisc.edu --type=execute
```

To change the location of the LOCAL_DIR directory in the configuration file, do (from the directory where Condor is installed):

```
% condor_configure --local-dir=/path/to/new/local/directory
```

This will move the log,spool,execute directories to /path/to/new/local/directory from the current local directory.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_convert_history

Convert the history file to the new format

Synopsis

***condor_convert_history* [-help]**

***condor_convert_history* history-file1 [history-file2...]**

Description

As of Condor version 6.7.19, the Condor history file has a new format to allow fast searches backwards through the file. Not all queries can take advantage of the speed increase, but the ones that can are significantly faster.

Entries placed in the history file after upgrade to Condor 6.7.19 will automatically be saved in the new format. The new format adds information to the string which distinguishes and separates job entries. In order to search within this new format, no changes are necessary. However, to be able to search the entire history, the history file must be converted to the updated format. *condor_convert_history* does this.

Turn the *condor_schedd* daemon off while converting history files. Turn it back on after conversion is completed.

Arguments to *condor_convert_history* are the history files to convert. The history file is normally in the Condor spool directory; it is named *history*. Since the history file is rotated, there may be multiple history files, and all of them should be converted. On Unix platform variants, the easiest way to do this is:

```
cd `condor_config_val SPOOL`  
condor_convert_history history*
```

condor_convert_history makes a back up of each original history files in case of a problem. The names of these back up files are listed; names are formed by appending the suffix *.oldver* to the original file name. Move these back up files to a directory other than the spool directory. If kept in the spool directory, *condor_history* will find the back ups, and will appear to have duplicate jobs.

Exit Status

condor_convert_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_dagman

meta scheduler of the jobs submitted as the nodes of a DAG or DAGs

Synopsis

```
condor_dagman      [-debug level]      [-rescue filename]      [-maxidle numberOfJobs]
[-maxjobs numberOfJobs] [-maxpre NumberOfPREscripts] [-maxpost NumberOfPOSTscripts]
[-noeventchecks]   [-allowlogerror]   [-usedagdir]   -lockfile filename   [-waitfordebug]
[-autorescue 0—1]   [-dorescuefrom number]   -csdversion version_string
[-allowversionmismatch] [-DumpRescue] -dag dag_file [-dag dag_file_2 ... -dag dag_file_n ]
```

Description

condor_dagman is a meta scheduler for the Condor jobs within a DAG (directed acyclic graph) (or multiple DAGs). In typical usage, a submitter of jobs that are organized into a DAG submits the DAG using *condor_submit_dag*. *condor_submit_dag* does error checking on aspects of the DAG and then submits *condor_dagman* as a Condor job. *condor_dagman* uses log files to coordinate the further submission of the jobs within the DAG.

As part of *daemoncore*, the set of command-line arguments given in section 3.9.2 work for *condor_dagman*.

Arguments to *condor_dagman* are either automatically set by *condor_submit_dag* or they are specified as command-line arguments to *condor_submit_dag* and passed on to *condor_dagman*. The method by which the arguments are set is given in their description below.

condor_dagman can run multiple, independent DAGs. This is done by specifying multiple **-dag** arguments. Pass multiple DAG input files as command-line arguments to *condor_submit_dag*.

Debugging output may be obtained by using the **-debug *level*** option. Level values and what they produce is described as

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; normal output, errors and warnings
- level = 3; output errors, as well as all warnings
- level = 4; internal debugging output
- level = 5; internal debugging output; outer loop debugging
- level = 6; internal debugging output; inner loop debugging

- level = 7; internal debugging output; rarely used

Options

- debug *level*** An integer level of debugging output. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. This command-line option to *condor_submit_dag* is passed to *condor_dagman* or defaults to the value 3, as set by *condor_submit_dag*.
- rescue *filename*** Sets the file name of the rescue DAG to write in the case of a failure. As passed by *condor_submit_dag*, the name of the file will be the name of the DAG input file concatenated with the string *.rescue*. This argument is now optional, and in general it is preferred to not specify it. This allows *condor_dagman* to automatically generate an appropriate rescue DAG name.
- maxidle *NumberOfJobs*** Sets the maximum number of idle jobs allowed before *condor_dagman* stops submitting more jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. *NumberOfJobs* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the number of idle jobs is unlimited.
- maxjobs *numberOfJobs*** Sets the maximum number of jobs within the DAG that will be submitted to Condor at one time. *numberOfJobs* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of jobs is unlimited.
- maxpre *NumberOfPREscripts*** Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of PRE scripts is unlimited.
- maxpost *NumberOfPOSTscripts*** Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of POST scripts is unlimited.
- noeventchecks** This argument is no longer used; it is now ignored. Its functionality is now implemented by the DAGMAN_ALLOW_EVENTS configuration macro (see section 3.3.25).
- allowlogerror** This optional argument has *condor_dagman* try to run the specified DAG, even in the case of detected errors in the user log specification.

- usedagdir** This optional argument causes *condor_dagman* to run each specified DAG as if the directory containing that DAG file was the current working directory. This option is most useful when running multiple DAGs in a single *condor_dagman*.
- lockfile *filename*** Names the file created and used as a lock file. The lock file prevents execution of two of the same DAG, as defined by a DAG input file. A default lock file ending with the suffix `.dag.lock` is passed to *condor_dagman* by *condor_submit_dag*.
- waitfordebug** This optional argument causes *condor_dagman* to wait at startup until someone attaches to the process with a debugger and sets the `wait_for_debug` variable in `main_init()` to false.
- autorescue *0—1*** Whether to automatically run the newest rescue DAG for the given DAG file, if one exists (0 = false, 1 = true).
- dorescuefrom *number*** Forces *condor_dagman* to run the specified rescue DAG number for the given DAG. A value of 0 is the same as not specifying this option. Specifying a non-existent rescue DAG is a fatal error.
- csdversion *version_string*** *version_string* is the version of the *condor_submit_dag* program. At startup, *condor_dagman* checks for a version mismatch with the *condor_submit_dag* version in this argument.
- allowversionmismatch** This optional argument causes *condor_dagman* to allow a version mismatch between *condor_dagman* itself and the `.condor.sub` file produced by *condor_submit_dag* (or, in other words, between *condor_submit_dag* and *condor_dagman*). WARNING! This option should be used only if absolutely necessary. Allowing version mismatches can cause subtle problems when running DAGs.
- DumpRescue** This optional argument causes *condor_dagman* to immediately dump a rescue DAG and then exit, as opposed to actually running the DAG. (This feature is mainly intended for testing.)
- dag *filename*** *filename* is the name of the DAG input file that is set as an argument to *condor_submit_dag*, and passed to *condor_dagman*.

Exit Status

condor_dagman will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

condor_dagman is normally not run directly, but submitted as a Condor job by running *condor_submit_dag*. See the *condor_submit_dag* manual page 825 for examples.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_fetchlog

Retrieve a daemon's log file that is located on another computer

Synopsis

condor_fetchlog [-help | -version]

condor_fetchlog [-pool *centralmanagerhostname[:portnumber]*] [-master | -startd | -schedd | -collector | -negotiator | -kbdd] *machine-name subsystem[.extension]*

Description

condor_fetchlog contacts Condor running on the machine specified by *machine-name*, and asks it to return a log file from that machine. Which log file is determined from the *subsystem[.extension]* argument. The log file is printed to standard output. This command eliminates the need to remotely log in to a machine in order to retrieve a daemon's log file.

For security purposes of authentication and authorization, this command requires an administrator's level of access. See section 3.6.1 on page 282 for more details about Condor's security mechanisms.

The *subsystem[.extension]* argument is utilized to construct the log file's name. Without an optional *.extension*, the value of the configuration variable named *subsystem.LOG* defines the log file's name. When specified, the *.extension* is appended to this value.

Acceptable strings for the argument *subsystem* are as given as possible values of the predefined configuration variable \$(SUBSYSTEM). See the definition in section 3.3.1.

A value for the optional *.extension* argument may be one of the three strings:

1. *.old*
2. *.slot<X>*
3. *.slot<X>.old*

Within these strings, <X> is substituted with the slot number.

Options

-help Display usage information

- version** Display version information
- pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number
- master** Send the command to the *condor_master* daemon (default)
- startd** Send the command to the *condor_startd* daemon
- schedd** Send the command to the *condor_schedd* daemon
- collector** Send the command to the *condor_collector* daemon
- kbdd** Send the command to the *condor_kbdd* daemon

Examples

To get the *condor_negotiator* daemon's log from a host named `head.example.com` from within the current pool:

```
condor_fetchlog head.example.com NEGOTIATOR
```

To get the *condor_startd* daemon's log from a host named `execute.example.com` from within the current pool:

```
condor_fetchlog execute.example.com STARTD
```

This command requested the *condor_startd* daemon's log from the *condor_master*. If the *condor_master* has crashed or is unresponsive, ask another daemon running on that computer to return the log. For example, ask the *condor_startd* daemon to return the *condor_master*'s log:

```
condor_fetchlog -startd execute.example.com MASTER
```

Exit Status

condor_fetchlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_findhost

find machine(s) in the pool that can be used with minimal impact on currently running Condor jobs and best meet any specified constraints

Synopsis

condor_findhost [-help] [-m] [-n *num*] [-c *c_expr*] [-r *r_expr*] [-p *centralmanagerhostname*]

Description

condor_findhost searches a Condor pool of machines for the best machine or machines that will have the minimum impact on running Condor jobs if the machine or machines are taken out of the pool. The search may be limited to the machine or machines that match a set of constraints and rank expression.

condor_findhost returns a fully-qualified domain name for each machine. The search is limited (constrained) to a specific set of machines using the *-c* option. The search can use the *-r* option for rank, the criterion used for selecting a machine or machines from the constrained list.

Options

-help Display usage information and exit

-m Only search for entire machines. Slots within an entire machine are not considered.

-n *num* Find and list up to *num* machines that fulfill the specification. *num* is an integer greater than zero.

-c *c_expr* Constrain the search to only consider machines that result from the evaluation of *c_expr*. *c_expr* is a ClassAd expression.

-r *r_expr* *r_expr* is the rank expression evaluated to use as a basis for machine selection. *r_expr* is a ClassAd expression.

-p *centralmanagerhostname* Specify the pool to be searched by giving the central manager's host name. Without this option, the current pool is searched.

General Remarks

condor_findhost is used to locate a machine within a pool that can be taken out of the pool with the least disturbance of the pool.

An administrator should set preemption requirements for the Condor pool. The expression

```
(Interactive == TRUE )
```

will let *condor_findhost* know that it can claim a machine even if Condor would not normally preempt a job running on that machine.

Exit Status

The exit status of *condor_findhost* is zero on success. If not able to identify as many machines as requested, it returns one more than the number of machines identified. For example, if 8 machines are requested, and *condor_findhost* only locates 6, the exit status will be 7. If not able to locate any machines, or an error is encountered, *condor_findhost* will return the value 1.

Examples

To find and list four machines, preferring those with the highest mips (on Drystone benchmark) rating:

```
condor_findhost -n 4 -r "mips"
```

To find and list 24 machines, considering only those where the kflops attribute is not defined:

```
condor_findhost -n 24 -c "kflops==undefined"
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_glidein

add a remote grid resource to a local Condor pool

Synopsis

condor_glidein [-help]

condor_glidein [-admin *address*] [-anybody] [-archdir *dir*] [-basedir *basedir*]
 [-count *CPU count*] [;Execute Task Options;] [;Generate File Options;]
 [-gsi_daemon_name *cert_name*] [-idletime *minutes*] [-install_gsi_trusted_ca_dir *path*]
 [-install_gsi_gridmap *file*] [-localdir *dir*] [-memory *MBytes*] [-project *name*] [-queue *name*]
 [-runtime *minutes*] [-runonly] [;Set Up Task Options;] [-suffix *suffix*] [-slots *slot count*]
;contact argument;

Description

condor_glidein allows the temporary addition of a grid resource to a local Condor pool. The addition is accomplished by installing and executing some of the Condor daemons on the remote grid resource, such that it reports in as part of the local Condor pool. *condor_glidein* accomplishes two separate tasks: set up and execution. These separated tasks allow flexibility, in that the user may use *condor_glidein* to do only one of the tasks or both, in addition to customizing the tasks.

The set up task generates a script that may be used to start the Condor daemons during the execution task, places this script on the remote grid resource, composes and installs a configuration file, and it installs the *condor_master*, *condor_startd* and *condor_starter* daemons on the grid resource.

The execution task runs the script generated by the set up task. The goal of the script is to invoke the *condor_master* daemon. The Condor job *glidein_startup* appears in the queue of the local Condor pool for each invocation of *condor_glidein*. To remove the grid resource from the local Condor pool, use *condor_rm* to remove the *glidein_startup* job.

The Condor jobs to do both the set up and execute tasks utilize Condor-G and Globus protocols (gt2 or gt4) to communicate with the remote resource. Therefore, an X.509 certificate (proxy) is required for the user running *condor_glidein*.

Specify the remote grid machine with the command line argument *;contact argument;*. *;contact argument;* takes one of 4 forms:

1. *hostname*
2. *Globus contact string*
3. *hostname/jobmanager-;schedulername;*

4. **-contactfile** *filename*

The argument **-contactfile** *filename* specifies the full path and file name of a file that contains Globus contact strings. Each of the resources given by a Globus contact string is added to the local Condor pool.

The set up task of *condor_glidein* copies the binaries for the correct platform from a central server. To obtain access to the server, or to set up your own server, follow instructions on the Glidein Server Setup page, at <http://www.cs.wisc.edu/condor/glidein>. Set up need only be done once per site, as the installation is never removed.

By default, all files installed on the remote grid resource are placed in the directory `$(HOME)/Condor_glidein`. `$(HOME)` is evaluated and defined on the remote machine using a grid map. This directory must be in a shared file system accessible by all machines that will run the Condor daemons. By default, the daemon's log files will also be written in this directory. Change this directory with the **-localdir** option to make Condor daemons write to local scratch space on the execution machine. For debugging initial problems, it may be convenient to have the log files in the more accessible default directory. If using the default directory, occasionally clean up old log and execute directories to avoid running out of space.

Examples

To have 10 grid resources running PBS at a grid site with a gatekeeper named `gatekeeper.site.edu` join the local Condor pool:

```
% condor_glidein -count 10 gatekeeper.site.edu/jobmanager-pbs
```

If you try something like the above and *condor_glidein* is not able to automatically determine everything it needs to know about the remote site, it will ask you to provide more information. A typical result of this process is something like the following command:

```
% condor_glidein \  
  -count 10 \  
  -arch 6.6.7-i686-pc-Linux-2.4 \  
  -setup_jobmanager jobmanager-fork \  
  gatekeeper.site.edu/jobmanager-pbs
```

The Condor jobs that do the set up and execute tasks will appear in the queue for the local Condor pool. As a result of a successful glidein, use *condor_status* to see that the remote grid resources are part of the local Condor pool.

A list of common problems and solutions is presented in this manual page.

Generate File Options

- genconfig** Create a local copy of the configuration file that may be used on the remote resource. The file is named `glidein_condor_config.<suffix>`. The string defined by `<suffix>` defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option. The configuration file may be edited for later use with the **-useconfig** option.

- genstartup** Create a local copy of the script used on the remote resource to invoke the *condor_master*. The file is named `glidein_startup.<suffix>`. The string defined by `<suffix>` defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option. The file may be edited for later use with the **-usestartup** option.

- gensubmit** Generate submit description files, but do not submit. The submit description file for the set up task is named `glidein_setup.submit.<suffix>`. The submit description file for the execute task is named `glidein_run.submit.<suffix>`. The string defined by `<suffix>` defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option.

Set Up Task Options

- setuponly** Do only the set up task of *condor_glidein*. This option cannot be run simultaneously with **-runonly**.

- setup_here** Do the set up task on the local machine, instead of at a remote grid resource. This may be used, for example, to do the set up task of *condor_glidein* in an AFS area that is read-only from the remote grid resource.

- forcesetup** During the set up task, force the copying of files, even if this overwrites existing files. Use this to push out changes to the configuration.

- useconfig *config_file*** The set up task copies the specified configuration file, rather than generating one.

- usestartup *startup_file*** The set up task copies the specified startup script, rather than generating one.

- setup_jobmanager *jobmanagername*** Identifies the jobmanager on the remote grid resource to receive the files during the set up task. If a reasonable default can be discovered through

MDS, this is optional. *jobmanagername* is a string representing any gt2 name for the job manager. The correct string in most cases will be *jobmanager-fork*. Other common strings may be *jobmanager*, *jobmanager-condor*, *jobmanager-pbs*, and *jobmanager-lsf*.

Execute Task Options

-runonly Starts execution of the Condor daemons on the grid resource. If any of the necessary files or executables are missing, *condor_glidein* exits with an error code. This option cannot be run simultaneously with **-setupononly**.

-run_here Runs *condor_master* directly rather than submitting a Condor job that causes the remote execution. To instead generate a script that does this, use **-run_here** in combination with **-gensubmit**. This may be useful for running Condor daemons on resources that are not directly accessible by Condor.

Options

-help Display brief usage information and exit.

-basedir *basedir* Specifies the base directory on the remote grid resource used for placing files. The default directory is $\$(HOME)/Condor_glidein$ on the grid resource.

-archdir *dir* Specifies the directory on the remote grid resource for placement of the Condor executables. The default value for **-archdir** is based upon version information on the grid resource. It is of the form `<basedir>/<condor-version>-<Globus canonicalsystemname>`. An example of the directory (without the base directory) for Condor version 6.1.13 running on a Sun Sparc machine with Solaris 2.6 is `6.1.13-sparc-sun-solaris-2.6`.

-localdir *dir* Specifies the directory on the remote grid resource in which to create log and execution subdirectories needed by Condor. If limited disk quota in the home or base directory on the grid resource is a problem, set **-localdir** to a large temporary space, such as `/tmp` or `/scratch`. If the batch system requires invocation of Condor daemons in a temporary scratch directory, `'.'` may be used for the definition of the **-localdir** option.

-arch *architecture* Identifies the platform of the required tarball containing the correct Condor daemon executables to download and install. If a reasonable default can be discovered through MDS, this is optional. A list of possible values may be found at <http://www.cs.wisc.edu/condor/glidein/binaries>. The architecture name is the same as the

tarball name without the suffix `tar.gz`. An example is `6.6.5-i686-pc-Linux-2.4`.

-queue *name* The argument *name* is a string used at the grid resource to identify a job queue.

-project *name* The argument *name* is a string used at the grid resource to identify a project name.

-memory *MBytes* The maximum memory size in Megabytes to request from the grid resource.

-count *CPU count* The number of CPUs requested to join the local pool. The default is 1.

-slots *slot count* For machines with multiple CPUs, the CPUs maybe divided up into slots. *slot count* is the number of slots that results. By default, Condor divides multiple-CPU resources such that each CPU is a slot, each with an equal share of RAM, disk, and swap space. This option configures the number of slots, so that multi-threaded jobs can run in a slot with multiple CPUs. For example, if 4 CPUs are requested and **-slots** is not specified, Condor will divide the request up into 4 slots with 1 CPU each. However, if **-slots 2** is specified, Condor will divide the request up into 2 slots with 2 CPUs each, and if **-slots 1** is specified, Condor will put all 4 CPUs into one slot.

-idletime *minutes* The amount of time that a remote grid resource will remain idle state, before the daemons shut down. A value of 0 (zero) means that the daemons never shut down due to remaining in the idle state. In this case, the **-runtime** option defines when the daemons shut down. The default value is 20 minutes.

-runtime *minutes* The maximum amount of time the Condor daemons on the remote grid resource will run before shutting themselves down. This option is useful for resources with enforced maximum run times. Setting **-runtime** to be a few minutes shorter than the enforced limit gives the daemons time to perform a graceful shut down.

-anybody Sets the Condor `START` expression for the added remote grid resource to `True`. This permits any user's job which can run on the added remote grid resource to run. Without this option, only jobs owned by the user executing *condor_glidein* can execute on the remote grid resource. WARNING: Using this option may violate the usage policies of many institutions.

-admin *address* Where to send e-mail with problems. The default is the login of the user running *condor_glidein* at UID domain of the local Condor pool.

-suffix *X* Suffix to use when generating files. Default is process id.

-gsi_daemon_name cert_name Includes and enables GSI authentication in the configuration for the remote grid resource. The argument is the GSI certificate name that the daemons will use to authenticate themselves.

-install_gsi_trusted_ca_dir path The argument identifies the directory containing the trusted CA certificates that the daemons are to use (for example, `/etc/grid-security/certificates`). The contents of this directory will be installed at the remote site in the directory `<basedir>/grid-security`.

-install_gsi_gridmap file The argument is the file name of the GSI-specific X.509 map file that the daemons will use. The file will be installed at the remote site in `<basedir>/grid-security`. The file contains entries mapping certificates to user names. At the very least, it must contain an entry for the certificate given by the command-line option **-gsi_daemon_name**. If other Condor daemons use different certificates, then this file will also list any certificates that the daemons will encounter for the *condor_schedd*, *condor_collector*, and *condor_negotiator*. See section 3.6.3 for more information.

Exit Status

condor_glidein will exit with a status value of 0 (zero) upon complete success, or with non-zero values upon failure. The status value will be 1 (one) if *condor_glidein* encountered an error making a directory, was unable to copy a tar file, encountered an error in parsing the command line, or was not able to gather required information. The status value will be 2 (two) if there was an error in the remote set up. The status value will be 3 (three) if there was an error in remote submission. The status value will be -1 (negative one) if no resource was specified in the command line.

Common problems are listed below. Many of these are best discovered by looking in the `StartLog` log file on the remote grid resource.

WARNING: The file xxx is not writable by condor This error occurs when *condor_glidein* is run in a directory that does not have the proper permissions for Condor to access files. An AFS directory does not give Condor the user's AFS ACLs.

Glideins fail to run due to GLIBC errors Check the list of available glidein binaries (<http://www.cs.wisc.edu/condor/glidein/binaries>), and try specifying the architecture name that includes the correct glibc version for the remote grid site.

Glideins join pool but no jobs run on them One common cause of this problem is that the remote grid resources are in a different file system domain, and the submitted Condor jobs have an implicit requirement that they must run in the same file system domain. See section 2.5.4 for details on using Condor's file transfer capabilities to solve this problem. Another cause of this problem is a communication failure. For example, a firewall may be preventing the

condor_negotiator or the *condor_schedd* daemons from connecting to the *condor_startd* on the remote grid resource. Although work is being done to remove this requirement in the future, it is currently necessary to have full bidirectional connectivity, at least over a restricted range of ports. See page 166 for more information on configuring a port range.

Glideins run but fail to join the pool This may be caused by the local pool's security settings or by a communication failure. Check that the security settings in the local pool's configuration file allow write access to the remote grid resource. To not modify the security settings for the pool, run a separate pool specifically for the remote grid resources, and use flocking to balance jobs across the two pools of resources. If the log files indicate a communication failure, then see the next item.

The startd cannot connect to the collector This may be caused by several things. One is a fire-wall. Another is when the compute nodes do not have even outgoing network access. Configuration to work without full network access to and from the compute nodes is still in the experimental stages, so for now, the short answer is that you must at least have a range of open (bidirectional) ports and set up the configuration file as described on page 166. Use the option **-genconfig**, edit the generated configuration file, and then do the glidein execute task with the option **-useconfig**.)

Another possible cause of connectivity problems may be the use of UDP by the *condor_startd* to register itself with the *condor_collector*. Force it to use TCP as described on page 167.

Yet another possible cause of connectivity problems is when the remote grid resources have more than one network interface, and the default one chosen by Condor is not the correct one. One way to fix this is to modify the glidein startup script using the **-genstartup** and **-usestartup** options. The script needs to determine the IP address associated with the correct network interface, and assign this to the environment variable `_condor_NETWORK_INTERFACE`.

NFS file locking problems If the **-localdir** option uses files on NFS (not recommended, but sometimes convenient for testing), the Condor daemons may have trouble manipulating file locks. Try inserting the following into the configuration file:

```
IGNORE_NFS_LOCK_ERRORS = True
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_history

View log of Condor jobs completed to date

Synopsis

condor_history [-help]

condor_history [-backwards] [-completedsince *postgreimestamp*] [-constraint *expr*]
 [-f *filename*] [-format *formatString AttributeName*] [-l | -long | -xml] [-match *number*]
 [-name *schedd-name*] [cluster | cluster.process | owner]

Description

condor_history displays a summary of all Condor jobs listed in the specified history files, or in the Quill database, when Quill is enabled. If no history files are specified (with the -f option) and Quill is not enabled, the local history file as specified in Condor's configuration file (\$ (SPOOL) /history by default) is read. The default listing summarizes (in chronological order) each job on a single line, and contains the following items:

ID The cluster/process id of the job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

RUN_TIME Remote wall clock time accumulated by the job to date in days, hours, minutes, and seconds. See the definition of RemoteWallClockTime on page 884.

ST Completion status of the job (C = completed and X = removed).

COMPLETED The time the job was completed.

CMD The name of the executable.

If a job ID (in the form of *cluster_id* or *cluster_id.proc_id*) or an *owner* is provided, output will be restricted to jobs with the specified IDs and/or submitted by the specified owner. The -constraint option can be used to display jobs that satisfy a specified boolean expression.

The history file is kept in chronological order, implying that new entries are appended at the end of the file. As of Condor version 6.7.19, the format of the history file is altered to enable faster reading of the history file backwards (most recent job first). History files written with earlier versions of Condor, as well as those that have entries of both the older and newer format need to be converted to the new format. See the *condor_convert_history* manual page on page 709 for details on converting history files to the new format.

Options

- help** Display usage information and exit.

- backwards** List jobs in reverse chronological order. The job most recently added to the history file is first.

- completedsince *postgrestamp*** When Quill is enabled, display only job ads that were in the Completed job state on or after the date and time given by the *postgrestamp*. The *postgrestamp* follows the syntax as given for *PostgreSQL* version 8.0. The behavior of this option is undefined when Quill is *not* enabled.

- constraint *expr*** Display jobs that satisfy the expression.

- f *filename*** Use the specified file instead of the default history file. When Quill is enabled, this option will force the query to read from the history file, and not the database.

- format *formatStringAttributeName*** Display jobs with a custom format. See the *condor_q* man page **-format** option for details.

- l or -long** Display job ads in long format.

- match *number*** Limit the number of jobs displayed to *number*.

- name *schedd-name*** When Quill is enabled, query job ClassAds from the named *condor_schedd* daemon, not the default *condor_schedd* daemon.

- xml** Display job ClassAds in xml format. The xml format is fully defined at <http://www.cs.wisc.edu/condor/classad/refman/>.

Examples

To see all historical jobs since April 1, 2005 at 1pm,

```
%condor_history -completedsince '04/01/2005 13:00'
```

Exit Status

condor_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_hold

put jobs in the queue into the hold state

Synopsis

condor_hold [-help | -version]

condor_hold [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
[-addr "<a.b.c.d:port>"] [*cluster...* | *cluster.process...* | *user...* | -constraint *expression* ...

condor_hold [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
[-addr "<a.b.c.d:port>"] -all

Description

condor_hold places jobs from the Condor job queue in the hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be held are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can place the job on hold.

A job in the hold state remains in the job queue, but the job will not run until released with *condor_release*.

A currently running job that is placed in the hold state by *condor_hold* is sent a hard kill signal. For a standard universe job, this means that the job is removed from the machine without allowing a checkpoint to be produced first.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr "<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

cluster Hold all jobs in the specified cluster

cluster.process Hold the specific job in the cluster

user Hold all jobs belonging to specified user

-constraint expression Hold all jobs which match the job ClassAd expression constraint (within quotation marks). Note that quotation marks must be escaped with the backslash characters for most shells.

-all Hold all the jobs in the queue

See Also

condor_release (on page 764)

Examples

To place on hold all jobs (of the user that issued the *condor_hold* command) that are not currently running:

```
% condor_hold -constraint "JobStatus!=2"
```

Multiple options within the same command cause the union of all jobs that meet either (or both) of the options to be placed in the hold state. Therefore, the command

```
% condor_hold Mary -constraint "JobStatus!=2"
```

places all of Mary's queued jobs into the hold state, and the constraint holds all queued jobs not currently running. It also sends a hard kill signal to any of Mary's jobs that are currently running. Note that the jobs specified by the constraint will also be Mary's jobs, if it is Mary that issues this example *condor_hold* command.

Exit Status

condor_hold will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_load_history

Read a Condor history file into a Quill database

Synopsis

condor_load_history **-f** *historyfilename* [**-name** *schedd-name jobqueue-birthdate*]

Description

condor_load_history reads a Condor history file, adding its information to a Quill database. The Quill database is located via configuration variables. The history file to read is defined by the required **-f** *historyfilename* argument.

The combination of a *condor_schedd* daemon's name together with its process creation date (the job queue's birthdate) define a unique identifier that may be attached to the Quill database with the **-name** option. The format of birthdate expected is exactly the first line of the *job_queue.log* file. The location of this file may be determined using

```
% condor_config_val spool
```

Be aware and expect that the reading and processing of a sizable history file may take a large amount of time.

Options

-name *schedd-name jobqueue-birthdate* The *schedd-name* and *jobqueue-birthdate* combine to form a unique name for the database. The expected values are the name of the *condor_schedd* daemon and the first line of the *job_queue.log* file, which gives a job queue creation time.

Exit Status

condor_load_history will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_master

The master Condor Daemon

Synopsis

condor_master

Description

This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the *condor_master* will restart the affected daemons. In addition, if any daemon crashes, the *condor_master* will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

Section 3.1.2 in the Administrator's Manual has more information about the *condor_master* and other Condor daemons. See Section 3.9.2 for documentation on command line arguments for *condor_master*.

The DAEMON_LIST configuration macro is used by the *condor_master* to provide a per-machine list of daemons that should be started and kept running. For daemons that are specified in the DC_DAEMON_LIST configuration macro, the *condor_master* daemon will spawn them automatically appending a *-f* argument. For those listed in DAEMON_LIST, but not in DC_DAEMON_LIST, there will be no *-f* argument.

Options

-n name Provides an alternate name for the *condor_master* to override that given by the MASTER_NAME configuration variable.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_master_off

Shutdown Condor and the *condor_master*

Synopsis

condor_master_off [-help] [-version] [hostname ...]

Description

condor_master_off no longer exists.

General Remarks

condor_master_off no longer exists as a Condor command. Instead, use

```
condor_off -master
```

to accomplish this task.

See Also

See the *condor_off* manual page.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_off

Shutdown Condor daemons

Synopsis

condor_off [-help | -version]

condor_off [-graceful | -fast] [-debug] [-name *name* | *name* | -addr "<a.b.c.d:port>" | "<a.b.c.d:port>" ...] [-all]

[-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

condor_off [-graceful | -fast] [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *name*] [-addr "<a.b.c.d:port>"] ... [| -all]

[-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

Description

condor_off shuts down a set of the Condor daemons running on a set of one or more machines. It does this cleanly so that checkpointable jobs may gracefully exit with minimal loss of work.

The command *condor_off* without any arguments will shut down all daemons except *condor_master*. The *condor_master* can then handle both local and remote requests to restart the other Condor daemons if need be. To restart Condor running on a machine, see the *condor_on* command.

With the **-subsystem *master*** option, *condor_off* will shut down all daemons including the *condor_master*. Specification using the **-subsystem** option will shut down only the specified daemon.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 282 for further explanation.

Options

-help Display usage information

-version Display version information

-graceful Gracefully shutdown daemons (the default)

-fast Quickly shutdown daemons

-peaceful Wait indefinitely for jobs to finish

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "<*a.b.c.d:port*>" Send the command to a machine's master located at "<*a.b.c.d:port*>"

"<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-all Send the command to all machines in the pool

-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill* Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

Exit Status

condor_off will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To shut down all daemons (other than *condor_master*) on the local host:

```
% condor_off
```

To shut down only the *condor_collector* on three named machines:

```
% condor_off cinnamon cloves vanilla -subsystem collector
```

To shut down daemons within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command shuts down all daemons except the *condor_master* on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_off -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_on

Start up Condor daemons

Synopsis

condor_on [-help | -version]

condor_on [-debug] [-name *name* | *name* | -addr "<a.b.c.d:port>" | "<a.b.c.d:port>" ...]
[-all]

[-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

condor_on [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *name*]
[-addr "<a.b.c.d:port>" ...] [-all]

[-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

Description

condor_on starts up a set of the Condor daemons on a set of machines. This command assumes that the *condor_master* is already running on the machine. If this is not the case, *condor_on* will fail complaining that it cannot find the address of the master. The command *condor_on* with no arguments or with the **-subsystem** *master* option will tell the *condor_master* to start up the Condor daemons specified in the configuration variable DAEMON_LIST. If a daemon other than the *condor_master* is specified with the **-subsystem** option, *condor_on* starts up only that daemon.

This command cannot be used to start up the *condor_master* daemon.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 282 for further explanation.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "*<a.b.c.d:port>*" Send the command to a machine's master located at "*<a.b.c.d:port>*"

"<a.b.c.d:port>" Send the command to a machine located at "*<a.b.c.d:port>*"

-all Send the command to all machines in the pool

-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill* Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

Exit Status

condor_on will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To begin running all daemons (other than *condor_master*) given in the configuration variable `DAEMON_LIST` on the local host:

```
% condor_on
```

To start up only the *condor_negotiator* on two named machines:

```
% condor_on robin cardinal -subsystem negotiator
```

To start up only a daemon within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command starts up only the *condor_schedd* daemon on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_on -pool condor.cae.wisc.edu -name cae17 -subsystem schedd
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor.power

send packet intended to wake a machine from a low power state

Synopsis

condor.power [-h]

condor.power [-d] [-m *MACaddress*] [-s *subnet*] [*ClassAdFile*]

Description

condor.power sends one UDP Wake on LAN (WOL) packet to a machine specified either by command line arguments or by the contents of a machine ClassAd. The machine ClassAd may be in a file, where the file name specified by the optional argument *ClassAdFile* is given on the command line. With no command line arguments to specify the machine, and no file specified, *condor.power* quietly presumes that standard input is the file source which will specify the machine ClassAd that includes the public IP address and subnet of the machine.

condor.power needs a complete specification of the machine to be successful. If a MAC address is provided on the command line, but no subnet is given, then the default value for the subnet is used. If a subnet is provided on the command line, but no MAC address is given, then *condor.power* falls back to taking its information in the form of the machine ClassAd as provided in a file or on standard input. Note that this case implies that the command line specification of the subnet is ignored.

Options

-h Print usage information and exit.

-d Enable debugging messages.

-m *MACaddress* Specify the MAC address in the standard format of six groups of two hexadecimal digits separated by colons.

-s *subnet* Specify the subnet in the standard form of an IP address. Without this option, the default subnet used will be 255.255.255.255, causing a broadcast.

Exit Status

condor_power will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_preen

remove extraneous files from Condor directories

Synopsis

condor_preen [-mail] [-remove] [-verbose]

Description

condor_preen examines the directories belonging to Condor, and removes extraneous files and directories which may be left over from Condor processes which terminated abnormally either due to internal errors or a system crash. The directories checked are the LOG, EXECUTE, and SPOOL directories as defined in the Condor configuration files. *condor_preen* is intended to be run as user `root` or user `condor` periodically as a backup method to ensure reasonable file system cleanliness in the face of errors. This is done automatically by default by the *condor_master* daemon. It may also be explicitly invoked on an as needed basis.

When *condor_preen* cleans the SPOOL directory, it always leaves behind the files specified in the configuration variable `VALID_SPOOL_FILES` as given by the configuration. For the LOG directory, the only files removed or reported are those listed within the configuration variable `INVALID_LOG_FILES` list. The reason for this difference is that, in general, the files in the LOG directory ought to be left alone, with few exceptions. An example of exceptions are core files. As there are new log files introduced regularly, it is less effort to specify those that ought to be removed than those that are not to be removed.

Options

-mail Send mail to the user defined in the `PREEN_ADMIN` configuration variable, instead of writing to the standard output.

-remove Remove the offending files and directories rather than reporting on them.

-verbose List all files found in the Condor directories, even those which are not considered extraneous.

Exit Status

condor_preen will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_prio

change priority of jobs in the condor queue

Synopsis

condor_prio [-p *priority*] [+ | - *value*] [-n *schedd_name*] [-pool *pool_name*]
cluster | *cluster.process* | *username* | -a

Description

condor_prio changes the priority of one or more jobs in the condor queue. If a *cluster_id* and a *process_id* are both specified, *condor_prio* attempts to change the priority of the specified process. If a *cluster_id* is specified without a *process_id*, *condor_prio* attempts to change priority for all processes belonging to the specified cluster. If a *username* is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. If the -a flag is set, *condor_prio* attempts to change priority of all jobs in the condor queue. The user must specify a priority adjustment or new priority. If the -p option is specified, the priority of the job(s) are set to the next argument. The user can also adjust the priority by supplying a + or - immediately followed by a digit. The priority of a job can be any integer, with higher numbers corresponding to greater priority. Only the owner of a job or the super user can change the priority for it.

The priority changed by *condor_prio* is only compared to the priority of other jobs owned by the same user and submitted from the same machine. See the "Condor Users and Administrators Manual" for further details on Condor's priority scheme.

Options

-p *priority* Set priority to the specified value

+ | - *value* Change priority by the specified value

-n *schedd_name* Change priority of jobs queued at the specified schedd in the local pool

-pool *pool_name* Change priority of jobs queued at the specified schedd in the specified pool

-a Change priority of all the jobs in the queue

Exit Status

condor_prio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_4

Display information about jobs in queue

Synopsis

condor_4 [-help]

condor_4 [-debug] [-global] [-submitter *submitter*] [-name *name*]
 [-pool *centralmanagerhostname[:portnumber]*] [-analyze] [-better-analyze] [-run] [-hold]
 [-globus] [-goodput] [-io] [-dag] [-long] [-xml] [-format *fmt attr*] [-cputime] [-currentrun]
 [-avgqueuetime] [-jobads *file*] [-machineads *file*] [-direct *rdbsms | quilld | schedd*]
 [{*cluster* | *cluster.process* | *owner* | -constraint *expression* . . . }]

Description

condor_4 displays information about jobs in the Condor job queue. By default, *condor_4* queries the local job queue but this behavior may be modified by specifying:

- the **-global** option, which queries all job queues in the pool
- a schedd name with the **-name** option, which causes the queue of the named schedd to be queried
- a submitter with the **-submitter** option, which causes all queues of the named submitter to be queried

To restrict the display to jobs of interest, a list of zero or more restrictions may be supplied. Each restriction may be one of:

- a *cluster* and a *process* matches jobs which belong to the specified cluster and have the specified process number
- a *cluster* without a *process* matches all jobs belonging to the specified cluster
- a *owner* matches all jobs owned by the specified owner
- a **-constraint** *expression* which matches all jobs that satisfy the specified ClassAd expression. (See section 4.1 for a discussion of ClassAd expressions.)

If no *owner* restrictions are present in the list, the job matches the restriction list if it matches at least one restriction in the list. If *owner* restrictions are present, the job matches the list if it matches one of the *owner* restrictions *and* at least one non-owner restriction.

If the **-long** option is specified, *condor_q* displays a long description of the queried jobs by printing the entire job ClassAd. The attributes of the job ClassAd may be displayed by means of the **-format** option, which displays attributes with a `printf(3)` format. Multiple **-format** options may be specified in the option list to display several attributes of the job. If neither **-long** or **-format** are specified, *condor_q* displays a one line summary of information as follows:

ID The cluster/process id of the condor job.

OWNER The owner of the job.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

RUN-TIME Wall-clock time accumulated by the job to date in days, hours, minutes, and seconds.

ST Current status of the job, which varies somewhat according to the job universe and the timing of updates. U = unexpanded (never been run), H = on hold, R = running, I = idle (waiting for a machine to execute on), C = completed, and X = removed.

PRI User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

SIZE The virtual image size of the executable in megabytes.

CMD The name of the executable.

If the **-dag** option is specified, the OWNER column is replaced with NODENAME for jobs started by Condor DAGMan.

If the **-run** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

HOST(S) The host where the job is running.

If the **-globus** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

STATUS The state that Condor believes the job is in. Possible values are

PENDING The job is waiting for resources to become available in order to run.

ACTIVE The job has received resources, and the application is executing.

FAILED The job terminated before completion because of an error, user-triggered cancel, or system-triggered cancel.

DONE The job completed successfully.

SUSPENDED The job has been suspended. Resources which were allocated for this job may have been released due to a scheduler-specific reason.

UNSUBMITTED The job has not been submitted to the scheduler yet, pending the reception of the GLOBUS_GRAM_PROTOCOL_JOB_SIGNAL_COMMIT_REQUEST signal from a client.

STAGE_IN The job manager is staging in files, in order to run the job.

STAGE_OUT The job manager is staging out files generated by the job.

UNKNOWN

MANAGER A guess at what remote batch system is running the job. It is a guess, because Condor looks at the Globus jobmanager contact string to attempt identification. If the value is fork, the job is running on the remote host without a jobmanager. Values may also be condor, lsf, or pbs.

HOST The host to which the job was submitted.

EXECUTABLE The job as specified as the executable in the submit description file.

If the **-goodput** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

GOODPUT The percentage of RUN_TIME for this job which has been saved in a checkpoint. A low GOODPUT value indicates that the job is failing to checkpoint. If a job has not yet attempted a checkpoint, this column contains [?????].

CPU_UTIL The ratio of CPU_TIME to RUN_TIME for checkpointed work. A low CPU_UTIL indicates that the job is not running efficiently, perhaps because it is I/O bound or because the job requires more memory than available on the remote workstations. If the job has not (yet) checkpointed, this column contains [??????].

Mb/s The network usage of this job, in Megabits per second of run-time.

If the **-io** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

READ The total number of bytes the application has read from files and sockets.

WRITE The total number of bytes the application has written to files and sockets.

SEEK The total number of seek operations the application has performed on files.

XPUT The effective throughput (average bytes read and written per second) from the application's point of view.

BUFSIZE The maximum number of bytes to be buffered per file.

BLOCKSIZE The desired block size for large data transfers.

These fields are updated when a job produces a checkpoint or completes. If a job has not yet produced a checkpoint, this information is not available.

If the **-cputime** option is specified, the RUN_TIME column is replaced with:

CPU_TIME The remote CPU time accumulated by the job to date (which has been stored in a checkpoint) in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown. If the job has not produced a checkpoint, this column contains 0+00:00:00.)

The *-analyze* option may be used to determine why certain jobs are not running by performing an analysis on a per machine basis for each machine in the pool. The reasons may vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_REQUIREMENTS` expression. If the *-long* option is specified along with the *-analyze* option, the reason for failure is displayed on a per machine basis.

The *-better-analyze* option does a more thorough job of determining why jobs are not running than *-analyze*. There are scalability issues present when run on a pool with a large number of machines, as well as when run to analyze a large number of queued jobs. The *-better-analyze* option make take an excessively long time to complete in these cases. Therefore, it is recommended to constrain *-better-analyze* to only analyze one job at a time.

Options

-help Get a brief description of the supported options

-global Get queues of all the submitters in the system

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-submitter *submitter* List jobs of specific submitter from all the queues in the pool

-pool *centralmanagerhostname[:portnumber]* Use the *centralmanagerhostname* as the central manager to locate schedds. (The default is the `COLLECTOR_HOST` specified in the configuration file.

-analyze Perform an approximate analysis to determine how many resources are available to run the requested jobs. These results are only meaningful for jobs using Condor's matchmaker. This option is never meaningful for Scheduler universe jobs and only meaningful for grid universe jobs doing matchmaking.

-better-analyze Perform a more time-consuming, but potentially more extensive analysis to determine how many resources are available to run the requested jobs.

- run** Get information about running jobs.
- hold** Get information about jobs in the hold state. Also displays the time the job was placed into the hold state and the reason why the job was placed in the hold state.
- globus** Get information only about jobs submitted to grid resources described as **gt2** or **gt4**.
- goodput** Display job goodput statistics.
- io** Display job input/output summaries.
- dag** Display DAG jobs under their DAGMan.
- name *name*** Show only the job queue of the named schedd
- long** Display job ads in long format
- xml** Display job ads in xml format. The xml format is fully defined at <http://www.cs.wisc.edu/condor/classad/refman/>.
- format *fmt attr*** Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)` style conversion specifier. Attributes must be from the job ClassAd. Expressions are ClassAd expressions and may refer to attributes in the job ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. The conversion specifier must match the type of the attribute or expression. `%s` is suitable for strings such as `Owner`, `%d` for integers such as `ClusterId`, and `%f` for floating point numbers such as `RemoteWallClockTime`. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, you can include other text that will be reproduced directly. You can specify a format without any conversion specifiers but you must still give attribute. You can include `\n` to specify a line break.
- cputime** Instead of wall-clock allocation time (`RUN_TIME`), display remote CPU time accumulated by the job to date in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown.)

- currentrun** Normally, RUN_TIME contains all the time accumulated during the current run plus all previous runs. If this option is specified, RUN_TIME only displays the time accumulated so far on this current run.
- avgqueuetime** Display the average of time spent in the queue, considering all jobs not completed (those that do not have JobStatus == 4 or JobStatus == 3).
- jobads file** Display jobs from a list of ClassAds from a file, instead of the real ClassAds from the *condor_schedd* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_q -l* is used with the header stripped out.
- machineads file** When doing analysis, use the machine ads from the file instead of the ones from the *condor_collector* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_status -l* is used.
- direct rdbms | quilld | schedd** When the use of Quill is enabled, this option allows a direct query to either the rdbms, the *condor_quill* daemon, or the *condor_schedd* daemon for the requested queue information. It also prevents the queue location discovery algorithm from failing over to alternate sources of information for the queue in case of error. It is useful for debugging an installation of Quill. One of the strings *rdbms*, *quilld*, or *schedd* is required with this option.

Restriction list The restriction list may have zero or more items, each of which may be:

- cluster* match all jobs belonging to cluster
- cluster.proc* match all jobs belonging to cluster with a process number of *proc*
- constraint expression** match all jobs which match the ClassAd expression constraint

A job matches the restriction list if it matches any restriction in the list. Additionally, if *owner* restrictions are supplied, the job matches the list only if it also matches an *owner* restriction.

General Remarks

The default output from *condor_q* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the Condor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_q* is strongly encouraged to use the **-format** option (described above, examples given below).

Although *-analyze* provides a very good first approximation, the analyzer cannot diagnose all possible situations because the analysis is based on instantaneous and local information. Therefore, there

are some situations (such as when several submitters are contending for resources, or if the pool is rapidly changing state) which cannot be accurately diagnosed.

-goodput, *-cputime*, and *-io* are most useful for STANDARD universe jobs, since they rely on values computed when a job checkpoints.

Examples

The **-format** option provides a way to specify both the job attributes and formatting of those attributes. There must be only one conversion specification per **-format** option. As an example, to list only Jane Doe's jobs in the queue, choosing to print and format only the owner of the job, the command line arguments for the job, and the process ID of the job:

```
%condor_q -submitter jdoe -format "%s" Owner -format " %s " Args -format "ProcId = %d\n" ProcId
jdoe 16386 2800 ProcId = 0
jdoe 16386 3000 ProcId = 1
jdoe 16386 3200 ProcId = 2
jdoe 16386 3400 ProcId = 3
jdoe 16386 3600 ProcId = 4
jdoe 16386 4200 ProcId = 7
```

To display only the JobID's of Jane Doe's jobs you can use the following.

```
%condor_q -submitter jdoe -format "%d." ClusterId -format "%d\n" ProcId
27.0
27.1
27.2
27.3
27.4
27.7
```

An example that shows the difference (first set of output) between not using an option to *condor_q* and (second set of output) using the **-globus** option:

ID	OWNER	SUBMITTED	RUN_TIME	ST	PRI	SIZE	CMD
100.0	smith	12/11 13:20	0+00:00:02	R	0	0.0	sleep 10

1 jobs; 0 idle, 1 running, 0 held

ID	OWNER	STATUS	MANAGER	HOST	EXECUTABLE
100.0	smith	ACTIVE	fork	grid.example.com	/bin/sleep

Exit Status

condor_q will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_qedit

modify job attributes

Synopsis

condor_qedit [-debug] [-n schedd-name] [-pool pool-name]
 {cluster | cluster.proc | owner | -constraint constraint} attribute-name attribute-value ...

Description

condor_qedit modifies job ClassAd attributes of queued Condor jobs. The jobs are specified either by cluster number, job ID, owner, or by a ClassAd constraint expression. The *attribute-value* may be any ClassAd expression. String expressions must be surrounded by double quotes.

To ensure security and correctness, *condor_qedit* will not allow modification of the following ClassAd attributes:

- Owner
- ClusterId
- ProcId
- MyType
- TargetType
- JobStatus

Since JobStatus may not be changed with *condor_qedit*, use *condor_hold* to place a job in the hold state, and use *condor_release* to release a held job, instead of attempting to modify JobStatus directly.

If a job is currently running, modified attributes for that job will not affect the job until it restarts. As an example, for PeriodicRemove to affect when a currently running job will be removed from the queue, that job must first be evicted from a machine and returned to the queue. The same is true for other periodic expressions, such as PeriodicHold and PeriodicRelease.

Options

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-n schedd-name Modify job attributes in the queue of the specified schedd

-pool pool-name Modify job attributes in the queue of the schedd specified in the specified pool

Examples

```
% condor_qedit -name north.cs.wisc.edu -pool condor.cs.wisc.edu 249.0 answer 42
Set attribute "answer".
% condor_qedit -name perdita 1849.0 In "myinput"
Set attribute "In".
% condor_qedit jbasney NiceUser TRUE
Set attribute "NiceUser".
% condor_qedit -constraint 'JobUniverse == 1' Requirements '(Arch == "INTEL") && (OpSys == "SOLARIS26") &&
Set attribute "Requirements".
```

General Remarks

A job's ClassAd attributes may be viewed with

```
condor_q -long
```

Exit Status

condor_qedit will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_reconfig

Reconfigure Condor daemons

Synopsis

condor_reconfig [-help | -version]

condor_reconfig [-debug] [-name name | name] [-addr "<a.b.c.d:port>" | "<a.b.c.d:port>" ...] [-all]

[-subsystem master | startd | schedd | collector | negotiator | kbdd | quill] [-full]

condor_reconfig [-debug] [-pool centralmanagerhostname[:portnumber] | -name name] [-addr "<a.b.c.d:port>"] ... [-all]

[-subsystem master | startd | schedd | collector | negotiator | kbdd | quill] [-full]

Description

condor_reconfig reconfigures all of the Condor daemons in accordance with the current status of the Condor configuration file(s). Once reconfiguration is complete, the daemons will behave according to the policies stated in the configuration file(s). The main exception is with the `DAEMON_LIST` variable, which will only be updated if the *condor_restart* command is used. There are a few other configuration settings that can only be changed if the Condor daemons are restarted. Whenever this is the case, it will be mentioned in section 3.3 on page 142 which lists all of the settings used to configure Condor. In general, *condor_reconfig* should be used when making changes to the configuration files, since it is faster and more efficient than restarting the daemons.

The command *condor_reconfig* with no arguments or with the **-subsystem master** option will cause the reconfiguration of the *condor_master* daemon and all the child processes of the *condor_master*.

For security purposes (authentication and authorization), this command requires an administrator's level of access. Note that changes to the `ALLOW_*` and `DENY_*` configuration variables require the **-full** option. See section 3.6.1 on page 282 for further explanation.

Options

-help Display usage information

-version Display version information

-full Perform a full reconfiguration. In addition to re-reading the configuration files, a full reconfiguration will clear cached DNS information in the daemons. Use this option only

when the DNS information needs to be reinitialized.

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "*<a.b.c.d:port>*" Send the command to a machine's master located at "*<a.b.c.d:port>*"

"<a.b.c.d:port>" Send the command to a machine located at "*<a.b.c.d:port>*"

-all Send the command to all machines in the pool

-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill* Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

Exit Status

condor_reconfig will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To reconfigure the *condor_master* and all its children on the local host:

```
% condor_reconfig
```

To reconfigure only the *condor_startd* on a named machine:

```
% condor_reconfig -name bluejay -subsystem startd
```

To reconfigure a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reconfigures the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reconfig -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_reconfig_schedd

Reconfigure condor schedd

Synopsis

condor_reconfig_schedd [-help] [-version] [hostname ...]

Description

condor_reconfig_schedd no longer exists.

General Remarks

condor_reconfig_schedd no longer exists as a Condor command. Instead, use

```
condor_reconfig -schedd
```

to accomplish this task.

See Also

See the *condor_reconfig* manual page.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_release

release held jobs in the Condor queue

Synopsis

condor_release [-help | -version]

condor_release [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
[-addr "<*a.b.c.d:port*>"] [*cluster...* | *cluster.process...* | *user...* | -constraint *expression* ...]

condor_release [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
[-addr "<*a.b.c.d:port*>"] -all

Description

condor_release releases jobs from the Condor job queue that were previously placed in hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be released are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can release the job.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr "<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

cluster Release all jobs in the specified cluster

cluster.process Release the specific job in the cluster

user Release jobs belonging to specified user

-constraint expression Release all jobs which match the job ClassAd expression constraint

-all Release all the jobs in the queue

See Also

condor_hold (on page 730)

Examples

To release all of the jobs of a user named Mary:

```
% condor_release Mary
```

Exit Status

condor_release will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_reschedule

Update scheduling information to the central manager

Synopsis

condor_reschedule [-help | -version]

condor_reschedule [-debug] [-name *name* | *name* | -addr "<*a.b.c.d:port*>" | "<*a.b.c.d:port*>" ...] [-all]

condor_reschedule [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *name*] [-addr "<*a.b.c.d:port*>"] ... [| -all]

Description

condor_reschedule updates the information about a set of machines' resources and jobs to the central manager. This command is used to force an update before viewing the current status of a machine. Viewing the status of a machine is done with the *condor_status* command. *condor_reschedule* also starts a new negotiation cycle between resource owners and resource providers on the central managers, so that jobs can be matched with machines right away. This can be useful in situations where the time between negotiation cycles is somewhat long, and an administrator wants to see if a job in the queue will get matched without waiting for the next negotiation cycle.

A new negotiation cycle cannot occur more frequently than every 20 seconds. Requests for new negotiation cycle within that 20 second window will be deferred until 20 seconds have passed since that last cycle.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "<*a.b.c.d:port*>" Send the command to a machine's master located at "<*a.b.c.d:port*>"

"<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-all Send the command to all machines in the pool

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

Exit Status

condor_reschedule will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To update the information on three named machines:

```
% condor_reschedule robin cardinal bluejay
```

To reschedule on a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reschedules the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reschedule -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_restart

Restart the a set of Condor daemons

Synopsis

condor_restart [-help | -version]

condor_restart [-debug] [-name *name* | *name* | -addr "<a.b.c.d:port>" | "<a.b.c.d:port>" ...]
[-all]

[-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

condor_restart [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *name*]
[-addr "<a.b.c.d:port>" ...] [-all]

[-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

Description

condor_restart restarts a set of Condor daemon(s) on a set of machines. The daemon(s) will be put into a consistent state, killed, and then started anew.

If, for example, the *condor_master* needs to be restarted again with a fresh state, this is the command that should be used to do so. If the DAEMON_LIST variable in the configuration file has been changed, this command is used to restart the *condor_master* in order to see this change. The *condor_reconfigure* command cannot be used in the case where the DAEMON_LIST expression changes.

The command *condor_restart* with no arguments or with the **-subsystem *master*** option will safely shut down all running jobs and all submitted jobs from the machine(s) being restarted, then shut down all the child daemons of the *condor_master*, and then restart the *condor_master*. This, in turn, will allow the *condor_master* to start up other daemons as specified in the DAEMON_LIST configuration file entry.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 282 for further explanation.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "<*a.b.c.d:port*>" Send the command to a machine's master located at "<*a.b.c.d:port*>"

"<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-all Send the command to all machines in the pool

-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill* Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

Exit Status

condor_restart will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To restart the *condor_master* and all its children on the local host:

```
% condor_restart
```

To restart only the *condor_startd* on a named machine:

```
% condor_restart -name bluejay -subsystem startd
```

To restart a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command restarts the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_restart -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_rm

remove jobs from the Condor queue

Synopsis

condor_rm [-help | -version]

condor_rm [-debug] [-forcex] [-pool *centralmanagerhostname[:portnumber]* |
-name *scheddname*]| [-addr "<a.b.c.d:port>"] cluster...| cluster.process...| user... |
-constraint *expression* ...

condor_rm [-debug] [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]|
[-addr "<a.b.c.d:port>"] -all

Description

condor_rm removes one or more jobs from the Condor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be removed are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can remove the job.

When removing a grid job, the job may remain in the “X” state for a very long time. This is normal, as Condor is attempting to communicate with the remote scheduling system, ensuring that the job has been properly cleaned up. If it takes too long, or in rare circumstances is never removed, the job may be forced to leave the job queue by using the **-forcex** option. This forcibly removes jobs that are in the “X” state without attempting to finish any clean up at the remote scheduler.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager’s host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr "<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-forcex Force the immediate local removal of jobs in the 'X' state (only affects jobs already being removed)

cluster Remove all jobs in the specified cluster

cluster.process Remove the specific job in the cluster

user Remove jobs belonging to specified user

-constraint *expression* Remove all jobs which match the job ClassAd expression constraint

-all Remove all the jobs in the queue

General Remarks

Use the *-forcex* argument with caution, as it will remove jobs from the local queue immediately, but can "orphan" parts of the job that are running remotely and haven't yet been stopped or removed.

Examples

To remove all jobs of a user named Mary that are not currently running:

```
% condor_rm Mary -constraint Activity!="Busy\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

Exit Status

condor_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_router_history

Display the history for routed jobs

Synopsis

condor_router_history [**-h**]

condor_router_history [**--show_records**] [**--show_iwd**] [**--age** *days*] [**--days** *days*] [**--start** "YYYY-MM-DD HH:MM"]

Description

condor_router_history summarizes statistics for routed jobs over the previous 24 hours. With no command line options, statistics for run time, number of jobs completed, and number of jobs aborted are listed per route (site).

Options

- h** Display usage information and exit.
- show_records** Displays individual records in addition to the summary.
- show_iwd** Include working directory in displayed records.
- age** *days* Set the ending time of the summary to be *days* days ago.
- days** *days* Set the number of days to summarize.
- start** "YYYY-MM-DD HH:MM" Set the start time of the summary.

Exit Status

condor_router_history will exit with a status of 0 (zero) upon success, and non-zero otherwise.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_run

Submit a shell command-line as a Condor job

Synopsis

condor_run [-u *universe*] "*shell command*"

Description

condor_run bundles a shell command line into a Condor job and submits the job. The *condor_run* command waits for the Condor job to complete, writes the job's output to the terminal, and exits with the exit status of the Condor job. No output appears until the job completes.

Enclose the shell command line in double quote marks, so it may be passed to *condor_run* without modification. *condor_run* will not read input from the terminal while the job executes. If the shell command line requires input, redirect the input from a file, as illustrated by the example

```
% condor_run "myprog < input.data"
```

condor_run jobs rely on a shared file system for access to any necessary input files. The current working directory of the job must be accessible to the machine within the Condor pool where the job runs.

Specialized environment variables may be used to specify requirements for the machine where the job may run.

CONDOR_ARCH Specifies the architecture of the required platform. Values will be the same as the Arch machine ClassAd attribute.

CONDOR OPSYS Specifies the operating system of the required platform. Values will be the same as the OpSys machine ClassAd attribute.

CONDOR_REQUIREMENTS Specifies any additional requirements for the Condor job. It is recommended that the value defined for CONDOR_REQUIREMENTS be enclosed in parenthesis.

When one or more of these environment variables is specified, the job is submitted with:

```
Requirements = $CONDOR_REQUIREMENTS && Arch == $CONDOR_ARCH && \
  OpSys == $CONDOR_OPSYS
```

Without these environment variables, the job receives the default requirements expression, which requests a machine of the same platform as the machine on which *condor_run* is executed.

All environment variables set when *condor_run* is executed will be included in the environment of the Condor job.

condor_run removes the Condor job from the queue and deletes its temporary files, if *condor_run* is killed before the Condor job completes.

Options

-u *universe* Submit the job under the specified universe. The default is vanilla. While any universe may be specified, only the vanilla, standard, scheduler, and local universes result in a submit description file that may work properly.

Examples

condor_run may be used to compile an executable on a different platform. As an example, first set the environment variables for the required platform:

```
% setenv CONDOR_ARCH "SUN4u"  
% setenv CONDOR_OPSYS "SOLARIS28"
```

Then, use *condor_run* to submit the compilation as in the following three examples.

```
% condor_run "f77 -O -o myprog myprog.f"
```

or

```
% condor_run "make"
```

or

```
% condor_run "condor_compile cc -o myprog.condor myprog.c"
```

Files

condor_run creates the following temporary files in the user's working directory. The placeholder *%pid%* is replaced by the process id of *condor_run*.

- .condor_run.<pid>** A shell script containing the shell command line.
- .condor_submit.<pid>** The submit description file for the job.
- .condor_log.<pid>** The Condor job's log file; it is monitored by *condor_run*, to determine when the job exits.
- .condor_out.<pid>** The output of the Condor job before it is output to the terminal.
- .condor_error.<pid>** Any error messages for the Condor job before they are output to the terminal.

condor_run removes these files when the job completes. However, if *condor_run* fails, it is possible that these files will remain in the user's working directory, and the Condor job may remain in the queue.

General Remarks

condor_run is intended for submitting simple shell command lines to Condor. It does not provide the full functionality of *condor_submit*. Therefore, some *condor_submit* errors and system failures may not be handled correctly.

All processes specified within the single shell command line will be executed on the single machine matched with the job. Condor will not distribute multiple processes of a command line pipe across multiple machines.

condor_run will use the shell specified in the `SHELL` environment variable, if one exists. Otherwise, it will use `/bin/sh` to execute the shell command-line.

By default, *condor_run* expects Perl to be installed in `/usr/bin/perl`. If Perl is installed in another path, ask the Condor administrator to edit the path in the *condor_run* script, or explicitly call Perl from the command line:

```
% perl path-to-condor/bin/condor_run "shell-cmd"
```

Exit Status

condor_run exits with a status value of 0 (zero) upon complete success. The exit status of *condor_run* will be non-zero upon failure. The exit status in the case of a single error due to a system call will be the error number (`errno`) of the failed call.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_set_shutdown

Set a program to execute upon *condor_master* shut down

Synopsis

condor_set_shutdown [-help | -version]

condor_set_shutdown -exec *programname* [-debug] [-name *name* | *name* |
-addr "<a.b.c.d:port>" | "<a.b.c.d:port>" ...] [-all]

condor_set_shutdown -exec *programname* [-debug] [-pool *centralmanagerhostname[:portnumber]*
| -name *name*] [-addr "<a.b.c.d:port>" ...] [-all]

Description

condor_set_shutdown sets a program (typically a script) to execute when the *condor_master* daemon shuts down. The **-exec** *programname* argument is required, and specifies the program to run. The string *programname* must match the string that defines Name in the configuration variable MASTER_SHUTDOWN_<Name> in the *condor_master* daemon's configuration. If it does not match, the *condor_master* will log an error and ignore the request.

For purposes of authentication and authorization, this command requires the ADMINISTRATOR access level. See section 3.6.1 on page 282 for further explanation.

Options

-help Display usage information

-version Display version information

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "<a.b.c.d:port>" Send the command to a machine's master located at "<a.b.c.d:port>"

"<a.b.c.d:port>" Send the command to a machine located at "<a.b.c.d:port>"

-all Send the command to all machines in the pool

Exit Status

condor_set_shutdown will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To have all *condor_master* daemons run the program */bin/reboot* upon shut down, configure the *condor_master* to contain a definition similar to:

```
MASTER_SHUTDOWN_REBOOT = /sbin/reboot
```

where REBOOT is an invented name for this program that the *condor_master* will execute. On the command line, run

```
% condor_set_shutdown -exec reboot -all
% condor_off -graceful -all
```

where the string *reboot* matches the invented name.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_stats

Display historical information about the Condor pool

Synopsis

condor_stats [-f *filename*] [-orgformat] [-pool *centralmanagerhostname[:portnumber]*]
[time-range] *query-type*

Description

condor_stats displays historic information about a Condor pool. Based on the type of information requested, a query is sent to the *condor_collector* daemon, and the information received is displayed using the standard output. If the **-f** option is used, the information will be written to a file instead of to standard output. The **-pool** option can be used to get information from other pools, instead of from the local (default) pool. The *condor_stats* tool is used to query resource information (single or by platform), submitter and user information, and checkpoint server information. If a time range is not specified, the default query provides information for the previous 24 hours. Otherwise, information can be retrieved for other time ranges such as the last specified number of hours, last week, last month, or a specified date range.

The information is displayed in columns separated by tabs. The first column always represents the time, as a percentage of the range of the query. Thus the first entry will have a value close to 0.0, while the last will be close to 100.0. If the **-orgformat** option is used, the time is displayed as number of seconds since the Unix epoch. The information in the remainder of the columns depends on the query type.

Note that logging of pool history must be enabled in the *condor_collector* daemon, otherwise no information will be available.

One query type is required. If multiple queries are specified, only the last one takes effect.

Time Range Options

-lastday Get information for the last day.

-lastweek Get information for the last week.

-lastmonth Get information for the last month.

-lasthours *n* Get information for the *n* last hours.

-from *m d y* Get information for the time since the beginning of the specified date. A start date prior to the Unix epoch causes *condor_stats* to print its usage information and quit.

-to *m d y* Get information for the time up to the beginning of the specified date, instead of up to now. A finish date in the future causes *condor_stats* to print its usage information and quit.

Query Type Arguments

The query types that do not list all of a category require further specification as given by an argument.

-resourcequery *hostname* A single resource query provides information about a single machine. The information also includes the keyboard idle time (in seconds), the load average, and the machine state.

-resourcelist A query of a single list of resources to provide a list of all the machines for which the *condor_collector* daemon has historic information within the query's time range.

-resgroupquery *arch/opsys* — “Total” A query of a specified group to provide information about a group of machines based on their platform (operating system and architecture). The architecture is defined by the machine ClassAd *Arch*, and the operating system is defined by the machine ClassAd *OpSys*. The string “Total” ask for information about all platforms.

The columns displayed are the number of machines that are unclaimed, matched, claimed, preempting, and in the owner state.

-resgrouplist Queries for a list of all the group names for which the *condor_collector* has historic information within the query's time range.

-userquery *email_address/submit_machine* Query for a specific submitter on a specific machine. The information displayed includes the number of running jobs and the number of idle jobs. An example argument appears as

```
-userquery jondoe@sample.com/onemachine.sample.com
```

-userlist Queries for the list of all submitters for which the *condor_collector* daemon has historic information within the query's time range.

-usergroupquery *email_address* — “*Total*” Query for all jobs submitted by the specific user, regardless of the machine they were submitted from, or all jobs. The information displayed includes the number of running jobs and the number of idle jobs.

-usergrouplist Queries for the list of all users for which the *condor_collector* has historic information within the query’s time range.

-ckptquery *hostname* Query about a checkpoint server given its host name. The information displayed includes the number of Mbytes received, Mbytes sent, average receive bandwidth (in Kbytes/sec), and average send bandwidth (in Kbytes/sec).

-ckptlist Query for the entire list of checkpoint servers for which the *condor_collector* has historic information in the query’s time range.

Options

-f *filename* Write the information to a file instead of the standard output.

-pool *centralmanagerhostname[:portnumber]* Contact the specified central manager instead of the local one.

-orgformat Display the information in an alternate format for timing, which presents timestamps since the Unix epoch. This argument only affects the display of *resourcequery*, *resgroupquery*, *userquery*, *usergroupquery*, and *ckptquery*.

Exit Status

condor_stats will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_status

Display status of the Condor pool

Synopsis

condor_status **[-debug]** *[help options]* *[query options]* *[display options]* *[custom options]*
[name ...]

Description

condor_status is a versatile tool that may be used to monitor and query the Condor pool. The *condor_status* tool can be used to query resource information, submitter information, checkpoint server information, and daemon master information. The specific query sent and the resulting information display is controlled by the query options supplied. Queries and display formats can also be customized.

The options that may be supplied to *condor_status* belong to five groups:

- **Help options** provide information about the *condor_status* tool.
- **Query options** control the content and presentation of status information.
- **Display options** control the display of the queried information.
- **Custom options** allow the user to customize query and display information.
- **Host options** specify specific machines to be queried

At any time, only one *help option*, one *query option* and one *custom option* may be specified. Any number of *custom* and *host options* may be specified.

Options

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-help (Help option) Display usage information

-diagnose (Help option) Print out query ad without performing query

- any** (Query option) Query all ads and display their type, target type, and name
- avail** (Query option) Query *condor_startd* ads and identify resources which are available
- ckptsrvr** (Query option) Query *condor_ckpt_server* ads and display checkpoint server attributes
- claimed** (Query option) Query *condor_startd* ads and print information about claimed resources
- cod** (Query option) Display only machine ClassAds that have COD claims. Information displayed includes the claim ID, the owner of the claim, and the state of the COD claim.
- direct hostname** (Query option) Go directly to the given host name to get the ads to display
- java** (Query option) Display only Java-capable resources.
- license** (Query option) Display license attributes.
- master** (Query option) Query *condor_master* ads and display daemon master attributes
- negotiator** (Query option) Query *condor_negotiator* ads and display attributes
- pool centralmanagerhostname[:portnumber]** (Query option) Query the specified central manager using an optional port number. *condor_status* queries the machine specified by the configuration variable COLLECTOR_HOST by default.
- quill** (Query option) Display attributes of machines running Quill.
- run** (Query option) Display information about machines currently running jobs.
- schedd** (Query option) Query *condor_schedd* ads and display attributes
- server** (Query option) Query *condor_startd* ads and display resource attributes
- startd** (Query option) Query *condor_startd* ads

- state** (Query option) Query *condor_startd* ads and display resource state information
- storage** (Query option) Display attributes of machines with network storage resources.
- submitters** (Query option) Query ads sent by submitters and display important submitter attributes
- subsystem *type*** (Query option) If *type* is one of *collector*, *negotiator*, *master*, *schedd*, *startd*, or *quill*, then behavior is the same as the query option without the **-subsystem** option. For example, **-subsystem collector** is the same as **-collector**. A value of *type* of *CkptServer*, *Machine*, *DaemonMaster*, or *Scheduler* targets that type of ClassAd.
- vm** (Query option) Query *condor_startd* ClassAds, and display only VM-enabled machines. Information displayed includes the machine name, the virtual machine software version, the state of machine, the virtual machine memory, and the type of networking.
- expert** (Display option) Display shortened error messages
- long** (Display option) Display entire ClassAds (same as **-verbose**)
- sort *attr*** (Display option) Display entries in ascending order based on the value of the named attribute
- total** (Display option) Display totals only
- verbose** (Display option) Display entire ClassAds. Implies that totals will not be displayed.
- xml** (Display option) Display entire ClassAds, in XML format. The XML format is fully defined at <http://www.cs.wisc.edu/condor/classad/refman/>.
- constraint *const*** (Custom option) Add constraint expression. See section 4.1 for details on writing expressions.
- format *fmt attr*** (Custom option) Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)` style conversion specifier. Attributes must be from the resource ClassAd. Expressions are ClassAd expressions and may refer to attributes in the resource ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. The conversion specifier must match the type of the attribute or expression. `%s`

is suitable for strings such as `Name`, `%d` for integers such as `LastHeardFrom`, and `%f` for floating point numbers such as `LoadAvg`. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, one may include other text that will be reproduced directly. A format without any conversion specifiers may be specified, but an attribute is still required. Include `\n` to specify a line break.

General Remarks

- The default output from *condor_status* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the Condor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_status* is strongly encouraged to use the **-format** option (described above).
- The information obtained from *condor_startd* and *condor_schedd* daemons may sometimes appear to be inconsistent. This is normal since *condor_startd* and *condor_schedd* daemons update the Condor manager at different rates, and since there is a delay as information propagates through the network and the system.
- Note that the `ActivityTime` in the `Idle` state is *not* the amount of time that the machine has been idle. See the section on *condor_startd* states in the *Administrator's Manual* for more information.
- When using *condor_status* on a pool with SMP machines, you can either provide the host name, in which case you will get back information about all slots that are represented on that host, or you can list specific slots by name. See the examples below for details.
- If you specify host names, without domains, Condor will automatically try to resolve those host names into fully qualified host names for you. This also works when specifying specific nodes of an SMP machine. In this case, everything after the “@” sign is treated as a host name and that is what is resolved.
- You can use the **-direct** option in conjunction with almost any other set of options. However, at this time, the only daemon that will allow direct queries for its ad(s) is the *condor_startd*. So, the only options currently not supported with **-direct** are **-schedd** and **-master**. Most other options use *startd* ads for their information, so they work seamlessly with **-direct**. The only other restriction on **-direct** is that you may only use 1 **-direct** option at a time. If you want to query information directly from multiple hosts, you must run *condor_status* multiple times.
- Unless you use the local host name with **-direct**, *condor_status* will still have to contact a collector to find the address where the specified daemon is listening. So, using a **-pool** option in conjunction with **-direct** just tells *condor_status* which collector to query to find the address of the daemon you want. The information actually displayed will still be retrieved directly from the daemon you specified as the argument to **-direct**.

Examples

Example 1 To view information from all nodes of an SMP machine, use only the host name. For example, if you had a 4-CPU machine, named `vulture.cs.wisc.edu`, you might see

```
% condor_status vulture
```

Name	OpSys	Arch	State	Activity	LoadAv	Mem	ActvtyTime
slot1@vulture.cs.w	LINUX	INTEL	Claimed	Busy	1.050	512	0+01:47:42
slot2@vulture.cs.w	LINUX	INTEL	Claimed	Busy	1.000	512	0+01:48:19
slot3@vulture.cs.w	LINUX	INTEL	Unclaimed	Idle	0.070	512	1+11:05:32
slot4@vulture.cs.w	LINUX	INTEL	Unclaimed	Idle	0.000	512	1+11:05:34

	Total	Owner	Claimed	Unclaimed	Matched	Preempting	Backfill
INTEL/LINUX	4	0	2	2	0	0	0
Total	4	0	2	2	0	0	0

Example 2 To view information from a specific nodes of an SMP machine, specify the node directly. You do this by providing the name of the slot. This has the form `slot#@hostname`. For example:

```
% condor_status slot3@vulture
```

Name	OpSys	Arch	State	Activity	LoadAv	Mem	ActvtyTime
slot3@vulture.cs.w	LINUX	INTEL	Unclaimed	Idle	0.070	512	1+11:10:32

	Total	Owner	Claimed	Unclaimed	Matched	Preempting	Backfill
INTEL/LINUX	1	0	0	1	0	0	0
Total	1	0	0	1	0	0	0

Constraint option examples

To use the constraint option to see all machines with the OpSys of "LINUX", use

```
% condor_status -constraint OpSys=="LINUX\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

To see all machines that are currently in the Idle state, use

```
% condor_status -constraint State=="Idle\"
```

To see all machines that are bench marked to have a MIPS rating of more than 750, use

```
% condor_status -constraint 'Mips>750'
```

-cod option example

The **-cod** option displays the status of COD claims within a given Condor pool.

Name	ID	ClaimState	TimeInState	RemoteUser	JobId	Keyword
astro.cs.wi	COD1	Idle	0+00:00:04	wright		
chopin.cs.w	COD1	Running	0+00:02:05	wright	3.0	fractgen
chopin.cs.w	COD2	Suspended	0+00:10:21	wright	4.0	fractgen

	Total	Idle	Running	Suspended	Vacating	Killing
INTEL/LINUX	3	1	1	1	0	0
Total	3	1	1	1	0	0

Exit Status

condor_status will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_store_cred

securely stash user's password

Synopsis

condor_store_cred [-help]

condor_store_cred add [-c | -u username][-p password] [-n machinename] [-f filename]

condor_store_cred delete [-c | -u username][-n machinename]

condor_store_cred query [-c | -u username][-n machinename]

Description

On a Windows machine, *condor_store_cred* stores the password of a user/domain pair securely in the Windows registry. Using this stored password, Condor is able to run jobs with the user ID of the submitting user. In addition, Condor uses this password to acquire the submitting user's credentials when writing output or log files. The password is stored in the same manner as the system does when setting or changing account passwords. When *condor_store_cred* is invoked, it contacts the *condor_schedd* daemon to carry out the requested operations on behalf of the user. This is necessary since registry keys are accessible only by the Windows SYSTEM account, not by administrators or other users.

On a Unix machine, *condor_store_cred* is used to manage the pool password, placed in a file specified by the `SEC_PASSWORD_FILE` configuration variable, and for use in password authentication among Condor daemons.

The password is stashed in a persistent manner; it is maintained across system reboots.

The *add* argument stores the current user's password securely in the registry. The user is prompted to enter the password twice for confirmation, and characters are not echoed. If there is already a password stashed, the old password will be overwritten by the new password.

The *delete* deletes the current password, if it exists.

The *query* reports whether the password is stored or not.

Options

-c Apply the option to the pool password.

-f *filename* For Unix machines only, generates a pool password file named *filename* that may be used with the PASSWORD authentication method.

-help Displays a brief summary of command options.

-n *machinename* Apply the command on the given machine.

-p *password* Stores given password, rather than prompting.

-u *username* Specify the user name.

Exit Status

condor_store_cred will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_submit

Queue jobs for execution under Condor

Synopsis

condor_submit [-verbose] [-unused] [-name *schedd_name*] [-remote *schedd_name*]
[-pool *pool_name*] [-disable] [-password *passphrase*] [-debug] [-append *command ...*][-spool]
[-dump *filename*] [*submit description file*]

Description

condor_submit is the program for submitting jobs for execution under Condor. *condor_submit* requires a submit description file which contains commands to direct the queuing of jobs. One submit description file may contain specifications for the queuing of many Condor jobs at once. A single invocation of *condor_submit* may cause one or more clusters. A cluster is a set of jobs specified in the submit description file between **queue** commands for which the executable is not changed. It is advantageous to submit multiple jobs as a single cluster because:

- Only one copy of the checkpoint file is needed to represent all jobs in a cluster until they begin execution.
- There is much less overhead involved for Condor to start the next job in a cluster than for Condor to start a new cluster. This can make a big difference when submitting lots of short jobs.

Multiple clusters may be specified within a single submit description file. Each cluster must specify a single executable.

The job ClassAd attribute `ClusterId` identifies a cluster. See specifics for this attribute in the Appendix on page 879.

Note that submission of jobs from a Windows machine requires a stashed password to allow Condor to impersonate the user submitting the job. To stash a password, use the *condor_store_cred* command. See the manual page at page 793 for details.

SUBMIT DESCRIPTION FILE COMMANDS

Each submit description file describes one cluster of jobs to be placed in the Condor execution pool. All jobs in a cluster must share the same executable, but they may have different input and output files, and different program arguments. The submit description file is the only command-line argument to *condor_submit*. If the submit description file argument is omitted, *condor_submit* will read the submit description from standard input.

The submit description file must contain one *executable* command and at least one *queue* command. All of the other commands have default actions.

The commands which can appear in the submit description file are numerous. They are listed here in alphabetical order by category.

BASIC COMMANDS

arguments = <argument.list> List of arguments to be supplied to the program on the command line. In the Java Universe, the first argument must be the name of the class containing `main`.

There are two permissible formats for specifying arguments. The new syntax supports uniform quoting of spaces within arguments; the old syntax supports spaces in arguments only in special circumstances.

In the old syntax, arguments are delimited (separated) by space characters. Double-quotes must be escaped with a backslash (i.e. put a backslash in front of each double-quote).

Further interpretation of the argument string differs depending on the operating system. On Windows, your argument string is simply passed verbatim (other than the backslash in front of double-quotes) to the Windows application. Most Windows applications will allow you to put spaces within an argument value by surrounding the argument with double-quotes. In all other cases, there is no further interpretation of the arguments.

Example:

```
arguments = one \"two\" 'three'
```

Produces in Unix vanilla universe:

```
argument 1: one
argument 2: "two"
argument 3: 'three'
```

Here are the rules for using the new syntax:

1. Put double quotes around the entire argument string. This distinguishes the new syntax from the old, because these double-quotes are not escaped with backslashes, as required in the old syntax. Any literal double-quotes within the string must be escaped by repeating them.
2. Use white space (e.g. spaces or tabs) to separate arguments.
3. To put any white space in an argument, you must surround the space and as much of the surrounding argument as you like with single-quotes.
4. To insert a literal single-quote, you must repeat it anywhere inside of a single-quoted section.

Example:

```
arguments = "one "two" 'spacey 'quoted' argument"
```

Produces:

```
argument 1: one
argument 2: "two"
argument 3: spacey 'quoted' argument
```

Notice that in the new syntax, backslash has no special meaning. This is for the convenience of Windows users.

environment = <parameter_list> List of environment variables.

There are two different formats for specifying the environment variables: the old format and the new format. The old format is retained for backward-compatibility. It suffers from a platform-dependent syntax and the inability to insert some special characters into the environment.

The new syntax for specifying environment values:

1. Put double quote marks around the entire argument string. This distinguishes the new syntax from the old. The old syntax does not have double quote marks around it. Any literal double quote marks within the string must be escaped by repeating the double quote mark.
2. Each environment entry has the form


```
<name>=<value>
```
3. Use white space (space or tab characters) to separate environment entries.
4. To put any white space in an environment entry, surround the space and as much of the surrounding entry as desired with single quote marks.
5. To insert a literal single quote mark, repeat the single quote mark anywhere inside of a section surrounded by single quote marks.

Example:

```
environment = "one=1 two="2" three='spacey 'quoted' value'"
```

Produces the following environment entries:

```
one=1
two="2"
three=spacey 'quoted' value
```

Under the old syntax, there are no double quote marks surrounding the environment specification. Each environment entry remains of the form

<name>=<value>

Under Unix, list multiple environment entries by separating them with a semicolon (;). Under Windows, separate multiple entries with a vertical bar (|). There is no way to insert a literal semicolon under Unix or a literal vertical bar under Windows. Note that spaces are accepted, but rarely desired, characters within parameter names and values, because they are treated as literal characters, not separators or ignored white space. Place spaces within the parameter list only if required.

A Unix example:

```
environment = one=1;two=2;three="quotes have no 'special' meaning"
```

This produces the following:

```
one=1
two=2
three="quotes have no 'special' meaning"
```

If the environment is set with the **environment** command *and* **getenv** is also set to true, values specified with **environment** override values in the submitter's environment (regardless of the order of the **environment** and **getenv** commands).

error = <pathname> A path and file name used by Condor to capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, error messages are ignored for submission to a Windows machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another. The error file and the output file should not be the same file as the outputs will overwrite each other or be lost. For grid universe jobs, **error** may be a URL that the Globus tool *globus_url_copy* understands.

executable = <pathname> An optional path and a required file name of the executable file for this job cluster. Only one **executable** command within a submit description file is guaranteed to work properly. More than one often works.

If no path or a relative path is used, then the executable file is presumed to be relative to the current working directory of the user as the *condor_submit* command is issued.

If submitting into the standard universe, then the named executable must have been re-linked with the Condor libraries (such as via the *condor_compile* command). If submitting into the vanilla universe (the default), then the named executable need not be re-linked and can be any process which can run in the background (shell scripts work fine as well). If submitting into the Java universe, then the argument must be a compiled `.class` file.

getenv = <True | False> If **getenv** is set to `True`, then *condor_submit* will copy all of the user's current shell environment variables at the time of job submission into the job ClassAd. The job will therefore execute with the same set of environment variables that the user had at submit time. Defaults to `False`.

If the environment is set with the **environment** command *and* **getenv** is also set to true, values specified with **environment** override values in the submitter's environment (regardless of the order of the **environment** and **getenv** commands).

input = <pathname> Condor assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (**stdin**, **stdout**, and **stderr**), must refer to files. Thus, the file name specified with **input** should contain any keyboard input the program requires (that is, this file becomes **stdin**). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, input is ignored for submission to a Windows machine. For grid universe jobs, **input** may be a URL that the Globus tool *globus_url_copy* understands.

Note that this command does *not* refer to the command-line arguments of the program. The command-line arguments are specified by the **arguments** command.

log = <pathname> Use **log** to specify a file name where Condor will write a log file of what is happening with this job cluster. For example, Condor will place a log entry into this file when and where the job begins running, when the job produces a checkpoint, or moves (migrates) to another machine, and when the job completes. Most users find specifying a **log** file to be handy; its use is recommended. If no **log** entry is specified, Condor does not create a log for this cluster.

log_xml = <True | False> If **log_xml** is **True**, then the log file will be written in ClassAd XML. If not specified, XML is not used. Note that the file is an XML fragment; it is missing the file header and footer. Do not mix XML and non-XML within a single file. If multiple jobs write to a single log file, ensure that all of the jobs specify this option in the same way.

notification = <Always | Complete | Error | Never> Owners of Condor jobs are notified by e-mail when certain events occur. If defined by *Always*, the owner will be notified whenever the job produces a checkpoint, as well as when the job completes. If defined by *Complete* (the default), the owner will be notified when the job terminates. If defined by *Error*, the owner will only be notified if the job terminates abnormally. If defined by *Never*, the owner will not receive e-mail, regardless to what happens to the job. The statistics included in the e-mail are documented in section 2.6.7 on page 45.

notify_user = <email-address> Used to specify the e-mail address to use when Condor sends e-mail about a job. If not specified, Condor defaults to using the e-mail address defined by

```
job-owner@UID_DOMAIN
```

where the configuration variable `UID_DOMAIN` is specified by the Condor site administrator. If `UID_DOMAIN` has not been specified, Condor sends the e-mail to:

```
job-owner@submit-machine-name
```

output = <pathname> The **output** file captures any information the program would ordinarily write to the screen (that is, this file becomes **stdout**). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, output is ignored

for submission to a Windows machine. Multiple jobs should not use the same output file, since this will cause one job to overwrite the output of another. The output file and the error file should not be the same file as the outputs will overwrite each other or be lost. For grid universe jobs, **output** may be a URL that the Globus tool *globus_url_copy* understands.

Note that if a program explicitly opens and writes to a file, that file should *not* be specified as the **output** file.

priority = <**integer**> A Condor job priority can be any integer, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority. Note that this priority is on a per user basis. One user with many jobs may use this command to order his/her own jobs, and this will have no effect on whether or not these jobs will run ahead of another user's jobs.

queue [**number-of-procs**] Places one or more copies of the job into the Condor queue. The optional argument *number-of-procs* specifies how many times to submit the job to the queue, and it defaults to 1. If desired, any commands may be placed between subsequent **queue** commands, such as new **input**, **output**, **error**, **initialdir**, or **arguments** commands. This is handy when submitting multiple runs into one cluster with one submit description file.

universe = <**vanilla** | **standard** | **scheduler** | **local** | **grid** | **mpi** | **java** | **vm**> Specifies which Condor Universe to use when running this job. The Condor Universe specifies a Condor execution environment. The **standard** Universe tells Condor that this job has been re-linked via *condor_compile* with the Condor libraries and therefore supports checkpointing and remote system calls. The **vanilla** Universe is the default (except where the configuration variable `DEFAULT_UNIVERSE` defines it otherwise), and is an execution environment for jobs which have not been linked with the Condor libraries. *Note:* Use the **vanilla** Universe to submit shell scripts to Condor. The **scheduler** is for a job that should act as a metascheduler. The **grid** universe forwards the job to an external job management system. Further specification of the **grid** universe is done with the **grid_resource** command. The **mpi** universe is for running mpi jobs made with the MPICH package. The **java** universe is for programs written to the Java Virtual Machine. The **vm** universe facilitates the execution of a virtual machine.

COMMANDS FOR MATCHMAKING

rank = <**ClassAd Float Expression**> A ClassAd Floating-Point expression that states how to rank machines which have already met the requirements expression. Essentially, rank expresses preference. A higher numeric value equals better rank. Condor will give the job the machine with the highest rank. For example,

```
requirements = Memory > 60
rank = Memory
```

asks Condor to find all available machines with more than 60 megabytes of memory and give to the job the machine with the most amount of memory. See section 2.5.2 within the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

requirements = <ClassAd Boolean Expression> The **requirements** command is a boolean ClassAd expression which uses C-like operators. In order for any job in this cluster to run on a given machine, this requirements expression must evaluate to true on the given machine. For example, to require that whatever machine executes a Condor job has a least 64 Meg of RAM and has a MIPS performance rating greater than 45, use:

```
requirements = Memory >= 64 && Mips > 45
```

For scheduler and local universe jobs, the requirements expression is evaluated against the Scheduler ClassAd which represents the the *condor_schedd* daemon running on the submit machine, rather than a remote machine. Like all commands in the submit description file, if multiple requirements commands are present, all but the last one are ignored. By default, *condor_submit* appends the following clauses to the requirements expression:

1. Arch and OpSys are set equal to the Arch and OpSys of the submit machine. In other words: unless you request otherwise, Condor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.
2. Disk >= DiskUsage. The DiskUsage attribute is initialized to the size of the executable plus the size of any files specified in a **transfer_input_files** command. It exists to ensure there is enough disk space on the target machine for Condor to copy over both the executable and needed input files. The DiskUsage attribute represents the maximum amount of total disk space required by the job in kilobytes. Condor automatically updates the DiskUsage attribute approximately every 20 minutes while the job runs with the amount of space being used by the job on the execute machine.
3. (Memory * 1024) >= ImageSize. To ensure the target machine has enough memory to run your job.
4. If Universe is set to Vanilla, FileSystemDomain is set equal to the submit machine's FileSystemDomain.

View the requirements of a job which has already been submitted (along with everything else about the job ClassAd) with the command *condor_q -l*; see the command reference for *condor_q* on page 750. Also, see the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

FILE TRANSFER COMMANDS

should_transfer_files = <YES | NO | IF_NEEDED > The **should_transfer_files** setting is used to define if Condor should transfer files to and from the remote machine where the job runs. The file transfer mechanism is used to run jobs which are not in the standard universe (and can therefore use remote system calls for file access) on machines which do not have a shared file system with the submit machine. **should_transfer_files** equal to *YES* will cause Condor to always transfer files for the job. *NO* disables Condor's file transfer mechanism. *IF_NEEDED* will not transfer files for the job if it is matched with a resource in the same FileSystemDomain as the submit machine (and therefore, on a machine with the same shared file system). If the job is matched with a remote resource in a different FileSystemDomain, Condor will transfer the necessary files.

If defining **should_transfer_files** you *must* also define **when_to_transfer_output** (described below). For more information about this and other settings related to transferring files, see section 2.5.4 on page 26.

Note that **should_transfer_files** is not supported for jobs submitted to the grid universe.

stream_error = <True | False> If True, then stderr is streamed back to the machine from which the job was submitted. If False, stderr is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferErr` is False. The default value is True in the grid universe and False otherwise. This command must be used in conjunction with **error**, otherwise stderr will sent to /dev/null on Unix machines and ignored on Windows machines.

stream_input = <True | False> If True, then stdin is streamed from the machine on which the job was submitted. The default value is False. The command is only relevant for jobs submitted to the vanilla or java universes, and it is ignored by the grid universe. This command must be used in conjunction with **input**, otherwise stdin will be /dev/null on Unix machines and ignored on Windows machines.

stream_output = <True | False> If True, then stdout is streamed back to the machine from which the job was submitted. If False, stdout is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferOut` is False. The default value is True in the grid universe and False otherwise. This command must be used in conjunction with **output**, otherwise stdout will sent to /dev/null on Unix machines and ignored on Windows machines.

transfer_executable = <True | False> This command is applicable to jobs submitted to the grid, vanilla, and MPI universes. If **transfer_executable** is set to False, then Condor looks for the executable on the remote machine, and does not transfer the executable over. This is useful for an already pre-staged executable; Condor behaves more like rsh. The default value is True.

transfer_input_files = < file1,file2,file... > A comma-delimited list of all the files to be transferred into the working directory for the job before the job is started. By default, the file specified in the **executable** command and any file specified in the **input** command (for example, `stdin`) are transferred.

Only the transfer of files is available; the transfer of subdirectories is not supported.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 26.

transfer_output_files = < file1,file2,file... > This command forms an explicit list of output files to be transferred back from the temporary working directory on the execute machine to the submit machine. Most of the time, there is no need to use this command. Other than for grid universe jobs, if **transfer_output_files** is not specified, Condor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job. This is usually the desired behavior. Explicitly listing output files is typically only done when the job creates many files, and the user wants to keep a subset of those files. If there are multiple files, they must be delimited with commas. **WARNING:** Do not specify

transfer_output_files in the submit description file unless there is a really good reason – it is best to let Condor figure things out by itself based upon what the job produces.

For grid universe jobs, to have files other than standard output and standard error transferred from the execute machine back to the submit machine, do use **transfer_output_files**, listing all files to be transferred. These files are found on the execute machine in the working directory of the job.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 26.

transfer_output_remaps = < “ **name** = **newname** ; **name2** = **newname2** ... ” > This specifies the name (and optionally path) to use when downloading output files from the completed job. Normally, output files are transferred back to the initial working directory with the same name they had in the execution directory. This gives you the option to save them with a different path or name. If you specify a relative path, the final path will be relative to the job’s initial working directory.

name describes an output file name produced by your job, and *newname* describes the file name it should be downloaded to. Multiple remaps can be specified by separating each with a semicolon. If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

when.to.transfer.output = < **ON_EXIT** | **ON_EXIT_OR_EVICT** > Setting

when.to.transfer.output equal to *ON_EXIT* will cause Condor to transfer the job’s output files back to the submitting machine only when the job completes (exits on its own).

The *ON_EXIT_OR_EVICT* option is intended for fault tolerant jobs which periodically save their own state and can restart where they left off. In this case, files are spooled to the submit machine any time the job leaves a remote site, either because it exited on its own, or was evicted by the Condor system for any reason prior to job completion. The files spooled back are placed in a directory defined by the value of the *SPOOL* configuration variable. Any output files transferred back to the submit machine are automatically sent back out again as input files if the job restarts.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 26.

POLICY COMMANDS

hold = <**True** | **False**> If **hold** is set to **True**, then the job will be submitted in the hold state. Jobs in the hold state will not run until released by *condor_release*. Defaults to **false**.

leave.in.queue = <**ClassAd Boolean Expression**> When the ClassAd Expression evaluates to **True**, the job is not removed from the queue upon completion. This allows the user of a remotely spooled job to retrieve output files in cases where Condor would have removed them as part of the cleanup associated with completion. The job will only exit the queue once it has been marked for removal (via *condor_rm*, for example) and the **leave.in.queue** expression has become **False**. **leave.in.queue** defaults to **False**.

on_exit_hold = <ClassAd Boolean Expression> This expression is checked when the job exits and if true, places the job on hold. If false then nothing happens and the `on_exit_remove` expression is checked to determine if that needs to be applied.

For example: Suppose a job is known to run for a minimum of an hour. If the job exits after less than an hour, the job should be placed on hold and an e-mail notification sent, instead of being allowed to leave the queue.

```
on_exit_hold = (CurrentTime - JobStartDate) < (60 * $(MINUTE))
```

This expression places the job on hold if it exits for any reason before running for an hour. An e-mail will be sent to the user explaining that the job was placed on hold because this expression became True.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

If left unspecified, this will default to False.

This expression is available for the vanilla, java, parallel, mpi, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

on_exit_remove = <ClassAd Boolean Expression> This expression is checked when the job exits and if true, then it allows the job to leave the queue normally. If false, then the job is placed back into the Idle state. If the user job runs under the vanilla universe, then the job restarts from the beginning. If the user job runs under the standard universe, then it continues from where it left off, using the last checkpoint.

For example, suppose you have a job that occasionally segfaults, but you know if you run the job again with the same data, chances are that the will finish successfully. This is how you would represent that with `on_exit_remove` (assuming the signal identifier for segmentation fault is 11 on the platform where your job will be running):

```
on_exit_remove = (ExitBySignal == False) || (ExitSignal != 11)
```

This expression will only let the job leave the queue if the job was not killed by a signal (it exited normally on its own) or if it was killed by a signal other than 11 (representing segmentation fault). So, if it was killed by signal 11, it will stay in the job queue. In any other case of the job exiting, the job will leave the queue as it normally would have done.

As another example, if your job should only leave the queue if it exited on its own with status 0, you would use this `on_exit_remove` expression:

```
on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)
```

If the job was killed by a signal or exited with a non-zero exit status, Condor would leave the job in the queue to run again.

If left unspecified, the `on_exit_remove` expression will default to True.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

This expression is available for the vanilla, java, parallel, mpi, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks these periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

periodic_hold = <ClassAd Boolean Expression> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL`. If it becomes true, the job will be placed on hold. If unspecified, the default value is `False`.

See the Examples section for an example of a `periodic_*` expression.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

This expression is available for the vanilla, java, parallel, mpi, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd*, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

periodic_release = <ClassAd Boolean Expression> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL` while the job is in the Hold state. If the expression becomes `True`, the job will be released.

This expression is available for the vanilla, java, parallel, mpi, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

periodic_remove = <ClassAd Boolean Expression> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL`. If it becomes `True`, the job is removed from the queue. If unspecified, the default value is `False`.

See the Examples section for an example of a `periodic_*` expression.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions. So, the `periodic_remove` expression takes precedent over the `on_exit_remove` expression, if the two describe conflicting actions.

This expression is available for the vanilla, java, parallel, mpi, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd*, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

next_job_start_delay = <ClassAd Boolean Expression> This expression specifies the number of seconds to delay after starting up this job before the next job is started. The maximum allowed delay is specified by the Condor configuration variable `MAX_NEXT_JOB_START_DELAY`, which defaults to 10 minutes. Currently, this command does not apply to **scheduler** or **local** universe jobs.

COMMANDS SPECIFIC TO THE STANDARD UNIVERSE

allow_startup_script = <True | False> If True, a standard universe job will execute a script instead of submitting the job, and the consistency check to see if the executable has been linked using `condor_compile` is omitted. The **executable** command within the submit description file specifies the name of the script. The script is used to do preprocessing before the job is submitted. The shell script ends with an `exec` of the job executable, such that the process id of the executable is the same as that of the shell script. Here is an example script that gets a copy of a machine-specific executable before the `exec`.

```
#!/bin/sh

# get the host name of the machine
$host=`uname -n`

# grab a standard universe executable designed specifically
# for this host
scp elsewhere@cs.wisc.edu:${host} executable

# The PID MUST stay the same, so exec the new standard universe process.
exec executable ${1+"$@"}
```

If this command is not present (defined), then the value defaults to false.

append_files = `file1, file2, ...` If your job attempts to access a file mentioned in this list, Condor will force all writes to that file to be appended to the end. Furthermore, `condor_submit` will not truncate it. This list uses the same syntax as `compress_files`, shown above.

This option may yield some surprising results. If several jobs attempt to write to the same file, their output may be intermixed. If a job is evicted from one or more machines during the course of its lifetime, such an output file might contain several copies of the results. This option should be only be used when you wish a certain file to be treated as a running log instead of a precise result.

This option only applies to standard-universe jobs.

buffer_files = < “ `name = (size,block-size) ; name2 = (size,block-size) ...` ” >

buffer_size = <bytes-in-buffer>

buffer_block_size = <bytes-in-block> Condor keeps a buffer of recently-used data for each file a job accesses. This buffer is used both to cache commonly-used data and to consolidate small

reads and writes into larger operations that get better throughput. The default settings should produce reasonable results for most programs.

These options only apply to standard-universe jobs.

If needed, you may set the buffer controls individually for each file using the `buffer_files` option. For example, to set the buffer size to 1 Mbyte and the block size to 256 Kbytes for the file `input.data`, use this command:

```
buffer_files = "input.data=(1000000,256000)"
```

Alternatively, you may use these two options to set the default sizes for all files used by your job:

```
buffer_size = 1000000
buffer_block_size = 256000
```

If you do not set these, Condor will use the values given by these two configuration file macros:

```
DEFAULT_IO_BUFFER_SIZE = 1000000
DEFAULT_IO_BUFFER_BLOCK_SIZE = 256000
```

Finally, if no other settings are present, Condor will use a buffer of 512 Kbytes and a block size of 32 Kbytes.

compress_files = file1, file2, ... If your job attempts to access any of the files mentioned in this list, Condor will automatically compress them (if writing) or decompress them (if reading). The compress format is the same as used by GNU gzip.

The files given in this list may be simple file names or complete paths and may include `*` as a wild card. For example, this list causes the file `/tmp/data.gz`, any file named `event.gz`, and any file ending in `.gzip` to be automatically compressed or decompressed as needed:

```
compress_files = /tmp/data.gz, event.gz, *.gzip
```

Due to the nature of the compression format, compressed files must only be accessed sequentially. Random access reading is allowed but is very slow, while random access writing is simply not possible. This restriction may be avoided by using both `compress_files` and `fetch_files` at the same time. When this is done, a file is kept in the decompressed state at the execution machine, but is compressed for transfer to its original location.

This option only applies to standard universe jobs.

fetch_files = file1, file2, ... If your job attempts to access a file mentioned in this list, Condor will automatically copy the whole file to the executing machine, where it can be accessed quickly. When your job closes the file, it will be copied back to its original location. This list uses the same syntax as `compress_files`, shown above.

This option only applies to standard universe jobs.

file_remaps = < “**name** = **newname** ; **name2** = **newname2** ... ”> Directs Condor to use a new file name in place of an old one. *name* describes a file name that your job may attempt to open, and *newname* describes the file name it should be replaced with. *newname* may include an optional leading access specifier, *local:* or *remote:*. If left unspecified, the default access specifier is *remote:*. Multiple remaps can be specified by separating each with a semicolon.

This option only applies to standard universe jobs.

If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

Example One: Suppose that your job reads a file named `dataset.1`. To instruct Condor to force your job to read `other.dataset` instead, add this to the submit file:

```
file_remaps = "dataset.1=other.dataset"
```

Example Two: Suppose that you run many jobs which all read in the same large file, called `very.big`. If this file can be found in the same place on a local disk in every machine in the pool, (say `/bigdisk/bigfile`), you can instruct Condor of this fact by remapping `very.big` to `/bigdisk/bigfile` and specifying that the file is to be read locally, which will be much faster than reading over the network.

```
file_remaps = "very.big = local:/bigdisk/bigfile"
```

Example Three: Several remaps can be applied at once by separating each with a semicolon.

```
file_remaps = "very.big = local:/bigdisk/bigfile ; dataset.1 = other.dataset"
```

local_files = **file1, file2, ...** If your job attempts to access a file mentioned in this list, Condor will cause it to be read or written at the execution machine. This is most useful for temporary files not used for input or output. This list uses the same syntax as `compress_files`, shown above.

```
local_files = /tmp/*
```

This option only applies to standard universe jobs.

want_remote_io = <**True** | **False**> This option controls how a file is opened and manipulated in a standard universe job. If this option is true, which is the default, then the *condor_shadow* makes all decisions about how each and every file should be opened by the executing job. This entails a network round trip (or more) from the job to the *condor_shadow* and back again for every single `open()` in addition to other needed information about the file. If set to false, then when the job queries the *condor_shadow* for the first time about how to open a file, the *condor_shadow* will inform the job to automatically perform all of its file manipulation on the local file system on the execute machine and any file remapping will be ignored. This means that there **must** be a shared file system (such as NFS or AFS) between the execute machine and the submit machine and that **ALL** paths that the job could open on the execute machine must be valid. The ability of the standard universe job to checkpoint, possibly to a checkpoint server, is not affected by this attribute. However, when the job resumes it will be expecting the same file system conditions that were present when the job checkpointed.

COMMANDS FOR THE GRID

amazon_ami_id = **<Amazon EC2 AMI ID>** AMI identifier of the VM image to run for **amazon** jobs.

amazon_instance_type = **<VM Type>** Identifier for the type of VM desired for an **amazon** job. The default value is "m1.small".

amazon_keypair_file = **<pathname>** The complete path and filename of a file into which Condor will write an ssh key for use with **amazon** jobs. The key can be used to ssh into the virtual machine once it is running.

amazon_private_key = **<pathname>** Used for **amazon** jobs. Path and filename of a file containing the private key to be used to authenticate with Amazon's EC2 service via SOAP.

amazon_public_key = **<pathname>** Used for **amazon** jobs. Path and filename of a file containing the public X509 certificate to be used to authenticate with Amazon's EC2 service via SOAP.

amazon_security_groups = **group1, group2, ...** Used for **amazon** jobs. A list of Amazon EC2 security group names, which should be associated with the job.

amazon_user_data = **<data>** Used for **amazon** jobs. A block of data that can be accessed by the virtual machine job inside Amazon EC2.

amazon_user_data_file = **<pathname>** Used for **amazon** jobs. A file containing data that can be accessed by the virtual machine job inside Amazon EC2.

globus_rematch = **<ClassAd Boolean Expression>** This expression is evaluated by the *condor_gridmanager* whenever:

1. the **globus_resubmit** expression evaluates to True
2. the *condor_gridmanager* decides it needs to retry a submission (as when a previous submission failed to commit)

If **globus_rematch** evaluates to True, then *before* the job is submitted again to globus, the *condor_gridmanager* will request that the *condor_schedd* daemon renegotiate with the matchmaker (the *condor_negotiator*). The result is this job will be matched again.

globus_resubmit = **<ClassAd Boolean Expression>** The expression is evaluated by the *condor_gridmanager* each time the *condor_gridmanager* gets a job ad to manage. Therefore, the expression is evaluated:

1. when a grid universe job is first submitted to Condor-G
2. when a grid universe job is released from the hold state
3. when Condor-G is restarted (specifically, whenever the *condor_gridmanager* is restarted)

If the expression evaluates to True, then any previous submission to the grid universe will be forgotten and this job will be submitted again as a fresh submission to the grid universe. This may be useful if there is a desire to give up on a previous submission and try again. Note that this may result in the same job running more than once. Do not treat this operation lightly.

globus_rsl = <**RSL-string**> Used to provide any additional Globus RSL string attributes which are not covered by other submit description file commands or job attributes. Used for **grid universe** jobs, where the grid resource has a **grid-type-string** of **gt2**.

globus_xml = <**XML-string**> Used to provide any additional attributes in the GRAM XML job description that Condor writes which are not covered by regular submit description file parameters. Used for grid type **gt4** jobs.

grid_resource = <**grid-type-string**> <**grid-specific-parameter-list**> For each **grid-type-string** value, there are further type-specific values that must be specified. This submit description file command allows each to be given in a space-separated list. Allowable **grid-type-string** values are **amazon**, **gt2**, **gt4**, **condor**, **lsf**, **nordugrid**, **pbs**, and **unicore**. See section 5.3 for details on the variety of grid types.

For a **grid-type-string** of **amazon**, no additional parameters are used. See section 5.3.7 for details.

For a **grid-type-string** of **condor**, the first parameter is the name of the remote *condor_schedd* daemon. The second parameter is the name of the pool to which the remote *condor_schedd* daemon belongs. See section 5.3.1 for details.

For a **grid-type-string** of **gt2**, the single parameter is the name of the pre-WS GRAM resource to be used. See section 5.3.2 for details.

For a **grid-type-string** of **gt4**, the first parameter is the name of the WS GRAM service to be used. The second parameter is the name of WS resource to be used (usually the name of the back-end scheduler). See section 5.3.2 for details.

For a **grid-type-string** of **lsf**, no additional parameters are used. See section 5.3.6 for details.

For a **grid-type-string** of **nordugrid**, the single parameter is the name of the NorduGrid resource to be used. See section 5.3.3 for details.

For a **grid-type-string** of **pbs**, no additional parameters are used. See section 5.3.5 for details.

For a **grid-type-string** of **unicore**, the first parameter is the name of the Unicare Usite to be used. The second parameter is the name of the Unicare Vsite to be used. See section 5.3.4 for details.

keystore_alias = <**name**> A string to locate the certificate in a Java keystore file, as used for a **unicore** job.

keystore_file = <**pathname**> The complete path and file name of the Java keystore file containing the certificate to be used for a **unicore** job.

keystore_passphrase_file = <**pathname**> The complete path and file name to the file containing the passphrase protecting a Java keystore file containing the certificate. Relevant for a **unicore** job.

MyProxyCredentialName = <**symbolic name**> The symbolic name that identifies a credential to the *MyProxy* server. This symbolic name is set as the credential is initially stored on the server (using *myproxy-init*).

MyProxyHost = **<host>:<port>** The Internet address of the host that is the *MyProxy* server. The **host** may be specified by either a host name (as in `head.example.com`) or an IP address (of the form `123.456.7.8`). The **port** number is an integer.

MyProxyNewProxyLifetime = **<number-of-minutes>** The new lifetime (in minutes) of the proxy after it is refreshed.

MyProxyPassword = **<password>** The password needed to refresh a credential on the *MyProxy* server. This password is set when the user initially stores credentials on the server (using *myproxy-init*). As an alternative to using **MyProxyPassword** in the submit description file, the password may be specified as a command line argument to *condor_submit* with the *-password* argument.

MyProxyRefreshThreshold = **<number-of-seconds>** The time (in seconds) before the expiration of a proxy that the proxy should be refreshed. For example, if **MyProxyRefreshThreshold** is set to the value 600, the proxy will be refreshed 10 minutes before it expires.

MyProxyServerDN = **<credential subject>** A string that specifies the expected Distinguished Name (credential subject, abbreviated DN) of the *MyProxy* server. It must be specified when the *MyProxy* server DN does not follow the conventional naming scheme of a host credential. This occurs, for example, when the *MyProxy* server DN begins with a user credential.

nordugrid_rsl = **<RSL-string>** Used to provide any additional RSL string attributes which are not covered by regular submit description file parameters. Used when the **universe** is **grid**, and the type of grid system is **nordugrid**.

transfer_error = **<True | False>** For jobs submitted to the grid universe only. If **True**, then the error output (from `stderr`) from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is given by the **error** command. If **False**, no transfer takes place (from the remote machine to submit machine), and the name of the file is given by the **error** command. The default value is **True**.

transfer_input = **<True | False>** For jobs submitted to the grid universe only. If **True**, then the job input (`stdin`) is transferred from the machine where the job was submitted to the remote machine. The name of the file that is transferred is given by the **input** command. If **False**, then the job's input is taken from a pre-staged file on the remote machine, and the name of the file is given by the **input** command. The default value is **True**.

For transferring files other than `stdin`, see **transfer_input_files**.

transfer_output = **<True | False>** For jobs submitted to the grid universe only. If **True**, then the output (from `stdout`) from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is given by the **output** command. If **False**, no transfer takes place (from the remote machine to submit machine), and the name of the file is given by the **output** command. The default value is **True**.

For transferring files other than `stdout`, see **transfer_output_files**.

x509userproxy = **<full-pathname>** Used to override the default path name for X.509 user certificates. The default location for X.509 proxies is the `/tmp` directory, which is generally a local file system. Setting this value would allow Condor to access the proxy in a shared file

system (for example, AFS). Condor will use the proxy specified in the submit description file first. If nothing is specified in the submit description file, it will use the environment variable `X509_USER_CERT`. If that variable is not present, it will search in the default location.

x509userproxy is relevant when the **universe** is **grid**, and the type of grid system is one of **gt2**, **gt4**, or **nordugrid**.

COMMANDS FOR PARALLEL, JAVA, and SCHEDULER UNIVERSES

hold_kill_sig = **<signal-number>** For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is put on hold with *condor_hold*. **signal-number** may be either the platform-specific name or value of the signal. If this command is not present, the value of **kill_sig** is used.

jar_files = **<file.list>** Specifies a list of additional JAR files to include when using the Java universe. JAR files will be transferred along with the executable and automatically added to the classpath.

java_vm_args = **<argument.list>** Specifies a list of additional arguments to the Java VM itself. When Condor runs the Java program, these are the arguments that go before the class name. This can be used to set VM-specific arguments like stack size, garbage-collector arguments and initial property values.

machine_count = **<max>** For the parallel (and therefore, the mpi) universe, a single value (*max*) is required. It is neither a maximum or minimum, but the number of machines to be dedicated toward running the job.

remove_kill_sig = **<signal-number>** For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is removed with *condor_rm*. **signal-number** may be either the platform-specific name or value of the signal. This example shows it both ways for a Linux signal:

```
remove_kill_sig = SIGUSR1
remove_kill_sig = 10
```

If this command is not present, the value of **kill_sig** is used.

COMMANDS FOR THE VM UNIVERSE

vm_cdrom_files = **file1, file2, ...** A comma-separated list of input CD-ROM files.

vm_checkpoint = **<True | False>** A boolean value specifying whether or not to take checkpoints. If not specified, the default value is `False`. In the current implementation, setting both **vm_checkpoint** and **vm_networking** to `True` does not yet work in all cases. Networking cannot be used if a vm universe job uses a checkpoint in order to continue execution after migration to another machine.

vm_macaddr = <MACaddr> Defines that MAC address that the virtual machine's network interface should have, in the standard format of six groups of two hexadecimal digits separated by colons.

vm_memory = <MBytes-of-memory> The amount of memory in MBytes that a vm universe job requires.

vm_networking = <True | False> Specifies whether to use networking or not. In the current implementation, setting both **vm_checkpoint** and **vm_networking** to True does not yet work in all cases. Networking cannot be used if a vm universe job uses a checkpoint in order to continue execution after migration to another machine.

vm_networking_type = <nat | bridge> When **vm_networking** is True, this definition augments the job's requirements to match only machines with the specified networking. If not specified, then either networking type matches.

vm_no_output_vm = <True | False> When True, prevents Condor from transferring output files back to the machine from which the vm universe job was submitted. If not specified, the default value is False.

vm_should_transfer_cdrom_files = <True | False> Specifies whether Condor will transfer CD-ROM files to the execute machine (True) or rely on access through a shared file system (False).

vm_type = <vmware | xen> Specifies the underlying virtual machine software that this job expects.

vmware_dir = <pathname> The complete path and name of the directory where VMware-specific files and applications such as the VMDK (Virtual Machine Disk Format) and VMX (Virtual Machine Configuration) reside.

vmware_should_transfer_files = <True | False> Specifies whether Condor will transfer VMware-specific files located as specified by **vmware_dir** to the execute machine (True) or rely on access through a shared file system (False). Omission of this required command (for VMware vm universe jobs) results in an error message from *condor_submit*, and the job will not be submitted.

vmware_snapshot_disk = <True | False> When True, causes Condor to utilize a VMware snapshot disk for new or modified files. If not specified, the default value is True.

xen_cdrom_device = <device> Describes the Xen CD-ROM device when **vm_cdrom_files** is defined.

xen_disk = file1:device1:permission1, file2:device2:permission2, ... A list of comma separated disk files. Each disk file is specified by 3 colon separated fields. The first field is the path and file name of the disk file. The second field specifies the device, and the third field specifies permissions.

An example that specifies two disk files:

```
xen_disk = /myxen/diskfile.img:sda1:w,/myxen/swap.img:sda2:w
```

xen_initrd = **<image-file>** When **xen_kernel** gives a path and file name for the kernel image to use, this optional command may specify a path to and ramdisk (**initrd**) image file.

xen_kernel = **<included | any | path-to-kernel>** A value of **included** specifies that the kernel is included in the disk file. A value of **any** specifies that the kernel is deployed on the execute machine, and its location is given by configuration. If not one of these values, then the value is a path and file name of the kernel to be used.

xen_kernel_params = **<string>** A string that is appended to the Xen kernel command line.

xen_root = **<string>** A string that is appended to the Xen kernel command line to specify the root device. This string is required when **xen_kernel** is **any** or gives a path to a kernel. Omission for this required case results in an error message from *condor_submit*, and the job will not be submitted.

xen_transfer_files = **<list-of-files>** A comma separated list of all files that Condor is to transfer to the execute machine.

ADVANCED COMMANDS

concurrency_limits = **<string-list>** A list of resources that this job needs. The resources are presumed to have concurrency limits placed upon them, thereby limiting the number of concurrent jobs in execution which need the named resource. Commas and space characters delimit the items in the list. Each item in the list may specify a numerical value identifying the integer number of resources required for the job. The syntax follows the resource name by a colon character (:) and the numerical value. See section 3.12.11 for details on concurrency limits.

copy_to_spool = **<True | False>** If **copy_to_spool** is **True**, then *condor_submit* copies the executable to the local spool directory before running it on a remote host. As copying can be quite time consuming and unnecessary, the default value is **False** for all job universes other than the standard universe. When **False**, *condor_submit* does not copy the executable to a local spool directory. The default is **True** in standard universe, because resuming execution from a checkpoint can only be guaranteed to work using precisely the same executable that created the checkpoint.

coresize = **<size>** Should the user's program abort and produce a core file, **coresize** specifies the maximum size in bytes of the core file which the user wishes to keep. If **coresize** is not specified in the command file, the system's user resource limit "coredumpsize" is used. This limit is not used in HP-UX and DUX operating systems.

cron_day_of_month = **<Cron-evaluated Day>** The set of days of the month for which a deferral time applies. See section 2.12.2 for further details and examples.

cron_day_of_week = **<Cron-evaluated Day>** The set of days of the week for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_hour = **<Cron-evaluated Hour>** The set of hours of the day for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_minute = **<Cron-evaluated Minute>** The set of minutes within an hour for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_month = **<Cron-evaluated Month>** The set of months within a year for which a deferral time applies. See section 2.12.2 for details, semantics, and examples.

cron_prep_time = **<ClassAd Integer Expression>** Analogous to **deferral_prep_time**. The number of seconds prior to a job's deferral time that the job may be matched and sent to an execution machine.

cron_window = **<ClassAd Integer Expression>** Analogous to the submit command **deferral_window**. It allows cron jobs that miss their deferral time to begin execution.

See section 2.12.1 for further details and examples.

deferral_prep_time = **<ClassAd Integer Expression>** The number of seconds prior to a job's deferral time that the job may be matched and sent to an execution machine.

See section 2.12.1 for further details.

deferral_time = **<ClassAd Integer Expression>** Allows a job to specify the time at which its execution is to begin, instead of beginning execution as soon as it arrives at the execution machine. The deferral time is an expression that evaluates to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). Deferral time is evaluated with respect to the execution machine. This option delays the start of execution, but not the matching and claiming of a machine for the job. If the job is not available and ready to begin execution at the deferral time, it has missed its deferral time. A job that misses its deferral time will be put on hold in the queue.

See section 2.12.1 for further details and examples.

Due to implementation details, a deferral time may not be used for scheduler universe jobs.

deferral_window = **<ClassAd Integer Expression>** The deferral window is used in conjunction with the **deferral_time** command to allow jobs that miss their deferral time to begin execution.

See section 2.12.1 for further details and examples.

email_attributes = **<list-of-job-ad-attributes>** A comma-separated list of attributes from the job ClassAd. These attributes and their values will be included in the e-mail notification of job completion.

image_size = **<size>** This command tells Condor the maximum virtual image size to which you believe your program will grow during its execution. Condor will then execute your job only on machines which have enough resources, (such as virtual memory), to support executing your job. If you do not specify the image size of your job in the description file, Condor will automatically make a (reasonably accurate) estimate about its size and adjust this estimate as your program runs. If the image size of your job is underestimated, it may crash due to inability to acquire more address space, e.g. malloc() fails. If the image size is overestimated, Condor may have difficulty finding machines which have the required resources. *size* must be in Kbytes, e.g. for an image size of 8 megabytes, use a *size* of 8000.

initialdir = <**directory-path**> Used to give jobs a directory with respect to file input and output. Also provides a directory (on the machine from which the job is submitted) for the user log, when a full path is not specified.

For vanilla or MPI universe jobs where there is a shared file system, it is the current working directory on the machine where the job is executed.

For vanilla, grid, or MPI universe jobs where file transfer mechanisms are utilized (there is *not* a shared file system), it is the directory on the machine from which the job is submitted where the input files come from, and where the job's output files go to.

For standard universe jobs, it is the directory on the machine from which the job is submitted where the *condor_shadow* daemon runs; the current working directory for file input and output accomplished through remote system calls.

For scheduler universe jobs, it is the directory on the machine from which the job is submitted where the job runs; the current working directory for file input and output with respect to relative path names.

Note that the path to the executable is *not* relative to **initialdir**; if it is a relative path, it is relative to the directory in which the *condor_submit* command is run.

job_lease_duration = <**number-of-seconds**> For vanilla and java universe jobs only, the duration (in seconds) of a job lease. The default value is twenty minutes for universes that support it. If a job lease is not desired, the value can be explicitly set to 0 to disable the job lease semantics. See section 2.15.4 for details of job leases.

kill_sig = <**signal-number**> When Condor needs to kick a job off of a machine, it will send the job the signal specified by **signal-number**. **signal-number** needs to be an integer which represents a valid signal on the execution machine. For jobs submitted to the standard universe, the default value is the number for SIGTSTP which tells the Condor libraries to initiate a checkpoint of the process. For jobs submitted to the vanilla universe, the default is SIGTERM which is the standard way to terminate a program in Unix.

load_profile = <**True** | **False**> When **True**, loads the account profile of the dedicated run account for Windows jobs. May not be used with **run_as_owner**.

match_list_length = <**integer value**> Defaults to the value zero (0). When **match_list_length** is defined with an integer value greater than zero (0), attributes are inserted into the job ClassAd. The maximum number of attributes defined is given by the integer value. The job ClassAds introduced are given as

```
LastMatchName0 = "most-recent-Name"
LastMatchName1 = "next-most-recent-Name"
```

The value for each introduced ClassAd is given by the value of the Name attribute from the machine ClassAd of a previous execution (match). As a job is matched, the definitions for these attributes will roll, with LastMatchName1 becoming LastMatchName2, LastMatchName0 becoming LastMatchName1, and LastMatchName0 being set by the most recent value of the Name attribute.

An intended use of these job attributes is in the requirements expression. The requirements can allow a job to prefer a match with either the same or a different resource than a previous match.

max_job_retirement_time = **<integer expression>** An integer-valued expression (in seconds) that does nothing unless the machine that runs the job has been configured to provide retirement time (see section 3.5.8). Retirement time is a grace period given to a job to finish naturally when a resource claim is about to be preempted. No kill signals are sent during a retirement time. The default behavior in many cases is to take as much retirement time as the machine offers, so this command will rarely appear in a submit description file.

When a resource claim is to be preempted, this expression in the submit file specifies the maximum run time of the job (in seconds, since the job started). This expression has no effect, if it is greater than the maximum retirement time provided by the machine policy. If the resource claim is *not* preempted, this expression and the machine retirement policy are irrelevant. If the resource claim *is* preempted and the job finishes sooner than the maximum time, the claim closes gracefully and all is well. If the resource claim is preempted and the job does *not* finish in time, the usual preemption procedure is followed (typically a soft kill signal, followed by some time to gracefully shut down, followed by a hard kill signal).

Standard universe jobs and any jobs running with **nice_user** priority have a default **max_job_retirement_time** of 0, so no retirement time is utilized by default. In all other cases, no default value is provided, so the maximum amount of retirement time is utilized by default.

Setting this expression does not affect the job's resource requirements or preferences. For a job to only run on a machine with a minimum , or to preferentially run on such machines, explicitly specify this in the requirements and/or rank expressions.

nice_user = **<True | False>** Normally, when a machine becomes available to Condor, Condor decides which job to run based upon user and job priorities. Setting **nice_user** equal to **True** tells Condor not to use your regular user priority, but that this job should have last priority among all users and all jobs. So jobs submitted in this fashion run only on machines which no other non-nice_user job wants — a true “bottom-feeder” job! This is very handy if a user has some jobs they wish to run, but do not wish to use resources that could instead be used to run other people's Condor jobs. Jobs submitted in this fashion have “nice-user.” pre-appended in front of the owner name when viewed from *condor_q* or *condor_userprio*. The default value is **False**.

noop_job = **<ClassAd Boolean Expression>** When this boolean expression is **True**, the job is immediately removed from the queue, and Condor makes no attempt at running the job. The log file for the job will show a job submitted event and a job terminated event, along with an exit code of 0, unless the user specifies a different signal or exit code.

noop_job_exit_code = **<return value>** When **noop_job** is in the submit description file and evaluates to **True**, this command allows the job to specify the return value as shown in the job's log file job terminated event. If not specified, the job will show as having terminated with status 0. This overrides any value specified with **noop_job_exit_signal**.

noop_job_exit_signal = **<signal number>** When **noop_job** is in the submit description file and evaluates to **True**, this command allows the job to specify the signal number that the job's log event will show the job having terminated with.

remote_initialdir = <directory-path> The path specifies the directory in which the job is to be executed on the remote machine. This is currently supported in all universes except for the standard universe.

rendezvousdir = <directory-path> Used to specify the shared file system directory to be used for file system authentication when submitting to a remote scheduler. Should be a path to a preexisting directory.

request_cpus = <num-cpus> For pools that enable dynamic *condor_startd* provisioning (see section 3.12.7), the number of CPUs requested for this job.

request_disk = <quantity> For pools that enable dynamic *condor_startd* provisioning (see section 3.12.7), the amount of disk space requested for this job.

request_memory = <quantity> For pools that enable dynamic *condor_startd* provisioning (see section 3.12.7), the amount of memory space requested for this job.

+<attribute> = <value> A line which begins with a '+' (plus) character instructs *condor_submit* to insert the following *attribute* into the job ClassAd with the given *value*.

In addition to commands, the submit description file can contain macros and comments:

Macros Parameterless macros in the form of `$(macro_name)` may be inserted anywhere in Condor submit description files. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Three pre-defined macros are supplied by the submit description file parser. The third of the pre-defined macros is only relevant to MPI universe jobs. The `$(Cluster)` macro supplies the value of the `ClusterId` job ClassAd attribute, and the `$(Process)` macro supplies the value of the `ProcId` job ClassAd attribute. These macros are intended to aid in the specification of input/output files, arguments, etc., for clusters with lots of jobs, and/or could be used to supply a Condor process with its own cluster and process numbers on the command line. The `$(Node)` macro is defined only for MPI universe jobs. It is a unique value assigned for the duration of the job that essentially identifies the machine on which a program is executing.

To use the dollar sign character (\$) as a literal, without macro expansion, use

```
$(DOLLAR)
```

In addition to the normal macro, there is also a special kind of macro called a *substitution macro* that allows the substitution of a ClassAd attribute value defined on the resource machine itself (gotten after a match to the machine has been made) into specific commands within the submit description file. The substitution macro is of the form:

```
$$(<attribute>)
```

A common use of this macro is for the heterogeneous submission of an executable:

```
executable = povray.$$$(opsys)$$$(arch)
```

Values for the `opsys` and `arch` attributes are substituted at match time for any given resource. This allows Condor to automatically choose the correct executable for the matched machine.

An extension to the syntax of the substitution macro provides an alternative string to use if the machine attribute within the substitution macro is undefined. The syntax appears as:

```
$$$ (attribute:string_if_attribute_undefined)
```

An example using this extended syntax provides a path name to a required input file. Since the file can be placed in different locations on different machines, the file's path name is given as an argument to the program.

```
argument = $$$ (input_file_path:/usr/foo)
```

On the machine, if the attribute `input_file_path` is not defined, then the path `/usr/foo` is used instead.

A further extension to the syntax of the substitution macro allows the evaluation of a ClassAd expression to define the value. As all substitution macros, the expression is evaluated after a match has been made. Therefore, the expression may refer to machine attributes by prefixing them with the scope resolution prefix `TARGET.`, as specified in section 4.1.2. To place a ClassAd expression into the substitution macro, square brackets are added to delimit the expression. The syntax appears as:

```
$$$ ([ClassAd expression])
```

An example of a job that uses this syntax may be one that wants to know how much memory it can use. The application cannot detect this itself, as it would potentially use all of the memory on a multi-slot machine. So the job determines the memory per slot, reducing it by 10% to account for miscellaneous overhead, and passes this as a command line argument to the application. In the submit description file will be

```
arguments=--memory $$$ ([TARGET.Memory * 0.9])
```

To insert two dollar sign characters (`$$`) as literals into a ClassAd string, use

```
$$$ (DOLLARDOLLAR)
```

The environment macro, `$ENV`, allows the evaluation of an environment variable to be used in setting a submit description file command. The syntax used is

```
$ENV(variable)
```

An example submit description file command that uses this functionality evaluates the submitter's home directory in order to set the path and file name of a log file:

```
log = $ENV(HOME)/jobs/logfile
```

The environment variable is evaluated when the submit description file is processed.

The \$RANDOM_CHOICE macro allows a random choice to be made from a given list of parameters at submission time. For an expression, if some randomness needs to be generated, the macro may appear as

```
$RANDOM_CHOICE( 0 , 1 , 2 , 3 , 4 , 5 , 6 )
```

When evaluated, one of the parameters values will be chosen.

Comments Blank lines and lines beginning with a pound sign ('#') character are ignored by the submit description file parser.

Options

-verbose Verbose output - display the created job ClassAd

-unused As a default, causes no warnings to be issued about user-defined macros not being used within the submit description file. The meaning reverses (toggles) when the configuration variable `WARN_ON_UNUSED_SUBMIT_FILE_MACROS` is set to the nondefault value of `False`. Printing the warnings can help identify spelling errors of submit description file commands. The warnings are sent to stderr.

-name *schedd_name* Submit to the specified *condor_schedd*. Use this option to submit to a *condor_schedd* other than the default local one. *schedd_name* is the value of the Name ClassAd attribute on the machine where the *condor_schedd* daemon runs.

-remote *schedd_name* Submit to the specified *condor_schedd*, spooling all required input files over the network connection. *schedd_name* is the value of the Name ClassAd attribute on the machine where the *condor_schedd* daemon runs. This option is equivalent to using both **-name** and **-spool**.

-pool *pool_name* Look in the specified pool for the *condor_schedd* to submit to. This option is used with **-name** or **-remote**.

-disable Disable file permission checks.

-password *passphrase* Specify a password to the *MyProxy* server.

-debug Cause debugging information to be sent to `stderr`, based on the value of the configuration variable `SUBMIT_DEBUG`.

-append *command* Augment the commands in the submit description file with the given command. This command will be considered to immediately precede the `Queue` command within the submit description file, and come after all other previous commands. The submit description file is not modified. Multiple commands are specified by using the **-append** option multiple times. Each new command is given in a separate **-append** option. Commands with spaces in them will need to be enclosed in double quote marks.

-spool Spool all required input files, user log, and proxy over the connection to the *condor_schedd*. After submission, modify local copies of the files without affecting your jobs. Any output files for completed jobs need to be retrieved with *condor_transfer_data*.

-dump *filename* Sends all ClassAds to the specified file, instead of to the *condor_schedd*.

submit description file The pathname to the submit description file. If this optional argument is missing or equal to “-”, then the commands are taken from standard input.

Exit Status

condor_submit will exit with a status value of 0 (zero) upon success, and a non-zero value upon failure.

Examples

- **Submit Description File Example 1:** This example queues three jobs for execution by Condor. The first will be given command line arguments of *15* and *2000*, and it will write its standard output to *foo.out1*. The second will be given command line arguments of *30* and *2000*, and it will write its standard output to *foo.out2*. Similarly the third will have arguments of *45* and *6000*, and it will use *foo.out3* for its standard output. Standard error output (if any) from all three programs will appear in *foo.error*.

```
#####
#
# submit description file
# Example 1: queuing multiple jobs with differing
# command line arguments and output files.
#
```

```
#####

Executable      = foo
Universe         = standard

Arguments        = 15 2000
Output           = foo.out1
Error            = foo.err1
Queue

Arguments        = 30 2000
Output           = foo.out2
Error            = foo.err2
Queue

Arguments        = 45 6000
Output           = foo.out3
Error            = foo.err3
Queue
```

- **Submit Description File Example 2:** This submit description file example queues 150 runs of program *foo* which must have been compiled and linked for Sun workstations running Solaris 8. Condor will not attempt to run the processes on machines which have less than 32 Megabytes of physical memory, and it will run them on machines which have at least 64 Megabytes, if such machines are available. Stdin, stdout, and stderr will refer to *in.0*, *out.0*, and *err.0* for the first run of this program (process 0). Stdin, stdout, and stderr will refer to *in.1*, *out.1*, and *err.1* for process 1, and so forth. A log file containing entries about where and when Condor runs, takes checkpoints, and migrates processes in this cluster will be written into file *foo.log*.

```
#####
#
# Example 2: Show off some fancy features including
# use of pre-defined macros and logging.
#
#####

Executable      = foo
Universe         = standard
Requirements     = Memory >= 32 && OpSys == "SOLARIS28" && Arch == "SUN4u"
Rank             = Memory >= 64
Image_Size       = 28 Meg

Error            = err.$(Process)
Input            = in.$(Process)
Output           = out.$(Process)
Log              = foo.log

Queue 150
```

- **Command Line example:** The following command uses the **-append** option to add two commands before the job(s) is queued. A log file and an error log file are specified. The submit description file is unchanged.

```
condor_submit -a "log = out.log" -a "error = error.log" mysubmitfile
```

Note that each of the added commands is contained within quote marks because there are space characters within the command.

- `periodic_remove` example: A job should be removed from the queue, if the total suspension time of the job is more than half of the run time of the job.

Including the command

```
periodic_remove = CumulativeSuspensionTime >
                  ((RemoteWallClockTime - CumulativeSuspensionTime) / 2.0)
```

in the submit description file causes this to happen.

General Remarks

- For security reasons, Condor will refuse to run any jobs submitted by user root (UID = 0) or by a user whose default group is group wheel (GID = 0). Jobs submitted by user root or a user with a default group of wheel will appear to sit forever in the queue in an idle state.
- All path names specified in the submit description file must be less than 256 characters in length, and command line arguments must be less than 4096 characters in length; otherwise, *condor_submit* gives a warning message but the jobs will not execute properly.
- Somewhat understandably, behavior gets bizarre if the user makes the mistake of requesting multiple Condor jobs to write to the same file, and/or if the user alters any files that need to be accessed by a Condor job which is still in the queue. For example, the compressing of data or output files before a Condor job has completed is a common mistake.
- To disable checkpointing for Standard Universe jobs, include the line:

```
+WantCheckpoint = False
```

in the submit description file before the queue command(s).

See Also

Condor User Manual

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_submit_dag

Manage and queue jobs within a specified DAG for execution on remote machines

Synopsis

condor_submit_dag [-help — -version]

condor_submit_dag [-no_submit] [-verbose] [-force] [-maxidle *NumberOfJobs*]
 [-maxjobs *NumberOfJobs*] [-dagman *DagmanExecutable*] [-maxpre *NumberOfPREscripts*]
 [-maxpost *NumberOfPOSTscripts*] [-notification *value*] [-noeventchecks] [-allowlogerror]
 [-r *schedd_name*] [-debug *level*] [-usedagdir] [-outfile_dir *directory*] [-config *ConfigFileName*]
 [-insert_sub_file *FileName*] [-append *Command*] [-oldrescue *0—1*] [-autorescue *0—1*]
 [-dorescuefrom *number*] [-allowversionmismatch] [-no_recurse] [-update_submit]
 [-DumpRescue] *DAGInputFile1* [*DAGInputFile2* .. *DAGInputFileN*]

Description

condor_submit_dag is the program for submitting a DAG (directed acyclic graph) of jobs for execution under Condor. The program enforces the job dependencies defined in one or more *DAGInputFiles*. Each *DAGInputFile* contains commands to direct the submission of jobs implied by the nodes of a DAG to Condor. See the Condor User Manual, section 2.10 for a complete description.

Options

-help Display usage information.

-version Display version information.

-no_submit Produce the Condor submit description file for DAGMan, but do not submit DAGMan as a Condor job.

-verbose Cause *condor_submit_dag* to give verbose error messages.

-force Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist. Note that dagman.out will be appended to, not overwritten. If "new-style" rescue DAG mode (see section ??) is in effect, and any "new-style" rescue DAGs exist, the **-force** flag will cause them to be renamed, and the original DAG will be run. If "old-style" rescue DAG mode (see section ??) is in effect, any existing "old-style" rescue DAGs will be deleted, and

the original DAG will be run.

- maxidle *NumberOfJobs*** Sets the maximum number of idle jobs allowed before *condor_dagman* stops submitting more jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. *NumberOfJobs* is a positive integer. If the option is omitted, the number of idle jobs is unlimited. Note that for this argument, each individual process within a cluster counts as a job, which is inconsistent with **-maxjobs**.
- maxjobs *NumberOfJobs*** Sets the maximum number of jobs within the DAG that will be submitted to Condor at one time. *NumberOfJobs* is a positive integer. If the option is omitted, the default number of jobs is unlimited. Note that for this argument, each cluster counts as one job, no matter how many individual processes are in the cluster.
- dagman *DagmanExecutable*** Allows the specification of an alternate *condor_dagman* executable to be used instead of the one found in the user's path. This must be a fully qualified path.
- maxpre *NumberOfPREscripts*** Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. If this option is omitted, the default number of PRE scripts is unlimited.
- maxpost *NumberOfPOSTscripts*** Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. If this option is omitted, the default number of POST scripts is unlimited.
- notification *value*** Sets the e-mail notification for DAGMan itself. This information will be used within the Condor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See **notification** within the section of submit description file commands in the *condor_submit* manual page on page 795 for specification of *value*.
- noeventchecks** This argument is no longer used; it is now ignored. Its functionality is now implemented by the DAGMAN_ALLOW_EVENTS configuration macro (see section 3.3.25).
- allowlogerror** This optional argument has *condor_dagman* try to run the specified DAG, even in the case of detected errors in the user log specification.
- r *schedd_name*** Submit to a remote schedd. The jobs will be submitted to the schedd on the specified remote host. On Unix systems, the Condor administrator for your site must override the default AUTHENTICATION_METHODS configuration setting to enable remote file system (FS_REMOTE) authentication.

- debug *level*** Passes the the *level* of debugging output desired to *condor_dagman*. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. A default value of 3 is passed to *condor_dagman* when not specified with this option. See the *condor_dagman* manual page on page 711 for detailed descriptions of these values.
- usedagdir** This optional argument causes *condor_dagman* to run each specified DAG as if *condor_submit_dag* had been run in the directory containing that DAG file. This option is most useful when running multiple DAGs in a single *condor_dagman*. Note that the **-usedagdir** flag must not be used when running an "old-style" rescue DAG (see section ??).
- outfile_dir *directory*** Specifies the directory in which the *.dagman.out* file will be written. The *directory* may be specified relative to the current working directory as *condor_submit_dag* is executed, or specified with an absolute path. Without this option, the *.dagman.out* file is placed in the same directory as the first DAG input file listed on the command line.
- config *ConfigFileName*** Specifies a configuration file to be used for this DAGMan run. Note that the options specified in the configuration file apply to all DAGs if multiple DAGs are specified. Further note that it is a fatal error if the configuration file specified by this option conflicts with a configuration file specified in any of the DAG files, if they specify one. For more information about how *condor_dagman* configuration files work, see section ??.
- insert_sub_file *FileName*** Specifies a file to insert into the *.condor.sub* file created by *condor_submit_dag*. The specified file must contain only legal submit file commands. Only one file can be inserted. (If both the DAGMAN_INSERT_SUB_FILE configuration variable and **-insert_sub_file** are specified, **-insert_sub_file** overrides DAGMAN_INSERT_SUB_FILE.) The specified file is inserted into the *.condor.sub* file before the Queue command and before any commands specified with the **-append** option.
- append *Command*** Specifies a command to append to the *.condor.sub* file created by *condor_submit_dag*. The specified command is appended to the *.condor.sub* file immediately before the Queue command. Multiple commands are specified by using the **-append** option multiple times. Each new command is given in a separate **-append** option. Commands with spaces in them must be enclosed in double quotes. Commands specified with the **-append** option are appended to the *.condor.sub* file *after* commands inserted from a file specified by the **-insert_sub_file** option or the DAGMAN_INSERT_SUB_FILE configuration variable, so the **-append** command(s) will override commands from the inserted file if the commands conflict.
- oldrescue *0—1*** Whether to use "old-style" rescue DAG naming (see section ??) when creating a rescue DAG (0 = false, 1 = true).

- autorescue 0—1** Whether to automatically run the newest rescue DAG for the given DAG file, if one exists (0 = false, 1 = true).
- dorescuefrom *number*** Forces *condor_dagman* to run the specified rescue DAG number for the given DAG. A value of 0 is the same as not specifying this option. Specifying a non-existent rescue DAG is a fatal error.
- allowversionmismatch** This optional argument causes *condor_dagman* to allow a version mismatch between *condor_dagman* itself and the *.condor.sub* file produced by *condor_submit_dag* (or, in other words, between *condor_submit_dag* and *condor_dagman*). WARNING! This option should be used only if absolutely necessary. Allowing version mismatches can cause subtle problems when running DAGs.
- no_recurse** This optional argument causes *condor_submit_dag* to *not* run itself recursively on nested DAGs (the default is that recursion is enabled). (DAG nodes specified with the SUBDAG EXTERNAL keyword or with submit file names ending in *.condor.sub* are considered nested DAGs, and, by default, *condor_submit_dag -no_submit -update_submit* is run on the corresponding DAG files.)
- update_submit** This optional argument causes an existing *.condor.sub* file to not be treated as an error; rather, the *.condor.sub* file will be overwritten, but the existing values of **-maxjobs**, **-maxidle**, **-maxpre**, and **-maxpost** will be preserved.
- DumpRescue** This optional argument tells *condor_dagman* to immediately dump a rescue DAG and then exit, as opposed to actually running the DAG. (This feature is mainly intended for testing.)

See Also

Condor User Manual

Exit Status

condor_submit_dag will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To run a single DAG:

```
% condor_submit_dag diamond.dag
```

To run a DAG when it has already been run and the output files exist:

```
% condor_submit_dag -force diamond.dag
```

To run a DAG, limiting the number of idle node jobs in the DAG to a maximum of five:

```
% condor_submit_dag -maxidle 5 diamond.dag
```

To run a DAG, limiting the number of concurrent PRE scripts to 10 and the number of concurrent POST scripts to five:

```
% condor_submit_dag -maxpre 10 -maxpost 5 diamond.dag
```

To run two DAGs, each of which is set up to run in its own directory:

```
% condor_submit_dag -usedagdir dag1/diamond1.dag dag2/diamond2.dag
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_transfer_data

transfer spooled data

Synopsis

condor_transfer_data [-help | -version]

condor_transfer_data [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
[-addr "<a.b.c.d:port>"] *cluster...* | *cluster.process...* | *user...* | -constraint *expression* ...

condor_transfer_data [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
[-addr "<a.b.c.d:port>"] -all

Description

condor_transfer_data causes Condor to transfer spooled data. It is meant to be used in conjunction with the **-spool** option of *condor_submit*, as in

```
condor_submit -spool mysubmitfile
```

Submission of a job with the **-spool** option causes Condor to spool all input files, the user log, and any proxy across a connection to the machine where the *condor_schedd* daemon is running. After spooling these files, the machine from which the job is submitted may disconnect from the network or modify its local copies of the spooled files.

When the job finishes, the job has `JobStatus = 4`, meaning that the job has completed. The output of the job is spooled, and *condor_transfer_data* retrieves the output of the completed job.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr "<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

cluster Transfer spooled data belonging to the specified cluster

cluster.process Transfer spooled data belonging to a specific job in the cluster

user Transfer spooled data belonging to the specified user

-constraint *expression* Transfer spooled data for jobs which match the job ClassAd expression *constraint*

-all Transfer all spooled data

Exit Status

condor_transfer_data will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_updates_stats

Display output from *condor_status*

Synopsis

condor_updates_stats [**--help** | **-h**] [**--version**]

condor_updates_stats [**--long** | **-l**] [**--history=<min>-<max>**] [**--interval=<seconds>**]
[**--notime**] [**--time**] [**--summary** | **-s**]

Description

condor_updates_stats parses the output from *condor_status*, and it displays the information relating to update statistics in a useful format. The statistics are displayed with the most recent update first; the most recent update is numbered with the smallest value.

The number of historic points that represent updates is configurable on a per-source basis. See `COLLECTOR_DAEMON_HISTORY_SIZE` in section 3.3.16.

Options

—help Display usage information and exit.

-h Same as **—help**.

—version Display Condor version information and exit.

—long All update statistics are displayed. Without this option, the statistics are condensed.

-l Same as **—long**.

—history=<min>-<max> Sets the range of update numbers that are printed. By default, the entire history is displayed. To limit the range, the minimum and/or maximum number may be specified. If a minimum is not specified, values from 0 to the maximum are displayed. If the maximum is not specified, all values after the minimum are displayed. When both minimum and maximum are specified, the range to be displayed includes the endpoints as well as all values in between. If no = sign is given, command-line parsing fails, and usage information is displayed. If an = sign is given, with no minimum or maximum values, the default of the

entire history is displayed.

—interval=<seconds> The assumed update interval, in seconds. Assumed times for the updates are displayed, making the use of the **—time** option together with the **—interval** option redundant.

—notime Do not display assumed times for the updates. If more than one of the options **—notime** and **—time** are provided, the final one within the command line parsed determines the display.

—time Display assumed times for the updates. If more than one of the options **—notime** and **—time** are provided, the final one within the command line parsed determines the display.

—summary Display only summary information, not the entire history for each machine.

-s Same as **—summary**.

Exit Status

condor_updates_stats will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

Examples

Assuming the default of 128 updates kept, and assuming that the update interval is 5 minutes, *condor_updates_stats* displays:

```
$ condor_status -l host1 | condor_updates_stats --interval=300
(Reading from stdin)
*** Name/Machine = 'HOST1.cs.wisc.edu' MyType = 'Machine' ***
Type: Main
Stats: Total=2277, Seq=2276, Lost=3 (0.13%)
  0 @ Mon Feb 16 12:55:38 2004: Ok
...
 28 @ Mon Feb 16 10:35:38 2004: Missed
 29 @ Mon Feb 16 10:30:38 2004: Ok
...
127 @ Mon Feb 16 02:20:38 2004: Ok
```

Within this display, update numbered 27, which occurs later in time than the missed update numbered 28, is Ok. Each change in state, in reverse time order, displays in this condensed version.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_userlog

Display and summarize job statistics from job log files.

Synopsis

condor_userlog [-help] [-total | -raw] [-debug] [-evict] [-j *cluster* | *cluster.proc*] [-all] [-hostname] *logfile* ...

Description

condor_userlog parses the information in job log files and displays summaries for each workstation allocation and for each job. See the manual page for *condor_submit* on page 795 for instructions for specifying that Condor write a log file for your jobs.

If **-total** is not specified, *condor_userlog* will first display a record for each workstation allocation, which includes the following information:

Job The cluster/process id of the Condor job.

Host The host where the job ran. By default, the host's IP address is displayed. If **-hostname** is specified, the host name will be displayed instead.

Start Time The time (month/day hour:minute) when the job began running on the host.

Evict Time The time (month/day hour:minute) when the job was evicted from the host.

Wall Time The time (days+hours:minutes) for which this workstation was allocated to the job.

Good Time The allocated time (days+hours:min) which contributed to the completion of this job. If the job exited during the allocation, then this value will equal "Wall Time." If the job performed a checkpoint, then the value equals the work saved in the checkpoint during this allocation. If the job did not exit or perform a checkpoint during this allocation, the value will be 0+00:00. This value can be greater than 0 and less than "Wall Time" if the application completed a periodic checkpoint during the allocation but failed to checkpoint when evicted.

CPU Usage The CPU time (days+hours:min) which contributed to the completion of this job.

condor_userlog will then display summary statistics per host:

Host/Job The IP address or host name for the host.

Wall Time The workstation time (days+hours:minutes) allocated by this host to the jobs specified in the query. By default, all jobs in the log are included in the query.

Good Time The time (days+hours:minutes) allocated on this host which contributed to the completion of the jobs specified in the query.

CPU Usage The CPU time (days+hours:minutes) obtained from this host which contributed to the completion of the jobs specified in the query.

Avg Alloc The average length of an allocation on this host (days+hours:minutes).

Avg Lost The average amount of work lost (days+hours:minutes) when a job was evicted from this host without successfully performing a checkpoint.

Goodput This percentage is computed as Good Time divided by Wall Time.

Util. This percentage is computed as CPU Usage divided by Good Time.

condor_userlog will then display summary statistics per job:

Host/Job The cluster/process id of the Condor job.

Wall Time The total workstation time (days+hours:minutes) allocated to this job.

Good Time The total time (days+hours:minutes) allocated to this job which contributed to the job's completion.

CPU Usage The total CPU time (days+hours:minutes) which contributed to this job's completion.

Avg Alloc The average length of a workstation allocation obtained by this job in minutes (days+hours:minutes).

Avg Lost The average amount of work lost (days+hours:minutes) when this job was evicted from a host without successfully performing a checkpoint.

Goodput This percentage is computed as Good Time divided by Wall Time.

Util. This percentage is computed as CPU Usage divided by Good Time.

Finally, *condor_userlog* will display a summary for all hosts and jobs.

Options

-help Get a brief description of the supported options

-total Only display job totals

-raw Display raw data only

- debug** Debug mode
- j** Select a specific cluster or cluster.proc
- evict** Select only allocations which ended due to eviction
- all** Select all clusters and all allocations
- hostname** Display host name instead of IP address

General Remarks

Since the Condor job log file format does not contain a year field in the timestamp, all entries are assumed to occur in the current year. Allocations which begin in one year and end in the next will be silently ignored.

Exit Status

condor_userlog will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_userprio

Manage user priorities

Synopsis

```
condor_userprio      [-pool centralmanagerhostname[:portnumber]]      [-all]      [-usage]
[-setprio username value]      [-setfactor username value]      [-setaccum username value]
[-setbegin username value]      [-setlast username value]      [-resetusage username]      [-resetall]
[-delete username] [-getreslist username] [-allusers] [-activefrom month day year] [-l]
```

Description

condor_userprio with no arguments, lists the active users (see below) along with their priorities, in increasing priority order. The -all option can be used to display more detailed information about each user, which includes the following columns:

Effective Priority The effective priority value of the user, which is used to calculate the user's share when allocating resources. A lower value means a higher priority, and the minimum value (highest priority) is 0.5. The effective priority is calculated by multiplying the real priority by the priority factor.

Real Priority The value of the real priority of the user. This value follows the user's resource usage.

Priority Factor The system administrator can set this value for each user, thus controlling a user's effective priority relative to other users. This can be used to create different classes of users.

Res Used The number of resources currently used (e.g. the number of running jobs for that user).

Accumulated Usage The accumulated number of resource-hours used by the user since the usage start time.

Usage Start Time The time since when usage has been recorded for the user. This time is set when a user job runs for the first time. It is reset to the present time when the usage for the user is reset (with the -resetusage or -resetall options).

Last Usage Time The most recent time a resource usage has been recorded for the user.

The -usage option displays the username, accumulated usage, usage start time and last usage time for each user, sorted on accumulated usage.

The -setprio, -setfactor options are used to change a user's real priority and priority factor. The -setaccum option sets a user's accumulated usage. The -setbegin, -setlast options are used to change a user's begin usage time and last usage time. The -setaccum option sets a user's accumulated usage.

The `-resetusage` and `-resetall` options are used to reset the accumulated usage for users. The usage start time is set to the current time when the accumulated usage is reset. These options require administrator privileges.

By default only users for whom usage was recorded in the last 24 hours or whose priority is greater than the minimum are listed. The `-activefrom` and `-allusers` options can be used to display users who had some usage since a specified date, or ever. The summary line for last usage time will show this date.

The `-getreslist` option is used to display the resources currently used by a user. The output includes the start time (the time the resource was allocated to the user), and the match time (how long has the resource been allocated to the user).

Note that when specifying user names on the command line, the name must include the UID domain (e.g. `user@uid-domain` - exactly the same way user names are listed by the `userprio` command).

The `-pool` option can be used to contact a different central-manager instead of the local one (the default).

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 282 for further explanation.

Options

-pool *centralmanagerhostname[:portnumber]* Contact specified *centralmanagerhostname* with an optional port number instead of the local central manager. This can be used to check other pools. NOTE: The host name (and optionally port) specified refer to the host name (and port) of the *condor_negotiator* to query for user priorities. This is slightly different than most Condor tools that support `-pool`, which expect the host name (and optionally port) of the *condor_collector*, instead.

-all Display detailed information about each user.

-usage Display usage information for each user.

-setprio *username value* Set the real priority of the specified user to the specified value.

-setfactor *username value* Set the priority factor of the specified user to the specified value.

-setaccum *username value* Set the accumulated usage of the specified user to the specified floating point value.

- setbegin *username value*** Set the begin usage time of the specified user to the specified value.
- setlast *username value*** Set the last usage time of the specified user to the specified value.
- resetusage *username*** Reset the accumulated usage of the specified user to zero.
- resetall** Reset the accumulated usage of all the users to zero.
- delete *username*** Remove the specified *username* from Condor's accounting.
- getreslist *username*** Display all the resources currently allocated to the specified user.
- allusers** Display information for all the users who have some recorded accumulated usage.
- activefrom *month day year*** Display information for users who have some recorded accumulated usage since the specified date.
- l** Show the class-ad which was received from the central-manager in long format.

Exit Status

condor_userprio will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_vacate

Vacate jobs that are running on the specified hosts

Synopsis

condor_vacate [-help | -version]

condor_vacate [-graceful | -fast] [-debug] [-pool *centralmanagerhostname[:portnumber]* |
-name *name*]| [-addr "<*a.b.c.d:port*>"] ... [| -all]

Description

condor_vacate causes Condor to checkpoint any running jobs on a set of machines and force the jobs to vacate the machine. The job(s) remains in the submitting machine's job queue.

Given the (default) **-graceful** option, a job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under the vanilla universe is killed, and Condor restarts the job from the beginning somewhere else. *condor_vacate* has no effect on a machine with no Condor job currently running.

There is generally no need for the user or administrator to explicitly run *condor_vacate*. Condor takes care of jobs in this way automatically following the policies given in configuration files.

Options

-help Display usage information

-version Display version information

-graceful Inform the job to checkpoint, then soft-kill it.

-fast Hard-kill jobs instead of checkpointing them

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *name* Send the command to a machine identified by *name*

name Send the command to a machine identified by *name*

-addr "*<a.b.c.d:port>*" Send the command to a machine's master located at "*<a.b.c.d:port>*"

"*<a.b.c.d:port>*" Send the command to a machine located at "*<a.b.c.d:port>*"

-all Send the command to all machines in the pool

Exit Status

condor_vacate will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To send a *condor_vacate* command to two named machines:

```
% condor_vacate robin cardinal
```

To send the *condor_vacate* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_vacate -pool condor.cae.wisc.edu -name cae17
```

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_vacate_job

vacate jobs in the Condor queue from the hosts where they are running

Synopsis

condor_vacate_job [-help | -version]

condor_vacate_job [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
 [-addr "<*a.b.c.d:port*>"] [-fast *cluster... | cluster:process... | user...* | -constraint *expression*
 ...

condor_vacate_job [-pool *centralmanagerhostname[:portnumber]* | -name *scheddname*]
 [-addr "<*a.b.c.d:port*>"] [-fast] -all

Description

condor_vacate_job finds one or more jobs from the Condor job queue and vacates them from the host(s) where they are currently running. The jobs remain in the job queue and return to the idle state.

A job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under any other universe will be sent a soft kill signal (SIGTERM by default, or whatever is defined as the `SoftKillSig` in the job ClassAd), and Condor will restart the job from the beginning somewhere else.

If the **-fast** option is used, the job(s) will be immediately killed, meaning that standard universe jobs will not be allowed to checkpoint, and the job will have to revert to the last checkpoint or start over from the beginning.

If the **-name** option is specified, the named *condor_schedd* is targeted for processing. If the **-addr** option is used, the *condor_schedd* at the given address is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be vacated are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can vacate the job.

Using *condor_vacate_job* on jobs which are not currently running has no effect.

Options

-help Display usage information

-version Display version information

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number

-name *scheddname* Send the command to a machine identified by *scheddname*

-addr "<*a.b.c.d:port*>" Send the command to a machine located at "<*a.b.c.d:port*>"

cluster Vacate all jobs in the specified cluster

cluster.process Vacate the specific job in the cluster

user Vacate jobs belonging to specified user

-constraint *expression* Vacate all jobs which match the job ClassAd expression constraint

-all Vacate all the jobs in the queue

-fast Perform a fast vacate and hard kill the jobs

General Remarks

Do not confuse *condor_vacate_job* with *condor_vacate*. *condor_vacate* is given a list of hosts to vacate, regardless of what jobs happen to be running on them. Only machine owners and administrators have permission to use *condor_vacate* to evict jobs from a given host. *condor_vacate_job* is given a list of job to vacate, regardless of which hosts they happen to be running on. Only the owner of the jobs or queue super users have permission to use *condor_vacate_job*.

Examples

To vacate job 23.0:

```
% condor_vacate_job 23.0
```

To vacate all jobs of a user named Mary:

```
% condor_vacate_job mary
```

To vacate all standard universe jobs owned by Mary:

```
% condor_vacate_job -constraint 'JobUniverse == 1 && Owner == "mary"'
```

Note that the entire constraint, including the quotation marks, must be enclosed in single quote marks for most shells.

Exit Status

condor_vacate_job will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_version

print Condor version and platform information

Synopsis

condor_version [-help]

condor_version [-arch] [-opsys] [-syscall]

Description

With no arguments, *condor_version* prints the currently installed Condor version number and platform information. The version number includes a build identification number, as well as the date built.

Options

help Print usage information

arch Print this machine's ClassAd value for Arch

opsys Print this machine's ClassAd value for OpSys

syscall Get any requested version and/or platform information from the `libcondorsyscall.a` that this Condor pool is configured to use, instead of using the values that are compiled into the tool itself. This option may be used in combination with any other options to modify where the information is coming from.

Exit Status

condor_version will exit with a status value of 0 (zero) upon success, and it should never exit with a failing value.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

condor_wait

Wait for jobs to finish

Synopsis

condor_wait [-help | -version]

condor_wait [-debug] [-wait *seconds*] [-num *number-of-jobs*] *log-file* [**job ID**]

Description

condor_wait watches a user log file (created with the **log** command within a submit description file) and returns when one or more jobs from the log have completed or aborted.

Because *condor_wait* expects to find at least one job submitted event in the log file, at least one job must have been successfully submitted with *condor_submit* before *condor_wait* is executed.

condor_wait will wait forever for jobs to finish, unless a shorter wait time is specified.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-wait *seconds* Wait no more than the integer number of *seconds*. The default is unlimited time.

-num *number-of-jobs* Wait for the integer *number-of-jobs* jobs to end. The default is all jobs in the log file.

log file The name of the log file to watch for information about the job.

job ID A specific job or set of jobs to watch. If the **job ID** is only the job ClassAd attribute ClusterId, then *condor_wait* waits for all jobs with the given ClusterId. If the **job ID** is a pair of the job ClassAd attributes, given by ClusterId.ProcId, then *condor_wait* waits for the specific job with this **job ID**. If this option is not specified, all jobs that exist in

the log file when *condor_wait* is invoked will be watched.

General Remarks

condor_wait is an inexpensive way to test or wait for the completion of a job or a whole cluster, if you are trying to get a process outside of Condor to synchronize with a job or set of jobs.

It can also be used to wait for the completion of a limited subset of jobs, via the **-num** option.

Examples

```
condor_wait logfile
```

This command waits for all jobs that exist in *logfile* to complete.

```
condor_wait logfile 40
```

This command waits for all jobs that exist in *logfile* with a job ClassAd attribute *ClusterId* of 40 to complete.

```
condor_wait -num 2 logfile
```

This command waits for any two jobs that exist in *logfile* to complete.

```
condor_wait logfile 40.1
```

This command waits for job 40.1 that exists in *logfile* to complete.

```
condor_wait -wait 3600 logfile 40.1
```

This waits for job 40.1 to complete by watching *logfile*, but it will not wait more than one hour (3600 seconds).

Exit Status

condor_wait exits with 0 if and only if the specified job or jobs have completed or aborted. *condor_wait* returns 1 if unrecoverable errors occur, such as a missing log file, if the job does not exist in the log file, or the user-specified waiting time has expired.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

filelock_midwife

create an artifact of the creation of a process

Synopsis

filelock_midwife -help

filelock_midwife [-file *filename*] *program* [*programargs*]

Description

filelock_midwife starts a given *program*, while creating an artifact of the program's birth. At a later time the *filelock_undertaker* can examine the artifact to determine whether the program is still running, or whether the program has exited. *filelock_midwife* accomplishes this by obtaining a file lock on the given artifact file before starting the program.

Warning: *filelock_midwife* will not work on NFS unless the separate file lock server is running.

Options

-file *filename* The *filename* to use for the artifact file. The file `lock.file` is the default file used when this option is not specified.

***program* [*programargs*]** Forks a process and executes *program* with *programargs* as command-line arguments (when specified).

Exit Status

filelock_midwife will exit with a status of 0 (zero) upon success, and non-zero otherwise.

See Also

uniq_pid_midwife (on page 874), *filelock_undertaker* (on page 854).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

filelock_undertaker

determine whether a process has exited

Synopsis

filelock_undertaker -help

filelock_undertaker [-file *filename*] [-block]

Description

filelock_undertaker can examine an artifact file created by *filelock_midwife* and determine whether the program started by the *midwife* has exited. It does this by attempting to acquire a file lock.

Be warned that this will not work on NFS unless the separate file lock server is running.

Options

-block If the process has not exited, block until it does.

-file *filename* The name of the artifact file. created by *filelock_midwife*. The file `lock.file` is the default file used when this option is not specified.

Exit Status

filelock_undertaker will exit with a status of 0 (zero) if the monitored process has exited, with a status of 1 (one) if the monitored process has definitely not exited, with a status of 2 if it is uncertain whether the process has exited (this is generally due to a failure by the *filelock_midwife*), or with any other value for program failure.

See Also

uniq_pid_undertaker (on page 876), *filelock_midwife* (on page 852).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

install_release

install an arbitrary software release into a named directory

Synopsis

install_release [-help]

install_release [-f] [-basedir *directory*] [-log *filename*] [-wget] [-globuslocation *directory*]
[-o *otherfile1...*] *package*

Description

install_release installs an arbitrary software release into a named directory. In addition it creates a log of the installed files for easy uninstallation. This program can install packages of type tar, gzip, or gzip'ed tar. The installation package can be located on a mounted file system, an http server, an ftp server, or a grid ftp server.

Options

-basedir *directory* The directory where the package should be installed. When not specified, the directory defaults to the current working directory.

-f Forcefully overwrite files if they exist.

-globuslocation *directory* This program does not come prepackaged with *globus-url-copy* or the supporting libraries. If globus is not installed in the `/opt/globus` directory, the user must specify the installation location of globus using this option.

-help Display brief usage information and exit.

-log *filename* The file name for the installation log.

-o *otherfile1...* A space-separated list of files that will be installed along with the installation package. The files will only be copied. No extraction or decompression will be performed on these files. These files will be logged in the installation log.

package The full path to the installation package. Locations on file systems can be specified without the `file:` prefix, but other locations must prefix with the appropriate protocol (`http:`, `ftp:`, or `gsiftp:`).

-wget This program defaults to using *globus-url-copy* to fetch the installation package. This option specifies that this program should use *wget* for http and ftp requests and Perl's copy function for file system requests. *wget* must be installed on the machine and must be in the user's path.

Exit Status

install_release will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

See Also

cleanup_release (on page 673)

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_q

Query active Stork data placement jobs.

Synopsis

stork_q [-help | -version]

stork_q [-debug] [-name *server_specification*]

Description

stork_q prints the entire queue of active Stork jobs. Output is provided in the syntax of the ClassAd language. See <http://www.cs.wisc.edu/condor/classad> for information on the ClassAd language. Completed jobs are not printed. If the **-name** option is specified, the named *stork_server* is targeted for processing. Otherwise, the local *stork_server* is targeted.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-name *server_specification* Specification of *stork_server* using *machinename:port* or *<a.b.c.d:port>*.

Exit Status

stork_q will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_list_cred

list all credentials of a user stored on the *stork_credd* daemon on behalf of Stork

Synopsis

stork_list_cred [-help | -version]

stork_list_cred [-debug] -n *hostname:portnumber*

Description

stork_list_cred lists all credentials stored on the *stork_credd* daemon of the user issuing the *stork_list_cred* command. Two fields appear for each item listed: the unique name for the credential, and the string "X.509". As more credential types are handled by the *stork_credd* daemon, this second field will identify the credential type.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-n *hostname:portnumber* Identify the *stork_credd* daemon by host name and port number.

Exit Status

stork_list_cred will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_rm

remove a Stork job

Synopsis

stork_rm [-help | -version]

stork_rm [-debug] [-name *server_specification*] *job-id*

Description

stork_rm removes a Stork job from the queue, using the required command-line argument to identify the Stork job. *stork_rm* removes a single job from the Stork job queue. If the **-name** option is specified, the named *stork_server* is targeted for processing. Otherwise, the local *stork_server* is targeted. The job to be removed is identified by the job id returned by *stork_submit*.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-name *server_name* Name of *stork_server*.

-name *server_specification* Specification of *stork_server* using *machinename:port* or *<a.b.c.d:port>*.

Exit Status

stork_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_rm_cred

cause the *stork_credd* daemon to remove a credential

Synopsis

stork_rm_cred [-help | -version]

stork_rm_cred [-debug] [-N *credential-name*] -n *hostname:portnumber*

Description

stork_rm_cred removes a credential from the *stork_credd* daemon.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-n *hostname:portnumber* Identify the *stork_credd* daemon by host name and port number.

-N *credential-name* The unique name of the credential to be removed. When not specified, the *stork_credd* daemon uses the string "DEFAULT" as the unique name.

Exit Status

stork_rm_cred will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_store_cred

store a credential on the *stork_credd* daemon for use by Stork

Synopsis

stork_store_cred [-help | -version]

stork_store_cred [-debug] [-N *credential-name*] [-m [*user@*]*hostname*[:*portnumber*]]
[-D *proxy-server-name*] [-S] -n *hostname:portnumber* -f *filename* -t x509

Description

stork_store_cred stores a credential to a *stork_credd* daemon.

The required -t option identifies that the credential is an X.509 certificate.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-n *hostname:portnumber* Identify the *stork_credd* daemon by host name and port number.

-f *filename* A full path and file name where the credential is stored.

-N *credential-name* A unique name used by the *stork_credd* daemon to identify the credential.
When not specified, the *stork_credd* daemon uses the string "DEFAULT" as the unique name.

-m [*user@*]*hostname*[:*portnumber*] An identification of the *MyProxy* server.

-D *proxy-server-name* The distinguished name of the *MyProxy* server.

-S Read the *MyProxy* password from standard input.

Exit Status

stork_store_cred will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_status

print a Stork job's status

Synopsis

stork_status [-help | -version]

stork_status [-debug] [-name *server_specification*] *job-id*

Description

stork_status prints information about a Stork job. Jobs may be current or completed. If the **-name** option is specified, the named *stork_server* is targeted for processing. Otherwise, the local *stork_server* is targeted. The job to be removed is identified by the job id (as returned by *stork_submit*). The information is printed as a ClassAd. See <http://www.cs.wisc.edu/condor/classad> for information on the ClassAd language.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-name *server_name* Name of *stork_server*.

-name *server_specification* Specification of *stork_server* using *machinename:port* or *<a.b.c.d:port>*.

Exit Status

stork_status will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

stork_submit

submit a Stork job

Synopsis

stork_submit [-help | -version]

stork_submit [-debug] [-stdin] [-name *server_specification*] [-lognotes "*prose*"]
submit-description-file

Description

stork_submit is used to submit a Stork data placement job. Upon job submission, an integer identifier is assigned to the submission, and it is printed to standard output. This job identifier is required by other commands that manage Stork jobs.

The name of the Stork submit description file is the single, required, command-line argument. Stork places no constraints on the submit description file name. See section 2.13, for a more complete description of Stork.

STORK SUBMIT DESCRIPTION FILE COMMANDS Stork submit description files use ClassAd syntax, different from Condor submit file syntax. See <http://www.cs.wisc.edu/condor/classad/>.

dap_type = transfer; Required command identifying that there will be a transfer from source to destination.

arguments = "<argument_list>"; List of arguments to be supplied to the module on the command line. Arguments are delimited (separated) by space characters.

input = "<pathname>"; Stork assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (*stdin*, *stdout*, and *stderr*), must refer to files. Thus, the file name specified with **input** should contain any keyboard input the program requires (that is, this file becomes *stdin*). If not specified, the default value of */dev/null* is used for submission to a Unix machine.

stork_submit will prepend the current working directory if the pathname is relative (does not start with a */* character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host.

Note that this command does *not* refer to the command-line arguments of the program. The command-line arguments are specified by the **arguments** command.

output = "<pathname>"; The **output** file name will capture any information the program would normally write to the screen (that is, this file becomes `stdout`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. Multiple jobs should not use the same output file, since this will cause one job to overwrite the output of another. The output file and the error file should not be the same file as the outputs will overwrite each other or be lost.

Note that if your program explicitly opens and writes to a file, that file should *not* be specified as the output file.

stork_submit will prepend the current working directory if the pathname is relative (does not start with a `/` character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host.

err = "<pathname>"; The **err** file name will capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another. The error file and the output file should not be the same file as the outputs will overwrite each other or be lost.

stork_submit will prepend the current working directory if the pathname is relative (does not start with a `/` character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host.

log = "<pathname>"; Use **log** to specify a file name where Stork will write a log file of what is happening with this job cluster. For example, Stork will log into this file when and where the job begins running, when the job is checkpointed and/or migrated, when the job completes, etc. Most users find specifying a **log** file to be very handy; its use is recommended. If no **log** entry is specified, Stork does not create a log for this job.

stork_submit will prepend the current working directory if the pathname is relative (does not start with a `/` character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host. **log** file paths should not use NFS file systems.

log_xml = "True"; | "False"; If **log_xml** is true, then the log file will be written in ClassAd XML. If it isn't specified, XML is not used. Note that it's an XML fragment, and is missing the file header and footer. Also note that you should never mix XML and non-XML in a single file: if multiple jobs write to a single log file, it is up to you to make sure that all of them specify (or don't specify) this option in the same way.

src_url = <protocol-name:URL> A (required) URL to identify the data source, as well as the protocol to be used at the source. `file:///` URLs must refer to valid paths on the Stork server host.

dest_url = <protocol-name:URL> A (required) URL to identify the data destination, as well as the protocol to be used at the destination. `file:///` URLs must refer to valid paths on the Stork server host.

x509proxy = <path-to-proxy> The path to and file name of an X.509 proxy when needed for GSI authentication. A value of **"default"** directs Stork to search in the standard Globus GSI proxy locations.

cred_name = <credential-handle> Alternatively, a X.509 proxy may be managed via the Stork credential manager. The **cred_name** specifies the name by which the credential was stored in the credential manager. See the *stork_store_cred* manual on page 866.

alt_protocols = <sourceprotocol-desinationprotocol,sourceprotocol-desinationprotocol, ...>
A comma separated list of alternative protocol pairings to be used when a data transfer fails. For each pair, the protocol to use at the source of the transfer is followed by a - (dash) and the protocol to be used at the destination of the transfer. The list is used (together with the original **src.url** and **dest.url** protocols) in a round robin fashion. The source and destination URLs are unchanged; only the protocols to be used are changed.

Options

-help Display usage information

-version Display version information

-debug Show extra debugging information.

-stdin Read commands from `stdin` instead of from a file.

-name *server_specification* Specification of *stork_server* using *machinename:port* or <a.b.c.d:port>.

-lognotes "prose" The string given within quote marks is appended to the data placement ClassAd before the job is submitted.

Exit Status

stork_submit will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

uniq_pid_midwife

create an artifact of the creation of a process

Synopsis

uniq_pid_midwife [- **-noblock**] [- **-file** *filename*] [- **-precision** *seconds*] *program* [*programargs*]

Description

uniq_pid_midwife starts a given program, while creating an artifact of the program's birth. At a later time the *uniq_pid_undertaker* can examine the artifact to determine whether the program is still running or whether it has exited. *uniq_pid_midwife* accomplishes this by recording an enforced unique process identifier to the artifact.

Options

-file *filename* The *filename* to use for the artifact file. Defaults to `pid.file`.

-precision *seconds* The precision the operating system is expected to have in regards to process creation times. Defaults to an operating system specific value. The default is the best choice in most cases.

-noblock Exit after the program has been confirmed, typically 3 times the precision. Defaults to block until the program exits.

***program* [*programargs*]** Forks a process and executes *program* with *programargs* as command-line arguments (when specified).

Exit Status

uniq_pid_midwife will exit with a status of 0 (zero) upon success, and non-zero otherwise.

See Also

uniq_pid_undertaker (on page 876), *filelock_midwife* (on page 852).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

uniq_pid_undertaker

determine whether a process has exited

Synopsis

uniq_pid_undertaker [**- -block**] [**- -file** *file*] [**- -precision** *seconds*]

Description

uniq_pid_undertaker can examine an artifact file created by *uniq_pid_midwife* and determine whether the program started by the *midwife* has exited.

Options

- -block If the process has not exited, block until it does.

- -file *file* The name of the *uniq_pid_midwife* created artifact file. Defaults to `pid.file`.

- -precision *seconds* Uses *seconds* as the precision range within which the operating system will provide a process's birthday. Defaults to an operating system specific value. Only use this option if the same *seconds* value was provided to *uniq_pid_midwife*.

Exit Status

uniq_pid_undertaker will exit with a status of 0 (zero) if the monitored process has exited, with a status of 1 (one) if the monitored process has definitely not exited, with a status of 2 if it is uncertain whether the process has exited (this is generally due to a failure by the *uniq_pid_midwife*), or with any other value for program failure.

See Also

uniq_pid_midwife (on page 874), *filelock_undertaker* (on page 854).

Author

Condor Team, University of Wisconsin–Madison

Copyright

Copyright © 1990-2009 Condor Team, Computer Sciences Department, University of Wisconsin–Madison, Madison, WI. All Rights Reserved. Licensed under the Apache License, Version 2.0.

See the *Condor Version 7.2.3 Manual* or <http://www.condorproject.org/license> for additional notices.

Appendix A: ClassAd Attributes

ClassAd Types

ClassAd attributes vary, depending on the entity producing the ClassAd. Therefore, each ClassAd has an attribute named `MyType`, which describes the type of ClassAd. In addition, the *condor_collector* appends attributes to any daemon's ClassAd, whenever the *condor_collector* is queried. These additional attributes are listed in the unnumbered subsection labeled ClassAd Attributes Added by the *condor_collector* on page 894.

Here is a list of defined values for `MyType`, as well as a reference to a list attributes relevant to that type.

Job Each submitted job describes its state, for use by the *condor_negotiator* daemon in finding a machine upon which to run the job. ClassAd attributes that appear in a job ClassAd are listed and described in the unnumbered subsection labeled Job ClassAd Attributes on page 879.

Machine Each machine in the pool (and hence, the *condor_startd* daemon running on that machine) describes its state. ClassAd attributes that appear in a machine ClassAd are listed and described in the unnumbered subsection labeled Machine ClassAd Attributes on page 886.

DaemonMaster Each *condor_master* daemon describes its state. ClassAd attributes that appear in a DaemonMaster ClassAd are listed and described in the unnumbered subsection labeled DaemonMaster ClassAd Attributes on page 892.

Scheduler Each *condor_schedd* daemon describes its state. ClassAd attributes that appear in a Scheduler ClassAd are listed and described in the unnumbered subsection labeled Scheduler ClassAd Attributes on page 893.

Negotiator Each *condor_negotiator* daemon describes its state. ClassAd attributes that appear in a Negotiator ClassAd are listed and described in the unnumbered subsection labeled Negotiator ClassAd Attributes on page 894.

Query

This section has not yet been written

Job ClassAd Attributes

AllRemoteHosts String containing a comma-separated list of all the remote machines running a parallel or mpi universe job.

Args: String representing the arguments passed to the job.

CkptArch: String describing the architecture of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is undefined.

CkptOpSys: String describing the operating system of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is undefined.

ClusterId: Integer cluster identifier for this job. A cluster is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluster, but shares a common cluster identifier. The value changes each time a job or set of jobs are queued for execution under Condor.

Cmd: The path to and the file name of the job to be executed.

ConcurrencyLimits A string list, delimited by commas and space characters. The items in the list identify named resources that the job requires.

CommittedTime The number of seconds of wall clock time that the job has been allocated a machine, excluding the time spent on run attempts that were evicted without a checkpoint. Like RemoteWallClockTime, this includes time the job spent in a suspended state. In the case of RemoteWallClockTime, one can disregard suspended time by subtracting out CumulativeSuspensionTime. However, no analogous attribute representing committed suspension time exists, so it is not possible to compute how much of CommittedTime was actual time spent running.

CompletionDate The time when the job completed, or the value 0 if the job has not yet completed. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

CumulativeSuspensionTime A running total of the number of seconds the job has spent in suspension for the life of the job.

CurrentHosts The number of hosts in the claimed state, due to this job.

DiskUsage: Amount of disk space (Kbytes) in the Condor execute directory on the execute machine that this job has used. The initial estimate may be specified in the job submit file.

EmailAttributes A string containing a comma-separated list of job ClassAd attributes. For each attribute name in the list, its value will be included in the e-mail notification upon job completion.

EnteredCurrentStatus An integer containing the epoch time of when the job entered into its current status. So for example, if the job is on hold, the ClassAd expression

`CurrentTime - EnteredCurrentStatus`

will equal the number of seconds that the job has been on hold.

ExecutableSize Size of the executable in Kbytes.

ExitBySignal An attribute that is `True` when a user job exits via a signal and `False` otherwise. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitBySignal` is set to `False`.

ExitCode When a user job exits by means other than a signal, this is the exit return code of the user job. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitCode` is set to 0.

ExitSignal When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitSignal` will be undefined.

ExitStatus The way that Condor previously dealt with a job's exit status. This attribute should no longer be used. It is not always accurate in heterogeneous pools, or if the job exited with a signal. Instead, see the attributes: `ExitBySignal`, `ExitCode`, and `ExitSignal`.

GridJobStatus A string containing the job's status as reported by the remote job management system.

GridResource A string defined by the right hand side of the the submit description file command `grid_resource`. It specifies the target grid type, plus additional parameters specific to the grid type.

HoldKillSig Currently only for scheduler and local universe jobs, a string containing a name of a signal to be sent to the job if the job is put on hold.

HoldReasonCode An integer value that represents the reason that a job was put on hold.

HoldReasonSubCode An integer value that represents further information to go along with the `HoldReasonCode`, for some values of `HoldReasonCode`. See `HoldReasonCode` for the values.

HoldReason A string containing a human-readable message about why a job is on hold. This is the message that will be displayed in response to the command `condor_q -hold`. It can be used to determine if a job should be released or not.

HookKeyword A string that uniquely identifies a set of job hooks, and added to the `ClassAd` once a job is fetched.

<i>Integer Code</i>	<i>Reason for Hold</i>	<i>HoldReasonSubCode</i>
1	The user put the job on hold with <i>condor_hold</i> .	
2	Globus middleware reported an error.	The GRAM error number.
3	The PERIODIC_HOLD expression evaluated to True.	
4	The credentials for the job are invalid.	
5	A job policy expression evaluated to Undefined.	
6	The <i>condor_starter</i> failed to start the executable.	The Unix error number.
7	The standard output file for the job could not be opened.	The Unix error number.
8	The standard input file for the job could not be opened.	The Unix error number.
9	The standard output stream for the job could not be opened.	The Unix error number.
10	The standard input stream for the job could not be opened.	The Unix error number.
11	An internal Condor protocol error was encountered when transferring files.	
12	The <i>condor_starter</i> failed to download input files.	The Unix error number.
13	The <i>condor_starter</i> failed to upload output files.	The Unix error number.
14	The initial working directory of the job cannot be accessed.	The Unix error number.
15	The user requested the job be submitted on hold.	
16	Input files are being spooled.	
17	A standard universe job is not compatible with the <i>condor_shadow</i> version available on the submitting machine.	
18	An internal Condor protocol error was encountered when transferring files.	
19	HOOK_PREPARE_JOB was defined but couldn't be executed or returned failure.	
18	The job missed its deferred execution time and therefore failed to run.	

ImageSize: Estimate of the memory image size of the job in Kbytes. The initial estimate may be specified in the job submit file. Otherwise, the initial value is equal to the size of the executable. When the job checkpoints, the ImageSize attribute is set to the size of the checkpoint file (since the checkpoint file contains the job's memory image). A vanilla universe job's ImageSize is recomputed internally every 15 seconds.

JobLeaseDuration: The number of seconds set for a job lease, the amount of time that a job may continue running on a remote resource, despite its submitting machine's lack of response. See section 2.15.4 for details on job leases.

JobPrior: Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the greater (better) the priority.

JobRunCount: This attribute is retained for backwards compatibility. It may go away in the future. It is equivalent to NumShadowStarts for all universes except **scheduler**. For the **scheduler** universe, this attribute is equivalent to NumJobStarts.

JobStartDate: Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

JobStatus Integer which indicates the current status of the job.

<i>Value</i>	<i>Status</i>
0	Unexpanded (the job has never run)
1	Idle
2	Running
3	Removed
4	Completed
5	Held

JobUniverse Integer which indicates the job universe.

<i>Value</i>	<i>Universe</i>
1	standard
5	vanilla
7	scheduler
8	MPI
9	grid
10	java
11	parallel
12	local
13	vm

LastCheckpointPlatform An opaque string which is the CheckpointPlatform identifier from the last machine where this standard universe job had successfully produced a checkpoint.

LastCkptServer Host name of the last checkpoint server used by this job. When a pool is using multiple checkpoint servers, this tells the job where to find its checkpoint file.

LastCkptTime Time at which the job last performed a successful checkpoint. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

LastMatchTime An integer containing the epoch time when the job was last successfully matched with a resource (gatekeeper) Ad.

LastRejMatchReason If, at any point in the past, this job failed to match with a resource ad, this attribute will contain a string with a human-readable message about why the match failed.

LastRejMatchTime An integer containing the epoch time when Condor-G last tried to find a match for the job, but failed to do so.

-
- LastSuspensionTime** Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).
- LastVacateTime** Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).
- LocalSysCpu** An accumulated number of seconds of system CPU time that the job caused to the machine upon which the job was submitted.
- LocalUserCpu** An accumulated number of seconds of user CPU time that the job caused to the machine upon which the job was submitted.
- MaxHosts** The maximum number of hosts that this job would like to claim. As long as `CurrentHosts` is the same as `MaxHosts`, no more hosts are negotiated for.
- MaxJobRetirementTime** Maximum time in seconds to let this job run uninterrupted before kicking it off when it is being preempted. This can only decrease the amount of time from what the corresponding startd expression allows.
- MinHosts** The minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.
- NextJobStartDelay** An integer number of seconds delay time after this job starts until the next job is started. The value is limited by the configuration variable `MAX_NEXT_JOB_START_DELAY`.
- NiceUser** Boolean value which indicates whether
- NTDomain** A string that identifies the NT domain under which a job's owner authenticates on a platform running Windows.
- NumCkpts** A count of the number of checkpoints written by this job during its lifetime.
- NumGlobusSubmits** An integer that is incremented each time the *condor_gridmanager* receives confirmation of a successful job submission into Globus.
- NumJobMatches** An integer that is incremented by the *condor_schedd* each time the job is matched with a resource ad by the negotiator.
- NumJobStarts** An integer count of the number of times the job started executing. This is not (yet) defined for **standard** universe jobs.
- NumJobReconnects** An integer count of the number of times a job successfully reconnected after being disconnected. This occurs when the *condor_shadow* and *condor_starter* lose contact, for example because of transient network failures or a *condor_shadow* or *condor_schedd* restart. This attribute is only defined for jobs that can reconnect: those in the **vanilla** and **java** universes.
- NumRestarts** A count of the number of restarts from a checkpoint attempted by this job during its lifetime.
-

NumShadowExceptions An integer count of the number of times the *condor_shadow* daemon had a fatal error for a given job.

NumShadowStarts An integer count of the number of times a *condor_shadow* daemon was started for a given job. This attribute is not defined for **scheduler** universe jobs, since they do not have a *condor_shadow* daemon associated with them. For **local** universe jobs, this attribute *is* defined, even though the process that manages the job is technically a *condor_starter* rather than a *condor_shadow*. This keeps the management of the local universe and other universes as similar as possible.

NumSystemHolds An integer that is incremented each time Condor-G places a job on hold due to some sort of error condition. This counter is useful, since Condor-G will always place a job on hold when it gives up on some error condition. Note that if the user places the job on hold using the *condor_hold* command, this attribute is not incremented.

Owner: String describing the user who submitted this job.

ParallelShutdownPolicy A string that is only relevant to parallel universe jobs. Without this attribute defined, the default policy applied to parallel universe jobs is to consider the whole job completed when the first node exits, killing processes running on all remaining nodes. If defined to the following strings, Condor's behavior changes:

"WAIT_FOR_ALL" Condor will wait until every node in the parallel job has completed to consider the job finished.

ProcId: Integer process identifier for this job. Within a cluster of many jobs, each job has the same `ClusterId`, but will have a unique `ProcId`. Within a cluster, assignment of a `ProcId` value will start with the value 0. The job (process) identifier described here is unrelated to operating system PIDs.

QDate: Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

ReleaseReason A string containing a human-readable message about why the job was released from hold.

RemoteIwd: The path to the directory in which a job is to be executed on a remote machine.

RemoteSysCpu The total number of seconds of system CPU time (the time spent at system calls) the job used on remote machines. This does not count time spent on run attempts that were evicted without a checkpoint.

RemoteUserCpu The total number of seconds of user CPU time the job used on remote machines. This does not count time spent on run attempts that were evicted without a checkpoint.

RemoteWallClockTime Cumulative number of seconds the job has been allocated a machine. This also includes time spent in suspension (if any), so the total real time spent running is

`RemoteWallClockTime - CumulativeSuspensionTime`

Note that this number does not get reset to zero when a job is forced to migrate from one machine to another.

RemoveKillSig Currently only for scheduler universe jobs, a string containing a name of a signal to be sent to the job if the job is removed.

StreamErr: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferErr` is `True`, then standard error is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then standard error is transferred back to the submit machine (as a whole) after the job completes. If `TransferErr` is `False`, then this job attribute is ignored.

StreamOut: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferOut` is `True`, then job output is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then job output is transferred back to the submit machine (as a whole) after the job completes. If `TransferOut` is `False`, then this job attribute is ignored.

TotalSuspensions A count of the number of times this job has been suspended during its lifetime.

TransferErr: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the error output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Err`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Err`.

TransferExecutable An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job executable is transferred from the submit machine to the remote machine. The name of the file (on the submit machine) that is transferred is given by the job attribute `Cmd`. If `False`, no transfer takes place, and the name of the file used (on the remote machine) will be as given in the job attribute `Cmd`.

TransferIn An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job input is transferred from the submit machine to the remote machine. The name of the file that is transferred is given by the job attribute `In`. If `False`, then the job's input is taken from a file on the remote machine (pre-staged), and the name of the file is given by the job attribute `In`.

TransferOut: An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Out`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Out`.

WindowsBuildNumber An integer, extracted from the platform type of the machine upon which this job is submitted, representing a build number for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

WindowsMajorVersion An integer, extracted from the platform type of the machine upon which this job is submitted, representing a major version number (currently 5 or 6) for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

WindowsMinorVersion An integer, extracted from the platform type of the machine upon which this job is submitted, representing a minor version number (currently 0, 1, or 2) for a Windows operating system. This attribute only exists for jobs submitted from Windows machines.

The following job ClassAd attributes are relevant only for **vm** universe jobs.

VM_MACAddr: The MAC address of the virtual machine's network interface, in the standard format of six groups of two hexadecimal digits separated by colons. This attribute is currently limited to apply only to Xen virtual machines.

Machine ClassAd Attributes

Activity: String which describes Condor job activity on the machine. Can have one of the following values:

- "Idle":** There is no job activity
- "Busy":** A job is busy running
- "Suspended":** A job is currently suspended
- "Vacating":** A job is currently checkpointing
- "Killing":** A job is currently being killed
- "Benchmarking":** The startd is running benchmarks

Arch: String with the architecture of the machine. Typically one of the following:

- "ALPHA":** Digital Alpha CPU
- "HPPA1":** Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based workstation
- "HPPA2":** Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based workstation
- "IA64":** Intel Itanium
- "INTEL":** Intel x86 CPU (Pentium, Xeon, etc).
- "SGI":** Silicon Graphics MIPS CPU
- "SUN4u":** Sun UltraSparc CPU
- "SUN4x":** A Sun Sparc CPU other than an UltraSparc, i.e. sun4m or sun4c CPU found in older Sparc workstations such as the Sparc 10, Sparc 20, IPC, IPX, etc.
- "PPC":** Power Macintosh

"PPC64": 64-bit Power Macintosh

"x86_64": AMD/Intel 64-bit X86

CheckpointPlatform A string which opaquely encodes various aspects about a machine's operating system, hardware, and kernel attributes. It is used to identify systems where previously taken checkpoints for the standard universe may resume.

ClockDay: The day of the week, where 0 = Sunday, 1 = Monday, . . . , 6 = Saturday.

ClockMin: The number of minutes passed since midnight.

CondorLoadAvg The portion of the load average generated by Condor (either from remote jobs or running benchmarks).

ConsoleIdle The number of seconds since activity on the system console keyboard or console mouse has last been detected.

Cpus: Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

CurrentRank A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, **CurrentRank** is 0.0. When a machine is claimed, the attribute's value is computed by evaluating the machine's Rank expression with respect to the current job's ClassAd.

Disk: The amount of disk space on this machine available for the job in Kbytes (e.g. 23000 = 23 megabytes). Specifically, this is the amount of disk space available in the directory specified in the Condor configuration files by the EXECUTE macro, minus any space reserved with the RESERVED_DISK macro.

Dynamicslot For SMP machines that allow dynamic partitioning of a slot, this boolean value identifies that this dynamic slot may be partitioned.

EnteredCurrentActivity Time at which the machine entered the current Activity (see Activity entry above). On all platforms (including NT), this is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

FilesystemDomain A "domain" name configured by the Condor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

KeyboardIdle The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike **ConsoleIdle**, **KeyboardIdle** also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that **KeyboardIdle** will always be equal to or less than **ConsoleIdle**.

KFlops: Relative floating point performance as determined via a Linpack benchmark.

LastHeardFrom Time when the Condor central manager last received a status update from this machine. Expressed as the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

LoadAvg: A floating point number with the machine's current load average.

Machine: A string with the machine's fully qualified host name.

Memory: The amount of RAM in megabytes.

Mips: Relative integer performance as determined via a Dhrystone benchmark.

MyType: The ClassAd type; always set to the literal string "Machine".

Name: The name of this resource; typically the same value as the Machine attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

OpSys: String describing the operating system running on this machine. For Condor Version 7.2.3 typically one of the following:

"HPUX10": for HPUX 10.20

"HPUX11": for HPUX B.11.00

"LINUX": for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, or LINUX 2.6.x kernel systems

"OSF1": for Digital Unix 4.x

"OSX": for Darwin

"OSX10.2": for Darwin 6.4

"SOLARIS25": for Solaris 2.4 or 5.5

"SOLARIS251": for Solaris 2.5.1 or 5.5.1

"SOLARIS26": for Solaris 2.6 or 5.6

"SOLARIS27": for Solaris 2.7 or 5.7

"SOLARIS28": for Solaris 2.8 or 5.8

"SOLARIS29": for Solaris 2.9 or 5.9

"WINNT50": for Windows 2000

"WINNT51": for Windows XP

"WINNT52": for Windows Server 2003

"WINNT60": for Windows Vista

Requirements A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

MaxJobRetirementTime An expression giving the maximum time in seconds that the *startd* will wait for the job to finish before kicking it off if it needs to do so. This is evaluated in the context of the job ClassAd, so it may refer to job attributes as well as machine attributes.

PartitionableSlot For SMP machines, a boolean value identifying that this slot may be partitioned.

SlotID For SMP machines, the integer that identifies the slot. The value will be X for the slot with

```
name="slotX@full.hostname"
```

For non-SMP machines with one slot, the value will be 1. **NOTE:** This attribute was added in Condor version 6.9.3. For older versions of Condor, see *VirtualMachineID* below.

StartdIpAddr String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd.

State String which publishes the machine's Condor state. Can be:

"Owner": The machine owner is using the machine, and it is unavailable to Condor.

"Unclaimed": The machine is available to run Condor jobs, but a good match is either not available or not yet found.

"Matched": The Condor central manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

"Claimed": The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting": A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

TargetType Describes what type of ClassAd to match with. Always set to the string literal "Job", because machine ClassAds always want to be matched with jobs, and vice-versa.

TotalTimeBackfillBusy The number of seconds that this machine (slot) has accumulated within the backfill busy state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeBackfillIdle The number of seconds that this machine (slot) has accumulated within the backfill idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeBackfillKilling The number of seconds that this machine (slot) has accumulated within the backfill killing state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedBusy The number of seconds that this machine (slot) has accumulated within the claimed busy state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedIdle The number of seconds that this machine (slot) has accumulated within the claimed idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedRetiring The number of seconds that this machine (slot) has accumulated within the claimed retiring state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeClaimedSuspended The number of seconds that this machine (slot) has accumulated within the claimed suspended state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeMatchedIdle The number of seconds that this machine (slot) has accumulated within the matched idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeOwnerIdle The number of seconds that this machine (slot) has accumulated within the owner idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimePreemptingKilling The number of seconds that this machine (slot) has accumulated within the preempting killing state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimePreemptingVacating The number of seconds that this machine (slot) has accumulated within the preempting vacating state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeUnclaimedBenchmarking The number of seconds that this machine (slot) has accumulated within the unclaimed benchmarking state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

TotalTimeUnclaimedIdle The number of seconds that this machine (slot) has accumulated within the unclaimed idle state and activity pair since the *condor_startd* began executing. This attribute will only be defined if it has a value greater than 0.

UidDomain a domain name configured by the Condor administrator which describes a cluster of machines which all have the same passwd file entries, and therefore all have the same logins.

VirtualMachineID Starting with Condor version 6.9.3, this attribute is now longer used. Instead, use *SlotID*, as described above. This will only be present if *ALLOW_VM_CRUFT* is TRUE.

VirtualMemory The amount of currently available virtual memory (swap space) expressed in Kbytes.

WindowsBuildNumber An integer, extracted from the platform type, representing a build number for a Windows operating system. This attribute only exists on Windows machines.

WindowsMajorVersion An integer, extracted from the platform type, representing a major version number (currently 5 or 6) for a Windows operating system. This attribute only exists on Windows machines.

WindowsMinorVersion An integer, extracted from the platform type, representing a minor version number (currently 0, 1, or 2) for a Windows operating system. This attribute only exists on Windows machines.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

ClientMachine The host name of the machine that has claimed this resource

RemoteOwner The name of the user who originally claimed this resource.

RemoteUser The name of the user who is currently using this resource. In general, this will always be the same as the RemoteOwner, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, RemoteUser would hold the name of the entity currently using the resource, while RemoteOwner would hold the name of the entity that claimed the resource.

PreemptingOwner The name of the user who is preempting the job that is currently running on this resource.

PreemptingUser The name of the user who is preempting the job that is currently running on this resource. The relationship between PreemptingUser and PreemptingOwner is the same as the relationship between RemoteUser and RemoteOwner.

PreemptingRank A float which represents this machine owner's affinity for running the Condor job which is waiting for the current job to finish or be preempted. If not currently hosting a Condor job, PreemptingRank is undefined. When a machine is claimed and there is already a job running, the attribute's value is computed by evaluating the machine's Rank expression with respect to the preempting job's ClassAd.

TotalClaimRunTime A running total of the amount of time (in seconds) that all jobs (under the same claim) ran (have spent in the Claimed/Busy state).

TotalClaimSuspendTime A running total of the amount of time (in seconds) that all jobs (under the same claim) have been suspended (in the Claimed/Suspended state).

TotalJobRunTime A running total of the amount of time (in seconds) that a single job ran (has spent in the Claimed/Busy state).

TotalJobSuspendTime A running total of the amount of time (in seconds) that a single job has been suspended (in the Claimed/Suspended state).

There are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

JobId: The job's identifier (for example, 152.3), as seen from *condor_q* on the submitting machine.

JobStart: The time stamp in integer seconds of when the job began executing, since the Unix epoch (00:00:00 UTC, Jan 1, 1970). For idle machines, the value is UNDEFINED.

LastPeriodicCheckpoint If the job has performed a periodic checkpoint, this attribute will be defined and will hold the time stamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the LastPeriodicCheckpoint attribute will not be defined.

Finally, the single attribute, CurrentTime, is defined by the ClassAd environment.

CurrentTime Evaluates to the the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

DaemonMaster ClassAd Attributes

CkptServer: A string with with the fully qualified host name of the machine running a checkpoint server.

DaemonStartTime The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

Machine: A string with the machine's fully qualified host name.

MasterIpAddr: String with the IP and port address of the *condor_master* daemon which is publishing this DaemonMaster ClassAd.

MyAddress: Description is not yet written.

MyCurrentTime The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_master* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this resource; typically the same value as the Machine attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

PublicNetworkIpAddr Description is not yet written.

RealUid: The UID under which the *condor_master* is started.

UpdateSequenceNumber An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

Scheduler ClassAd Attributes

DaemonStartTime The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

JobQueueBirthdate Description is not yet written.

Machine A string with the machine's fully qualified host name.

MaxJobsRunning The same integer value as set in the configuration variable `MAX_JOBS_RUNNING`. See the definition at section 3.3.11 on page 195.

MyAddress Description is not yet written.

MyCurrentTime The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this resource; typically the same value as the *Machine* attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

NumUsers: The integer number of distinct users with jobs in this *condor_schedd*'s queue.

PublicNetworkIpAddress Description is not yet written.

QuillEnabled The same boolean value as set in the configuration variable `QUILL_ENABLED`. See the definition at section 3.3.30 on page 240.

ScheddIpAddress String with the IP and port address of the *condor_schedd* daemon which is publishing this Scheduler ClassAd.

ServerTime Description is not yet written.

StartLocalUniverse The same boolean value as set in the configuration variable `START_LOCAL_UNIVERSE`. See the definition at section 3.3.11 on page 195.

StartSchedulerUniverse The same boolean value as set in the configuration variable `START_SCHEDULER_UNIVERSE`. See the definition at section 3.3.11 on page 195.

TotalFlockedJobs The total number of jobs from this *condor_schedd* daemon that are currently flocked to other pools.

TotalHeldJobs The total number of jobs from this *condor_schedd* daemon that are currently on hold.

TotalIdleJobs The total number of jobs from this *condor_schedd* daemon that are currently idle.

TotalJobAds The total number of all jobs (in all states) from this *condor_schedd* daemon.

TotalRemovedJobs The current number of all running jobs from this *condor_schedd* daemon that have remove requests.

TotalRunningJobs The total number of jobs from this *condor_schedd* daemon that are currently running.

UpdateSequenceNumber An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

VirtualMemory Description is not yet written.

WantResAd A boolean value that when True causes the *condor_negotiator* daemon to send to this *condor_schedd* daemon a full machine ClassAd corresponding to a matched job.

Negotiator ClassAd Attributes

DaemonStartTime The time that this daemon was started, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

Machine A string with the machine's fully qualified host name.

MyAddress Description is not yet written.

MyCurrentTime The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_schedd* daemon last sent a ClassAd update to the *condor_collector*.

Name The name of this resource; typically the same value as the *Machine* attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate slots, each with with a unique name. These names will be of the form "slot#@full.hostname", for example, "slot1@vulture.cs.wisc.edu", which signifies slot number 1 from vulture.cs.wisc.edu.

NegotiatorIpAddr String with the IP and port address of the *condor_negotiator* daemon which is publishing this Negotiator ClassAd.

PublicNetworkIpAddr Description is not yet written.

UpdateSequenceNumber An integer, starting at zero, and incremented with each ClassAd update sent to the *condor_collector*. The *condor_collector* uses this value to sequence the updates it receives.

ClassAd Attributes Added by the *condor_collector*

These attributes are only added if `COLLECTOR_DAEMON_STATS` is True. See page 210 for more information on this setting.

LastHeardFrom The time inserted into a daemon's ClassAd representing the time that this *condor_collector* last received a message from the daemon. Time is represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

UpdatesHistory A bitmap representing the status of the most recent updates received from the daemon. This attribute is only added if `COLLECTOR_DAEMON_HISTORY_SIZE` is non-zero. See page 210 for more information on this setting.

UpdatesLost An integer count of the number of updates from the daemon that were lost since the *condor_collector* started running.

UpdatesSequenced An integer count of the number of updates received from the daemon for which the *condor_collector* can tell how many were lost, since the *condor_collector* started running.

UpdatesTotal An integer count started when the *condor_collector* started running, representing the sum of the number of updates actually received from the daemon plus the number of updates that the *condor_collector* determined were lost.

Appendix B: Magic Numbers

condor_shadow Exit Codes

<i>Value</i>	<i>Error Name</i>	
4	JOB_EXCEPTION	
44	DPRINTF_ERROR	
100	JOB_EXITED	
101	JOB_CKPTED	
102	JOB_KILLED	
103	JOB_COREDUMPED	
105	JOB_NO_MEM	
106	JOB_SHADOW_USAGE	
107	JOB_NOT_CKPTED	
107	JOB_SHOULD_REQUEUE	same number as JOB_NOT_CKPTED, to achieve the same behavior. Ho
108	JOB_NOT_STARTED	
109	JOB_BAD_STATUS	
110	JOB_EXEC_FAILED	
111	JOB_NO_CKPT_FILE	
112	JOB_SHOULD_HOLD	
113	JOB_SHOULD_REMOVE	
114	JOB_MISSED_DEFERRAL_TIME	
115	JOB_EXITED_AND_CLAIM_CLOSING	

User Log Event Codes

<i>Event Code</i>	<i>Description</i>
0	Submit
1	Execute
2	Executable error
3	Checkpointed
4	Job evicted
5	Job terminated
6	Image size
7	Shadow exception
8	Generic
9	Job aborted
10	Job suspended
11	Job unsuspended
12	Job held
13	Job released
14	Node execute
15	Node terminated
16	Post script terminated
17	Globus submit (no longer used)
18	Globus submit failed
19	Globus resource up (no longer used)
20	Globus resource down (no longer used)
21	Remote error
22	Job disconnected
23	Job reconnected
24	Job reconnect failed
25	Grid resource up
26	Grid resource down
27	Grid submit
28	Job ClassAd attribute values added to event log

Well-Known Ports

<i>Server</i>	<i>Port Number</i>
<i>condor_negotiator</i>	9614 (obsolete, now dynamically allocated)
<i>condor_collector</i>	9618
GT2 gatekeeper	2119
gridftp	2811
GT4 web services	8443

DaemonCore Commands and Signals

<i>Number</i>	<i>Name</i>
60000	DC_RAISESIGNAL
60001	DC_PROCESSEXIT
60002	DC_CONFIG_PERSIST
60003	DC_CONFIG_RUNTIME
60004	DC_RECONFIG
60005	DC_OFF_GRACEFUL
60006	DC_OFF_FAST
60007	DC_CONFIG_VAL
60008	DC_CHILDALIVE
60009	DC_SERVICEWAITPIDS
60010	DC_AUTHENTICATE
60011	DC_NOP
60012	DC_RECONFIG_FULL
60013	DC_FETCH_LOG
60014	DC_INVALIDATE_KEY
60015	DC_OFF_PEACEFUL
60016	DC_SET_PEACEFUL_SHUTDOWN
60017	DC_TIME_OFFSET

DaemonCore Daemon Exit Codes

<i>Exit Code</i>	<i>Description</i>
0	Normal exit of daemon
99	DAEMON_SHUTDOWN evaluated to True

INDEX

- <SUBSYS>_ADDRESS_FILE macro, 646
- <SUBSYS>_DAEMON_AD_FILE macro, 633, 646
- \$
 - as a literal character in a submit description file, 818
- \$ENV
 - in configuration file, 149
 - in submit description file, 819
- \$RANDOM_CHOICE()
 - in configuration, 149
 - in submit description file, 820
- \$RANDOM_INTEGER()
 - in configuration, 149, 555
- \$\$
 - as literal characters in a submit description file, 819
- _CONDOR_SCRATCH_DIR, 34
- _CONDOR_SLOT, 34
- ABORT_ON_EXCEPTION macro, 156
- ACCOUNTANT_LOCAL_DOMAIN macro, 212
- accounting
 - by group, 249
- ActivationTimer macro, 272
- activities and state figure, 261
- activity
 - of a machine, 260
 - transitions, 261–271
 - transitions summary, 270
- ActivityTimer macro, 272
- ADD_WINDOWS_FIREWALL_EXCEPTION macro, 178
- administrator's manual, 112–429
- AFS
 - interaction with, 108
- AfterHours macro, 277
- agents
 - condor_shadow, 16
- ALIVE_INTERVAL macro, 181, 197, 259
- ALL_DEBUG macro, 160
- ALLOW_* macro, 575
- ALLOW_* macros macro, 305
- ALLOW_ADMIN_COMMANDS macro, 178
- ALLOW_ADMINISTRATOR macro, 302
- ALLOW_ADVERTISE_MASTER macro, 303
- ALLOW_ADVERTISE_SCHEDD macro, 303
- ALLOW_ADVERTISE_STARTD macro, 303
- ALLOW_CLIENT macro, 229
- ALLOW_CLIENT macro, 303
- ALLOW_CONFIG macro, 533
- ALLOW_CONFIG macro, 302
- ALLOW_DAEMON macro, 653
- ALLOW_DAEMON macro, 302
- ALLOW_FORCE_RM macro, 664
- ALLOW_NEGOTIATOR macro, 302
- ALLOW_OWNER macro, 302
- ALLOW_READ macro, 302
- ALLOW_SOAP macro, 302
- ALLOW_VM_CRUFT macro, 34, 186, 890
- ALLOW_WRITE macro, 302
- AllRemoteHosts
 - job ClassAd attribute, 879
- ALWAYS_VM_UNIV_USE_NOBODY macro, 234
- Amazon EC2, 512
- AMAZON_EC2_URL macro, 219, 513
- AMAZON_GAHP macro, 219
- AMAZON_GAHP_WORKER_MAX_NUM macro, 598

- AMAZON_HTTP_PROXY macro, 219, 513
- API
 - Command line, 482
 - Condor GAHP, 482
 - DRMAA, 480
 - Perl module, 482
 - Web Service, 468
- APPEND_PREF_STANDARD macro, 206
- APPEND_PREF_VANILLA macro, 206
- APPEND_RANK macro, 206
- APPEND_RANK_STANDARD macro, 206, 351
- APPEND_RANK_VANILLA macro, 206
- APPEND_REQ_STANDARD macro, 206, 351
- APPEND_REQ_VANILLA macro, 206
- APPEND_REQUIREMENTS macro, 206
- ARCH macro, 148, 392
- Args
 - job ClassAd attribute, 879
- argv[0]
 - Condor use of, 111
- AUTH_SSL_CLIENT_CADIR macro, 231, 294
- AUTH_SSL_CLIENT_CAFILE macro, 231, 294
- AUTH_SSL_CLIENT_CERTFILE macro, 231, 294
- AUTH_SSL_CLIENT_KEYFILE macro, 231, 294
- AUTH_SSL_SERVER_CADIR macro, 231, 294
- AUTH_SSL_SERVER_CAFILE macro, 231, 294
- AUTH_SSL_SERVER_CERTFILE macro, 231, 294
- AUTH_SSL_SERVER_KEYFILE macro, 231, 294
- authentication, 287–298
 - GSI, 290
 - Kerberos, 294
 - Kerberos principal, 294
 - SSL, 293
 - using a file system, 297
 - using a remote file system, 297
 - Windows, 297
- authorization
 - for security, 302
- available platforms, 5
- Backfill, 412
 - BOINC Configuration in Condor, 416
 - BOINC Installation, 415
 - BOINC Overview, 414
 - Defining Condor policy, 413
 - Overview, 413
- backfill state, 257, 269
- BACKFILL_SYSTEM macro, 183, 413
- BackgroundLoad macro, 272
- batch system, 11
- BIN macro, 151
- BIND_ALL_INTERFACES macro, 165, 339, 601
- BOINC_Arguments macro, 417, 419
- BOINC_Environment macro, 417
- BOINC_Error macro, 417
- BOINC_Executable macro, 415, 416, 419
- BOINC_InitialDir macro, 415, 416, 418, 419
- BOINC_Output macro, 417
- BOINC_Owner macro, 415, 416, 419
- BOINC_Universe macro, 416
- C_GAHP_LOG macro, 219, 495
- C_GAHP_WORKER_THREAD_LOG macro, 219
- central manager, 112, 113
 - installation issues, 119
- certificate
 - X.509, 290
- CERTIFICATE_MAPFILE macro, 231, 298
- checkpoint, 2, 3, 16, 443
 - compression, 444
 - implementation, 443
 - library interface, 446
 - periodic, 3, 443, 555
 - stand alone, 444
- checkpoint image, 16
- checkpoint server, 113
 - configuration of, 347
 - installation, 346–351
 - multiple servers, 348
- Chirp, 53
 - Chirp.jar, 54
 - ChirpClient, 53
 - ChirpInputStream, 53
 - ChirpOutputStream, 53
- CKPT_PROBE macro, 156, 618

-
- CKPT_SERVER_CHECK_PARENT_INTERVAL
 - macro, 173
 - CKPT_SERVER_DEBUG macro, 348
 - CKPT_SERVER_DIR macro, 172, 348
 - CKPT_SERVER_HOST macro, 172, 332, 344, 348, 349
 - CKPT_SERVER_LOG macro, 348
 - CkptArch
 - job ClassAd attribute, 879
 - CkptOpSys
 - job ClassAd attribute, 879
 - claim lease, 259
 - CLAIM_WORKLIFE macro, 180, 270, 280
 - claimed state, 257, 265
 - ClassAd, 2, 4, 12, 430–442
 - attributes, 12, 432
 - attributes added by the *condor_collector*, 894
 - DaemonMaster attributes, 892
 - expression functions, 432
 - expression operators, 432, 440
 - expression syntax, 431
 - job, 12
 - job attributes, 879
 - machine, 12
 - machine attributes, 886
 - machine example, 13
 - Negotiator attributes, 894
 - Scheduler attributes, 893
 - scope of evaluation, MY., 439
 - scope of evaluation, TARGET., 439
 - ClassAd attribute
 - CurrentTime, 892
 - rank, 22, 441
 - rank examples, 23
 - requirements, 22, 441
 - ClassAd attribute added by the *condor_collector*
 - LastHeardFrom, 894
 - UpdatesHistory, 210, 895
 - UpdatesLost, 210, 895
 - UpdatesSequenced, 210, 895
 - UpdatesTotal, 210, 895
 - ClassAd DaemonMaster attribute
 - CkptServer, 892
 - DaemonStartTime, 892
 - Machine, 892
 - MasterIpAddr, 892
 - MyAddress, 892
 - MyCurrentTime, 892
 - Name, 892
 - PublicNetworkIpAddr, 892
 - RealUid, 892
 - UpdateSequenceNumber, 892
 - ClassAd functions, 432
 - ceiling(), 434
 - floor(), 434
 - formatTime(), 436
 - ifThenElse(), 433
 - int(), 433
 - interval(), 437
 - isBoolean(), 433
 - isError(), 433
 - isInteger(), 433
 - isReal(), 433
 - isString(), 433
 - isUndefined(), 433
 - random(), 434
 - real(), 434
 - regex(), 438
 - regexps(), 438
 - round(), 434
 - size(), 435
 - strcat(), 435
 - strcmp(), 435
 - stricmp(), 435
 - string(), 434
 - stringListAve(), 437
 - stringListIMember(), 438
 - stringListMax(), 437
 - stringListMember(), 438
 - stringListMin(), 437
 - stringListRegexpMember(), 438
 - stringListSize(), 437
 - stringListSum(), 437
 - substr(), 435
 - time(), 436
 - toLower(), 435
 - toUpper(), 435
 - ClassAd job attribute
 - AccountingGroup, 249
 - AllRemoteHosts, 879
 - Args, 879
-

CkptArch, 879
CkptOpSys, 879
ClusterId, 818, 879
Cmd, 879
CommittedTime, 879
CompletionDate, 879
ConcurrencyLimits, 879
CumulativeSuspensionTime, 879
CurrentHosts, 879
DeferralPrepTime, 96
DeferralTime, 95
DeferralWindow, 96
DiskUsage, 879
EmailAttributes, 879
EnteredCurrentStatus, 879
ExecutableSize, 880
ExitBySignal, 880
ExitCode, 880
ExitSignal, 880
ExitStatus, 880
GridJobStatus, 880
GridResource, 880
HoldKillSig, 880
HoldReason, 880
HoldReasonCode, 880
HoldReasonSubCode, 880
ImageSize, 880
JobLeaseDuration, 110, 881
JobPrio, 881
JobRunCount, 881
JobStartDate, 881
JobStatus, 881
JobUniverse, 882
LastCheckpointPlatform, 882
LastCkptServer, 882
LastCkptTime, 882
LastMatchTime, 882
LastRejMatchReason, 882
LastRejMatchTime, 882
LastSuspensionTime, 882
LastVacateTime, 883
LocalSysCpu, 883
LocalUserCpu, 883
MaxHosts, 883
MaxJobRetirementTime, 883
MinHosts, 883
NextJobStartDelay, 883
NiceUser, 883
NTDomain, 883
NumCkpts, 883
NumGlobusSubmits, 883
NumJobMatches, 883
NumJobReconnects, 883
NumJobStarts, 883
NumRestarts, 883
NumShadowExceptions, 883
NumShadowStarts, 884
NumSystemHolds, 884
Owner, 884
ParallelShutdownPolicy, 884
ProcId, 884
QDate, 884
ReleaseReason, 884
RemoteIwd, 884
RemoteSysCpu, 884
RemoteUserCpu, 884
RemoteWallClockTime, 884
RemoveKillSig, 885
StreamErr, 885
StreamOut, 885
TotalSuspensions, 885
TransferErr, 885
TransferExecutable, 885
TransferIn, 885
TransferOut, 885
VM_MACAddr, 886
WindowsBuildNumber, 885
WindowsMajorVersion, 885
WindowsMinorVersion, 886
ClassAd machine attribute
 Activity, 886
 Arch, 886
 AvailSince, 192
 AvailTime, 192
 AvailTimeEstimate, 192
 CheckpointPlatform, 887
 ClockDay, 887
 ClockMin, 887
 CondorLoadAvg, 887
 ConsoleIdle, 887
 Cpus, 887
 CurrentRank, 887

- Disk, 887
- DynamicSlot, 887
- EnteredCurrentActivity, 887
- FileSystemDomain, 887
- HookKeyword, 880
- KeyboardIdle, 887
- KFlops, 887
- LastAvailInterval, 192
- LastHeardFrom, 887
- LoadAvg, 888
- Machine, 888
- MaxJobRetirementTime, 888
- Memory, 888
- Mips, 888
- MyType, 888
- Name, 888
- OpSys, 888
- PartitionableSlot, 889
- Requirements, 888
- SlotID, 889
- StartdIpAddr, 889
- State, 889
- TargetType, 889
- TotalTimeBackfillBusy, 889
- TotalTimeBackfillIdle, 889
- TotalTimeBackfillKilling, 889
- TotalTimeClaimedBusy, 889
- TotalTimeClaimedIdle, 889
- TotalTimeClaimedRetiring, 890
- TotalTimeClaimedSuspended, 890
- TotalTimeMatchedIdle, 890
- TotalTimeOwnerIdle, 890
- TotalTimePreemptingKilling, 890
- TotalTimePreemptingVacating, 890
- TotalTimeUnclaimedBenchmarking, 890
- TotalTimeUnclaimedIdle, 890
- UidDomain, 890
- VirtualMachineID, 890
- VirtualMemory, 890
- WindowsBuildNumber, 890
- WindowsMajorVersion, 890
- WindowsMinorVersion, 891
- ClassAd machine attribute (in Claimed State)
 - ClientMachine, 891
 - PreemptingOwner, 891
 - PreemptingRank, 891
 - PreemptingUser, 891
 - RemoteOwner, 891
 - RemoteUser, 891
 - TotalClaimRunTime, 891
 - TotalClaimSuspendTime, 891
 - TotalJobRunTime, 891
 - TotalJobSuspendTime, 891
- ClassAd machine attribute (when running)
 - JobId, 891
 - JobStart, 892
 - LastPeriodicCheckpoint, 892
- ClassAd Negotiator attribute
 - DaemonStartTime, 894
 - Machine, 894
 - MyAddress, 894
 - MyCurrentTime, 894
 - Name, 894
 - NegotiatorIpAddr, 894
 - PublicNetworkIpAddr, 894
 - UpdateSequenceNumber, 894
- ClassAd Scheduler attribute
 - DaemonStartTime, 893
 - JobQueueBirthdate, 893
 - Machine, 893
 - MaxJobsRunning, 893
 - MyAddress, 893
 - MyCurrentTime, 893
 - Name, 893
 - NumUsers, 893
 - PublicNetworkIpAddr, 893
 - QuillEnabled, 893
 - ScheddIpAddr, 893
 - ServerTime, 893
 - StartLocalUniverse, 893
 - StartSchedulerUniverse, 893
 - TotalFlockedJobs, 893
 - TotalHeldJobs, 893
 - TotalIdleJobs, 893
 - TotalJobAds, 893
 - TotalRemovedJobs, 893
 - TotalRunningJobs, 894
 - UpdateSequenceNumber, 894
 - VirtualMemory, 894
 - WantResAd, 894
- CLASSAD_LIFETIME macro, 208
- cleanup_release, 673

- CLIENT_TIMEOUT macro, 208
- clipped platform
 - availability, 5
 - definition of, 5
- clock skew, 564
- cluster
 - definition, 879
- Cluster macro, 818
- ClusterId
 - job ClassAd attribute, 818, 879
- CM_IP_ADDR macro, 155
- Cmd
 - job ClassAd attribute, 879
- COD
 - attributes, 449
 - ClusterId, 451
 - ProcID, 451
 - authorizing users, 448
 - condor_cod tool, 452
 - condor_cod_activate command, 451, 455
 - condor_cod_deactivate command, 458
 - condor_cod_delegate_proxy command, 459
 - condor_cod_release command, 458
 - condor_cod_renew command, 457
 - condor_cod_request command, 453
 - condor_cod_resume command, 457
 - condor_cod_suspend command, 456
 - defining an application, 448
 - defining applications
 - Job ID, 451
 - defining attributes by configuration, 451
 - introduction, 447
 - limitations, 459
 - managing claims, 452
 - optional attributes, 449
 - Args, 450
 - Env, 450
 - Err, 450
 - In, 449
 - IWD, 449
 - JarFiles, 450
 - JobUniverse, 450
 - KillSig, 450
 - Out, 450
 - StarterUserLog, 450
 - StarterUserLogUseXML, 450
 - overview, 448
 - required attributes, 449
 - Cmd, 449
 - Owner, 449
- COD (Computing on Demand), 447–460
- COLLECTOR_ADDRESS_FILE macro, 327
- COLLECTOR_ADDRESS_FILE macro, 163
- COLLECTOR_CLASS_HISTORY_SIZE macro, 211
- COLLECTOR_DAEMON_HISTORY_SIZE macro, 210, 211, 677, 832, 895
- COLLECTOR_DAEMON_STATS macro, 210, 211, 894
- COLLECTOR_DEBUG macro, 211
- COLLECTOR_HOST macro, 150, 178, 327, 344, 565, 637, 640, 753
- COLLECTOR_NAME macro, 209
- COLLECTOR_QUERY_WORKERS macro, 211
- COLLECTOR_REQUIREMENTS macro, 208
- COLLECTOR_SOCKET_BUFSIZE macro, 209
- COLLECTOR_SOCKET_CACHE_SIZE macro, 167, 209, 345
- COLLECTOR_STATS_SWEEP macro, 210, 619
- COLLECTOR_TCP_SOCKET_BUFSIZE macro, 209, 659
- CommittedTime
 - job ClassAd attribute, 879
- compilers
 - supported with condor_compile, 7
- CompletionDate
 - job ClassAd attribute, 879
- COMPRESS_PERIODIC_CHKPT macro, 203
- COMPRESS_VACATE_CHKPT macro, 203
- Computing On Demand
 - Defining Applications
 - Job ID, 451
 - Optional attributes, 449
 - Required attributes, 449
- Computing on Demand (see COD), 447
- concurrency limits, 420
- CONCURRENCY_LIMIT_DEFAULT macro, 421
- ConcurrencyLimits
 - job ClassAd attribute, 879
- Condor
 - binaries, 547
 - configuration, 142

- contact information, 9, 578
- contributions, 8
- default policy, 272
- distribution, 547
- downloading, 547
- FAQ, 547–578
- flocking, 492
- Frequently Asked Questions, 547–578
- getting, 547
- limitations, under UNIX, 4
- mailing lists, 9, 578
- new versions, notification of, 578
- overview, 1–4
- Personal, 550
- platforms available, 5
- pool, 112
- resource allocation, 12
- resource management, 2
- shared functionality in daemons, 351
- source code, 548
- universe, 15
- Unix administrator, 119
- user manual, 11–111
- Condor commands
 - condor_advertise*, 675
 - condor_check_userlogs*, 679
 - condor_checkpoint*, 680
 - condor_chirp*, 683
 - condor_cod*, 686
 - condor_cold_start*, 140, 689
 - condor_cold_stop*, 140, 692
 - condor_compile*, 48, 695
 - list of supported compilers, 7
 - condor_config_bind*, 698
 - condor_config_val*, 700
 - condor_configure*, 704
 - condor_convert_history*, 709
 - condor_dagman*, 711
 - condor_fetchlog*, 715
 - condor_findhost*, 718
 - condor_glidein*, 519, 720
 - condor_history*, 727
 - condor_hold*, 40, 730
 - condor_install*, 704
 - condor_load_history*, 733
 - condor_master*, 735
 - condor_master_off*, 737
 - condor_off*, 738
 - condor_on*, 741
 - condor_power*, 744
 - condor_preen*, 746
 - condor_prio*, 41, 47, 748
 - condor_q*, 15, 38, 41, 750
 - condor_qedit*, 758
 - condor_reconfig*, 760
 - condor_reconfig_schedd*, 763
 - condor_release*, 40, 764
 - condor_reschedule*, 766
 - condor_restart*, 769
 - condor_rm*, 15, 40, 772
 - condor_run*, 777
 - condor_set_shutdown*, 781
 - condor_stats*, 783
 - condor_status*, 13, 15, 22, 38, 39, 787
 - condor_store_cred*, 793
 - condor_submit*, 15, 19, 109, 795
 - condor_submit_dag*, 825
 - condor_transfer_data*, 830
 - condor_updates_stats*, 832
 - condor_userprio*, 47, 838
 - condor_vacate*, 841
 - condor_vacate_job*, 844
 - condor_version*, 847
 - condor_wait*, 849
 - really slow; why?, 565
 - stork_list_cred*, 860
 - stork_q*, 858
 - stork_rm*, 862
 - stork_rm_cred*, 864
 - stork_status*, 868
 - stork_store_cred*, 866
 - stork_submit*, 870
- Condor daemon
 - condor_had*, 357
 - condor_master* manual page, 735
 - command line arguments, 352
 - condor_ckpt_server*, 115
 - condor_collector*, 115
 - condor_credd*, 116, 216, 533
 - condor_dbmsd*, 115
 - condor_gridmanager*, 115
 - condor_had*, 116

- condor_job_router*, 116, 521
- condor_kbdd*, 115
- condor_lease_manager*, 106, 116
- condor_master*, 114
- condor_negotiator*, 115
- condor_procd*, 116
- condor_quill*, 115
- condor_replication*, 116
- condor_schedd*, 114
- condor_shadow*, 16, 109
- condor_shadow*, 114
- condor_startd*, 114, 253
- condor_starter*, 114
- descriptions, 114
- stork_server*, 116
- Condor GAHP, 494
- Condor-C, 494–498
 - configuration, 494
 - job submission, 495
 - limitations, 498
- Condor-G, 498–509
 - GASS, 499
 - GRAM, 499
 - GSI, 498
 - job submission, 500
 - limitations, 509
 - proxy, 500
 - X.509 certificate, 500
- CONDOR_ADMIN macro, 154
- condor_advertise* command, 675
- condor_check_userlogs* command, 679
- condor_checkpoint* command, 680
- condor_chirp* command, 683
- condor_ckpt_server* daemon, 115
- condor_cod* command, 686
- condor_cold_start*, 689
- condor_cold_stop*, 692
- condor_collector*, 345
- condor_collector* daemon, 115
- condor_compile*, 559
- condor_compile* command, 695
 - list of supported compilers, 7
- CONDOR_CONFIG macro, 137, 638
- condor_config_bind* command, 698
- condor_config_val* command, 700
- condor_configure*, 123
- condor_configure* command, 704
- condor_convert_history* command, 709
- condor_credd* daemon, 116, 216, 533
- condor_dagman* command, 711
- condor_dbmsd* daemon, 115
- CONDOR_DEVELOPERS macro, 10, 208
- CONDOR_DEVELOPERS_COLLECTOR macro, 10, 209
- condor_fetchlog* command, 715
- condor_findhost* command, 718
- CONDOR_GAHP macro, 219, 495
- condor_glidein* command, 720
- condor_gridmanager* daemon, 115
- condor_had* daemon, 116
- condor_history* command, 727
- condor_hold* command, 730
- CONDOR_HOST macro, 150, 331
- CONDOR_IDS
 - environment variable, 120, 154
- CONDOR_IDS macro, 120, 153, 316, 613, 642
- condor_install* command, 704
- CONDOR_JOB_POLL_INTERVAL macro, 217
- condor_job_router* daemon, 116, 521
- condor_kbdd* daemon, 115
- condor_lease_manager* daemon, 106, 116
- condor_load_history* command, 733
- condor_master* command, 735
- condor_master* daemon, 114
- condor_master_off* command, 737
- condor_negotiator* daemon, 115
- condor_off* command, 738
- condor_on* command, 741
- condor_power* command, 744
- condor_preen* command, 746
- condor_prio* command, 748
- condor_procd* daemon, 116
- condor_q* command, 750
- condor_qedit* command, 758
- condor_quill* daemon, 115
- condor_reconfig* command, 760
- condor_reconfig_schedd* command, 763
- condor_release* command, 764
- condor_replication* daemon, 116
- condor_reschedule* command, 766
- condor_restart* command, 769
- condor_rm* command, 772

- condor_router_history, 775
- condor_run command, 777
- condor_schedd daemon, 114
 - receiving signal 25, 576
- condor_set_shutdown command, 781
- condor_shadow, 16, 39
- condor_shadow daemon, 114
- CONDOR_SSH_KEYGEN macro, 59
- CONDOR_SSHD macro, 59
- condor_startd daemon, 114
- condor_startd* daemon, 253
- condor_starter daemon, 114
- condor_stats command, 783
- condor_status command, 787
- condor_store_cred command, 793
- condor_submit command, 795
- condor_submit_dag command, 825
- CONDOR_SUPPORT_EMAIL macro, 154
- condor_transfer_data command, 830
- condor_updates_stats command, 832
- condor_userprio command, 838
- condor_vacate command, 841
- condor_vacate_job command, 844
- condor_version command, 847
- CONDOR_VIEW_HOST macro, 150, 617
- CONDOR_VM, 34
- condor_wait command, 849
- CondorView
 - Client, 138
 - Client installation, 139
 - configuration, 396
 - Server, 396
 - use of *crontab* program, 140
- configuration, 142
 - checkpoint server configuration variables, 172
 - Condor-wide configuration variables, 150
 - condor_collector configuration variables, 208
 - condor_credd configuration variables, 216
 - condor_gridmanager configuration variables, 216
 - condor_job_router configuration variables, 220
 - condor_lease_manager configuration variables, 221
 - condor_master configuration variables, 173
 - condor_negotiator configuration variables, 211
 - condor_preen configuration variables, 207
 - condor_schedd configuration variables, 195
 - condor_shadow configuration variables, 202
 - condor_startd configuration variables, 179
 - condor_starter configuration variables, 203
 - condor_submit configuration variables, 205
 - daemon logging configuration variables, 157
 - DaemonCore configuration variables, 162
 - DAGMan configuration variables, 223
 - example, 255
 - for flocking, 492
 - for glidein, 519
 - grid and glidein configuration variables, 223
 - grid_monitor configuration variables, 222
 - high availability configuration variables, 235
 - network-related configuration variables, 165
 - of machines, to implement a given policy, 252
 - pre-defined macros, 147
 - PrivSep configuration variables, 232
 - Quill configuration variables, 239
 - security configuration variables, 229
 - shared file system configuration variables, 168
 - SMP machines, 403
 - startd policy, 252
 - virtual machine configuration variables, 233
- configuration file
 - \$ENV definition, 149
 - evaluation order, 143
 - macro definitions, 143
 - macros, 148
 - pre-defined macros, 147
 - subsystem names, 147
- configuration files
 - location, 121
- configuration macro

- <SUBSYS>_<LEVEL>_LOG, 160
- <SUBSYS>_ADDRESS_FILE, 163, 326, 646
- <SUBSYS>_ARGS, 174
- <SUBSYS>_ATTRS, 163
- <SUBSYS>_DAEMON_AD_FILE, 163, 633, 646
- <SUBSYS>_DEBUG, 158
- <SUBSYS>_ENABLE_SOAP_SSL, 242
- <SUBSYS>_EXPRS, 163
- <SUBSYS>_LOCK, 158
- <SUBSYS>_LOG, 157
- <SUBSYS>_NOT_RESPONDING_TIMEOUT, 164
- <SUBSYS>_SOAP_SSL_PORT, 243
- <SUBSYS>_TIMEOUT_MULTIPLIER, 167
- <SUBSYS>, 173
- ABORT_ON_EXCEPTION, 156
- ACCOUNTANT_LOCAL_DOMAIN, 212
- ADD_WINDOWS_FIREWALL_EXCEPTION, 178
- ALIVE_INTERVAL, 181, 197, 259
- ALLOW_* macros, 305
- ALLOW_*, 575
- ALLOW_ADMIN_COMMANDS, 178
- ALLOW_CLIENT, 229
- ALLOW_CONFIG, 533
- ALLOW_DAEMON, 653
- ALLOW_FORCE_RM, 664
- ALLOW_VM_CRUFT, 34, 186, 890
- ALL_DEBUG, 160
- ALWAYS_VM_UNIV_USE_NOBODY, 234
- AMAZON_EC2_URL, 219, 513
- AMAZON_GAHP_WORKER_MAX_NUM, 598
- AMAZON_GAHP, 219
- AMAZON_HTTP_PROXY, 219, 513
- APPEND_PREF_STANDARD, 206
- APPEND_PREF_VANILLA, 206
- APPEND_RANK_STANDARD, 206, 351
- APPEND_RANK_VANILLA, 206
- APPEND_RANK, 206
- APPEND_REQUIREMENTS, 206
- APPEND_REQ_STANDARD, 206, 351
- APPEND_REQ_VANILLA, 206
- ARCH, 148, 392
- AUTH_SSL_CLIENT_CADIR, 231, 294
- AUTH_SSL_CLIENT_CAFILE, 231, 294
- AUTH_SSL_CLIENT_CERTFILE, 231, 294
- AUTH_SSL_CLIENT_KEYFILE, 231, 294
- AUTH_SSL_SERVER_CADIR, 231, 294
- AUTH_SSL_SERVER_CAFILE, 231, 294
- AUTH_SSL_SERVER_CERTFILE, 231, 294
- AUTH_SSL_SERVER_KEYFILE, 231, 294
- ActivationTimer, 272
- ActivityTimer, 272
- AfterHours, 277
- BACKFILL_SYSTEM, 183, 413
- BIND_ALL_INTERFACES, 165, 339, 601
- BIN, 151
- BOINC_Arguments, 417, 419
- BOINC_Environment, 417
- BOINC_Error, 417
- BOINC_Executable, 415, 416, 419
- BOINC_InitialDir, 415, 416, 418, 419
- BOINC_Output, 417
- BOINC_Owner, 415, 416, 419
- BOINC_Universe, 416
- BackgroundLoad, 272
- CERTIFICATE_MAPFILE, 231, 298
- CKPT_PROBE, 156, 618
- CKPT_SERVER_CHECK_PARENT_INTERVAL, 173
- CKPT_SERVER_DEBUG, 348
- CKPT_SERVER_DIR, 172, 348
- CKPT_SERVER_HOST, 172, 332, 344, 348, 349
- CKPT_SERVER_LOG, 348
- CLAIM_WORKLIFE, 180, 270, 280
- CLASSAD_LIFETIME, 208
- CLIENT_TIMEOUT, 208
- CM_IP_ADDR, 155
- COLLECTOR_ADDRESS_FILE, 327
- COLLECTOR_CLASS_HISTORY_SIZE, 211
- COLLECTOR_DAEMON_HISTORY_SIZE, 210, 211, 677, 832, 895
- COLLECTOR_DAEMON_STATS, 210, 211, 894
- COLLECTOR_DEBUG, 211

- COLLECTOR_HOST, 150, 178, 327, 344, 565, 637, 640, 753
- COLLECTOR_NAME, 209
- COLLECTOR_QUERY_WORKERS, 211
- COLLECTOR_REQUIREMENTS, 208
- COLLECTOR_SOCKET_BUFSIZE, 209
- COLLECTOR_SOCKET_CACHE_SIZE, 167, 209, 345
- COLLECTOR_STATS_SWEEP, 210, 619
- COLLECTOR_TCP_SOCKET_BUFSIZE, 209, 659
- COMPRESS_PERIODIC_CKPT, 203
- COMPRESS_VACATE_CKPT, 203
- CONCURRENCY_LIMIT_DEFAULT, 421
- CONDOR_ADMIN, 154
- CONDOR_CONFIG, 137, 638
- CONDOR_DEVELOPERS_COLLECTOR, 10, 209
- CONDOR_DEVELOPERS, 10, 208
- CONDOR_GAHP, 219, 495
- CONDOR_HOST, 150, 331
- CONDOR_IDS, 120, 153, 316, 613, 642
- CONDOR_JOB_POLL_INTERVAL, 217
- CONDOR_SSHD, 59
- CONDOR_SSH_KEYGEN, 59
- CONDOR_SUPPORT_EMAIL, 154
- CONDOR_VIEW_HOST, 150, 617
- CONSOLE_DEVICES, 125, 181, 393
- CONTINUE, 179, 270
- COUNT_HYPERTHREAD_CPUS, 183
- CPUBusyTime, 273
- CPUBusy, 272
- CPUIdle, 272
- CPUIsBusy, 272
- CREATE_CORE_FILES, 156
- CREDD_CACHE_LOCALLY, 216
- CREDD_HOST, 216, 655, 666
- CRED_CHECK_INTERVAL, 244
- CRED_INDEX_FILE, 244
- CRED_STORE_DIR, 244
- CRED_SUPER_USERS, 244
- C_GAHP_LOG, 219, 495
- C_GAHP_WORKER_THREAD_LOG, 219
- Cluster, 818
- ContinueIdleTime, 272
- DAEMONNAME_ENVIRONMENT, 174
- DAEMON_LIST, 173, 348, 393, 633, 642, 657, 735
- DAEMON_SHUTDOWN_FAST, 164, 638
- DAEMON_SHUTDOWN, 164, 638, 898
- DAGMAN_ABORT_DUPLICATES, 227, 647
- DAGMAN_ABORT_ON_SCARY_SUBMIT, 227, 638, 640
- DAGMAN_ALLOW_EVENTS, 225
- DAGMAN_AUTO_RESCUE, 89, 228, 605
- DAGMAN_CONDOR_RM_EXE, 226
- DAGMAN_CONDOR_SUBMIT_EXE, 226
- DAGMAN_COPY_TO_SPOOL, 228, 587
- DAGMAN_DEBUG_CACHE_ENABLE, 224
- DAGMAN_DEBUG_CACHE_SIZE, 224
- DAGMAN_DEBUG, 226
- DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION, 225
- DAGMAN_INSERT_SUB_FILE, 228, 604, 605
- DAGMAN_LOG_ON_NFS_IS_ERROR, 227
- DAGMAN_MAX_JOBS_IDLE, 225
- DAGMAN_MAX_JOBS_SUBMITTED, 225
- DAGMAN_MAX_RESCUE_NUM, 228, 605
- DAGMAN_MAX_SUBMITS_PER_INTERVAL, 224
- DAGMAN_MAX_SUBMIT_ATTEMPTS, 224
- DAGMAN_MUNGE_NODE_NAMES, 78, 225
- DAGMAN_OLD_RESCUE, 89, 228, 605
- DAGMAN_ON_EXIT_REMOVE, 227, 643
- DAGMAN_PENDING_REPORT_INTERVAL, 228
- DAGMAN_PROHIBIT_MULTI_JOBS, 226
- DAGMAN_RETRY_NODE_FIRST, 224, 227
- DAGMAN_RETRY_SUBMIT_FIRST, 224
- DAGMAN_STARTUP_CYCLE_DETECT, 224
- DAGMAN_STORK_RM_EXE, 226
- DAGMAN_STORK_SUBMIT_EXE, 226
- DAGMAN_SUBMIT_DELAY, 224
- DAGMAN_SUBMIT_DEPTH_FIRST, 227, 645
- DATABASE_PURGE_INTERVAL, 240, 366
- DATABASE_REINDEX_INTERVAL, 241, 367
- DBMSD_ARGS, 242, 634
- DBMSD_LOG, 242, 634

-
- DBMSD_NOT_RESPONDING_TIMEOUT, 242
 - DBMSD, 241, 634
 - DC_DAEMON_LIST, 173, 633
 - DEAD_COLLECTOR_MAX_AVOIDANCE_TIME, 156, 621, 622
 - DEDICATED_EXECUTE_ACCOUNT_REGEX, 169, 319, 420, 639, 641
 - DEDICATED_SCHEDULER_USE_FIFO, 202, 643
 - DEFAULT_CRED_EXPIRE_THRESHOLD, 244
 - DEFAULT_DOMAIN_NAME, 155, 331, 569
 - DEFAULT_IO_BUFFER_BLOCK_SIZE, 206
 - DEFAULT_IO_BUFFER_SIZE, 206
 - DEFAULT_PRIO_FACTOR, 212
 - DEFAULT_RANK_STANDARD, 206
 - DEFAULT_RANK_VANILLA, 206
 - DEFAULT_RANK, 206
 - DEFAULT_UNIVERSE, 205, 589, 800
 - DELEGATE_JOB_GSI_CREDENTIALS, 230, 654
 - DENY_*, 575
 - DENY_CLIENT, 229
 - DEPLOYMENT_RECOMMENDED_DIRS, 141
 - DEPLOYMENT_RECOMMENDED_EXECS, 141
 - DEPLOYMENT_RELEASE_DIR, 141
 - DEPLOYMENT_REQUIRED_DIRS, 141
 - DEPLOYMENT_REQUIRED_EXECS, 141
 - DISCONNECTED_KEYBOARD_IDLE_BOOST, 184, 403
 - D_COMMAND, 306
 - D_SECURITY, 306, 597
 - DedicatedScheduler, 183, 409
 - EMAIL_DOMAIN, 155
 - ENABLE_BACKFILL, 183, 413, 416
 - ENABLE_GRID_MONITOR, 223, 509
 - ENABLE_HISTORY_ROTATION, 154, 663
 - ENABLE_PERSISTENT_CONFIG, 162
 - ENABLE_RUNTIME_CONFIG, 162
 - ENABLE_SOAP_SSL, 243
 - ENABLE_SOAP, 242
 - ENABLE_USERLOG_LOCKING, 158
 - ENABLE_WEB_SERVER, 242
 - ENCRYPT_EXECUTE_DIRECTORY, 230
 - ENV, 149
 - EVENT_LOG_FSYNC, 161, 583
 - EVENT_LOG_JOB_AD_INFORMATION_ATTRS, 45, 161, 634
 - EVENT_LOG_LOCKING, 161, 583
 - EVENT_LOG_MAX_ROTATIONS, 161
 - EVENT_LOG_MAX_SIZE, 161
 - EVENT_LOG_ROTATION_LOCK, 161
 - EVENT_LOG_USE_XML, 161, 634
 - EVENT_LOG, 161, 634
 - EVICT_BACKFILL, 184, 271, 414
 - EXECUTE_LOGIN_IS_DEDICATED, 170, 639, 641, 645
 - EXECUTE, 152, 322, 401, 425, 566, 613, 887
 - EXEC_TRANSFER_ATTEMPTS, 204
 - FILESYSTEM_DOMAIN, 149, 170, 331
 - FILE_LOCK_VIA_MUTEX, 158
 - FLOCK_COLLECTOR_HOSTS, 199, 492
 - FLOCK_FROM, 493
 - FLOCK_NEGOTIATOR_HOSTS, 199, 493
 - FLOCK_TO, 492
 - FS_REMOTE_DIR, 230, 297
 - FULL_HOSTNAME, 147
 - FetchWorkDelay, 193, 461, 465
 - GAHP_ARGS, 218
 - GAHP, 218
 - GLEXEC_JOB, 223, 324, 595, 598
 - GLEXEC_STARTER, 325, 595, 598, 616, 641, 645
 - GLEXEC, 223, 324
 - GLIDEIN_SERVER_URLS, 223, 519
 - GLITE_LOCATION, 219, 511, 512
 - GLOBUS_GATEKEEPER_TIMEOUT, 218
 - GRIDFTP_URL_BASE, 219, 506
 - GRIDMANAGER_CHECKPROXY_INTERVAL, 217
 - GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT, 218
 - GRIDMANAGER_CONTACT_SCHEDD_DELAY, 217
 - GRIDMANAGER_DEBUG, 609
 - GRIDMANAGER_EMPTY_RESOURCE_DELAY, 217
-

-
- GRIDMANAGER_GAHP_CALL_TIMEOUT, 218, 613
 - GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT, 218
 - GRIDMANAGER_JOB_PROBE_INTERVAL, 217
 - GRIDMANAGER_LOG, 217, 631
 - GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE, 218
 - GRIDMANAGER_MAX_PENDING_REQUESTS, 218
 - GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE, 218, 616
 - GRIDMANAGER_MAX_PENDING_SUBMITS, 218
 - GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE, 217, 616
 - GRIDMANAGER_MAX_WS_DESTROYS_PER_RESOURCE, 218, 614
 - GRIDMANAGER_MINIMUM_PROXY_TIME, 217
 - GRIDMANAGER_RESOURCE_PROBE_DELAY, 217
 - GRIDMANAGER_RESOURCE_PROBE_INTERVAL, 217
 - GRIDMAP, 230, 292, 299
 - GRID_MONITOR_HEARTBEAT_TIMEOUT, 223
 - GRID_MONITOR_NO_STATUS_TIMEOUT, 223, 654
 - GRID_MONITOR_RETRY_DURATION, 223
 - GRID_MONITOR, 223, 509
 - GROUP_AUTOREGROUP_<groupname>, 215
 - GROUP_AUTOREGROUP_chemistry, 251
 - GROUP_AUTOREGROUP, 215, 251
 - GROUP_NAMES, 215, 251
 - GROUP_PRIO_FACTOR_<groupname>, 215, 251
 - GROUP_QUOTA_<groupname>, 215
 - GSIDAEEMON.CERT, 229, 291
 - GSIDAEEMON.DIRECTORY, 229, 291, 292
 - GSIDAEEMON.KEY, 229, 291
 - GSIDAEEMON.NAME, 229, 605, 654
 - GSIDAEEMON.PROXY, 230, 291
 - GSIDAEEMON.TRUSTED_CA_DIR, 229, 291
 - GT2_GAHP, 219
 - GT4_GAHP, 219
 - HAD_ARGS, 238
 - HAD_CONNECTION_TIMEOUT, 237
 - HAD_DEBUG, 238
 - HAD_LIST, 237
 - HAD_LOG, 238
 - HAD_UPDATE_INTERVAL, 238
 - HADOBJECT.PRIMARY, 237
 - HAD_USE_REPLICATION, 238, 358
 - HAD, 238
 - HA_<SUBSYS>_LOCK_HOLD_TIME, 237
 - HA_<SUBSYS>_LOCK_URL, 236
 - HA_<SUBSYS>_POLL_PERIOD, 237
 - HA_LOCK_HOLD_TIME, 236
 - HA_LOCK_URL, 236
 - HA_POLL_PERIOD, 237
 - HIBERNATE_CHECK_INTERVAL, 193, 426, 601
 - HIBERNATE, 193, 426, 601
 - HIGHPORT, 166, 327
 - HISTORY, 154
 - HOLD_JOB_IF_CREDENTIAL_EXPIRES, 217
 - HOOK_EVICT_CLAIM, 193, 462
 - HOOK_FETCH_WORK, 193, 461, 462, 464
 - HOOK_JOB_CLEANUP, 468
 - HOOK_JOB_EXIT, 463
 - HOOK_JOB_FINALIZE, 468
 - HOOK_PREPARE_JOB, 462, 881
 - HOOK_REPLY_CLAIM, 193
 - HOOK_REPLY_FETCH, 462
 - HOOK_TRANSLATE, 467
 - HOOK_UPDATE_JOB_INFO, 462, 464, 468, 585
 - HOSTALLOW... , 162
 - HOSTALLOW_*, 575
 - HOSTALLOW_ADMINISTRATOR, 133, 343
 - HOSTALLOW_CONFIG, 343
 - HOSTALLOW_DAEMON, 653
 - HOSTALLOW_NEGOTIATOR_SCHEDD, 493
-

-
- HOSTALLOW_NEGOTIATOR, 343
 - HOSTALLOW_READ, 133, 313
 - HOSTALLOW_WRITE, 119, 133, 137, 520, 668
 - HOSTALLOW, 162
 - HOSTDENY_*, 575
 - HOSTDENY, 162
 - HOSTNAME, 147
 - HighLoad, 272
 - IGNORE_NFS_LOCK_ERRORS, 171
 - INCLUDE, 151
 - INVALID_LOG_FILES, 208, 746
 - IN_HIGHPORT, 166, 327
 - IN_LOWPORT, 166, 327
 - IP_ADDRESS, 147
 - IS_OWNER, 180, 261
 - IS_VALID_CHECKPOINT_PLATFORM, 179, 254
 - JAVA_CLASSPATH_ARGUMENT, 192
 - JAVA_CLASSPATH_DEFAULT, 192
 - JAVA_CLASSPATH_SEPARATOR, 192
 - JAVA_EXTRA_ARGUMENTS, 192, 423
 - JAVA_MAXHEAP_ARGUMENT, 192, 423, 587
 - JAVA, 192, 422, 506
 - JOB_INHERITS_STARTER_ENVIRONMENT, 205, 599, 616, 658
 - JOB_IS_FINISHED_INTERVAL, 197
 - JOB_PROXY_OVERRIDE_FILE, 665
 - JOB_RENICE_INCREMENT, 204, 254
 - JOB_ROUTER_DEFAULTS, 220
 - JOB_ROUTER_ENTRIES_CMD, 220, 528
 - JOB_ROUTER_ENTRIES_FILE, 220
 - JOB_ROUTER_ENTRIES_REFRESH, 220
 - JOB_ROUTER_ENTRIES, 220, 528
 - JOB_ROUTER_MAX_JOBS, 221
 - JOB_ROUTER_NAME, 221
 - JOB_ROUTER_POLLING_PERIOD, 221, 468
 - JOB_ROUTER_SOURCE_JOB_CONSTRAINT, 221
 - JOB_START_COUNT, 196, 197
 - JOB_START_DELAY, 196, 197, 627
 - KEEP_POOL_HISTORY, 210, 396
 - KERBEROS_CLIENT_KEYTAB, 232
 - KERBEROS_MAP_FILE, 294, 299
 - KERBEROS_SERVER_KEYTAB, 232
 - KERBEROS_SERVER_PRINCIPAL, 232, 294
 - KERBEROS_SERVER_SERVICE, 232
 - KERBEROS_SERVER_USER, 232
 - KILLING_TIMEOUT, 268, 271
 - KILL, 179, 271
 - KeyboardBusy, 272
 - LIBEXEC, 151, 201
 - LIB, 151
 - LINUX_HIBERNATION_METHOD, 194
 - LOCAL_CONFIG_DIR, 153
 - LOCAL_CONFIG_FILE, 122, 147, 152, 391–393, 555
 - LOCAL_CREDD, 533
 - LOCAL_DIR, 120, 123, 151, 322
 - LOCAL_QUEUE_BACKUP_DIR, 595
 - LOCAL_XACT_BACKUP_FILTER, 595
 - LOCK_FILE_UPDATE_INTERVAL, 165, 611
 - LOCK, 121, 154
 - LOGS_USE_TIMESTAMP, 158, 633
 - LOG_ON_NFS_IS_ERROR, 207, 633
 - LOG, 152, 156, 182, 353
 - LOWPORT, 166, 327
 - LSF_GAHP, 219, 511
 - LastCkpt, 272
 - LeaseManager.CLASSAD_LOG, 222
 - LeaseManager.DEBUG_ADS, 222
 - LeaseManager.DEFAULT_MAX_LEASE_DURATION, 222
 - LeaseManager.GETADS_INTERVAL, 221
 - LeaseManager.MAX_LEASE_DURATION, 222
 - LeaseManager.MAX_TOTAL_LEASE_DURATION, 222
 - LeaseManager.PRUNE_INTERVAL, 222
 - LeaseManager.QUERY_ADTYPE, 222
 - LeaseManager.QUERY_CONSTRAINTS, 222
 - LeaseManager.UPDATE_INTERVAL, 222
 - MAIL, 154, 392
 - MASTER_<SUBSYS>_CONTROLLER, 237
-

-
- MASTER_<name>_BACKOFF_CEILING, 176, 642
 - MASTER_<name>_BACKOFF_CONSTANT, 175, 642
 - MASTER_<name>_BACKOFF_FACTOR, 176
 - MASTER_<name>_RECOVER_FACTOR, 176
 - MASTER_ADDRESS_FILE, 178
 - MASTER_ATTRS, 178
 - MASTER_BACKOFF_CEILING, 176, 642
 - MASTER_BACKOFF_CONSTANT, 175, 642
 - MASTER_BACKOFF_FACTOR, 176
 - MASTER_CHECK_INTERVAL, 208
 - MASTER_CHECK_NEW_EXEC_INTERVAL, 175
 - MASTER_DEBUG, 178
 - MASTER_HAD_BACKOFF_CONSTANT, 358
 - MASTER_HA_LIST, 236, 355, 633
 - MASTER_INSTANCE_LOCK, 178
 - MASTER_NAME, 177, 735
 - MASTER_NEW_BINARY_DELAY, 175, 642
 - MASTER_RECOVER_FACTOR, 176, 642
 - MASTER_SHUTDOWN_<Name>, 175
 - MASTER_UPDATE_INTERVAL, 175, 642
 - MASTER_WAITS_FOR_GCB_BROKER, 168, 638, 643
 - MATCH_TIMEOUT, 258, 265, 270
 - MAXJOBRETIREMENTTIME, 180, 271
 - MAX_<SUBSYS>_<LEVEL>_LOG, 160
 - MAX_<SUBSYS>_LOG, 157, 160, 665
 - MAX_ACCOUNTANT_DATABASE_SIZE, 212
 - MAX_CKPT_SERVER_LOG, 348
 - MAX_CLAIM_ALIVES_MISSED, 181, 197
 - MAX_CONCURRENT_DOWNLOADS, 196, 622, 627
 - MAX_CONCURRENT_UPLOADS, 196, 622, 627
 - MAX_C_GAHP_LOG, 219
 - MAX_DAGMAN_LOG, 71, 226
 - MAX_DISCARDED_RUN_TIME, 172, 346
 - MAX_EVENT_LOG, 161, 596, 634
 - MAX_HAD_LOG, 238
 - MAX_HISTORY_LOG, 155
 - MAX_HISTORY_ROTATIONS, 155, 663
 - MAX_JOBS_RUNNING, 39, 195, 328, 893
 - MAX_JOBS_SUBMITTED, 196
 - MAX_JOB_MIRROR_UPDATE_LAG, 221
 - MAX_JOB_QUEUE_LOG_ROTATIONS, 155
 - MAX_NEXT_JOB_START_DELAY, 197, 627, 806, 883
 - MAX_NUM_CPUS, 182
 - MAX_PENDING_STARTD_CONTACTS, 196, 598
 - MAX_REPLICATION_LOG, 239
 - MAX_SCHEDD_LOG, 315
 - MAX_SHADOW_EXCEPTIONS, 196
 - MAX_SLOT_TYPES, 185
 - MAX_TRACKING_GID, 420, 628
 - MAX_TRANSFERER_LIFETIME, 238
 - MAX_TRANSFERER_LOG, 239
 - MAX_VM_GAHP_LOG, 233
 - MEMORY, 183
 - MIN_TRACKING_GID, 420, 628
 - MPI_CONDOR_RSH_PATH, 201
 - MYPROXY_GET_DELEGATION, 242, 508
 - MachineBusy, 272
 - MaxSuspendTime, 272
 - MaxVacateTime, 272
 - NEGOTIATE_ALL_JOBS_IN_CLUSTER, 199, 248
 - NEGOTIATOR_ADDRESS_FILE, 326
 - NEGOTIATOR_CONSIDER_PREEMPTION, 215, 280, 608
 - NEGOTIATOR_CYCLE_DELAY, 211
 - NEGOTIATOR_DEBUG, 214
 - NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES, 212
 - NEGOTIATOR_HOST, 150
 - NEGOTIATOR_IGNORE_USER_PRIORITIES, 517
 - NEGOTIATOR_INFORM_STARTD, 212, 627
 - NEGOTIATOR_INTERVAL, 211
 - NEGOTIATOR_MATCHLIST_CACHING, 517, 665
 - NEGOTIATOR_MATCH_EXPRS, 214, 639, 640
 - NEGOTIATOR_MAX_TIME_PER_PIESPIN, 214
-

-
- NEGOTIATOR_MAX_TIME_PER_SUBMITTER, 214
 - NEGOTIATOR_POST_JOB_RANK, 213
 - NEGOTIATOR_PRE_JOB_RANK, 213
 - NEGOTIATOR_SOCKET_CACHE_SIZE, 212, 328
 - NEGOTIATOR_TIMEOUT, 212
 - NEGOTIATOR_USE_NONBLOCKING_STARTD_CONFIG, 167
 - NETWORK_INTERFACE, 165, 330, 332
 - NETWORK_MAX_PENDING_CONNECTS, 157
 - NET_REMAP_ENABLE, 167
 - NET_REMAP_INAGENT, 167, 643
 - NET_REMAP_ROUTE, 168
 - NET_REMAP_SERVICE, 167
 - NICE_USER_PRIO_FACTOR, 212, 245
 - NODE, 56, 61
 - NONBLOCKING_COLLECTOR_UPDATE, 167
 - NORDUGRID_GAHP, 220
 - NOT_RESPONDING_TIMEOUT, 164
 - NOT_RESPONDING_WANT_CORE, 165, 589
 - NO_DNS, 155
 - NUM_CPUS, 182, 186, 402
 - NUM_SLOTS_TYPE_<N>, 186, 402
 - NUM_SLOTS, 186, 402
 - Node, 818
 - NonCondorLoadAvg, 272
 - OBITUARY_LOG_LENGTH, 174
 - OFFLINE_EXPIRE_ADS_AFTER, 194, 428, 584
 - OFFLINE_LOG, 194, 428, 584
 - OPSYS, 148, 392
 - OUT_HIGHPORT, 167, 327
 - OUT_LOWPORT, 166, 327
 - PASSWD_CACHE_REFRESH, 156
 - PBS_GAHP, 219, 511
 - PERIODIC_CHECKPOINT, 179, 443, 555
 - PERIODIC_EXPR_INTERVAL, 200, 637
 - PERIODIC_EXPR_TIMESLICE, 200, 640
 - PERIODIC_MEMORY_SYNC, 203
 - PERSISTENT_CONFIG_DIR, 162
 - PER_JOB_HISTORY_DIR, 202, 647
 - PID, 148
 - POLLING_INTERVAL, 180, 266
 - POOL_HISTORY_DIR, 210, 396
 - POOL_HISTORY_MAX_STORAGE, 210, 396
 - POOL_HISTORY_SAMPLING_INTERVAL, 210
 - PPID, 149
 - PREEMPTION_RANK_STABLE, 214, 246, 584, 618
 - PREEMPTION_RANK, 214, 618
 - PREEMPTION_REQUIREMENTS_STABLE, 213, 246, 584, 618
 - PREEMPTION_REQUIREMENTS, 47, 213, 215, 246, 618, 753
 - PREEMPT, 179, 270, 463
 - PREEN_ADMIN, 207, 746
 - PREEN_ARGS, 174
 - PREEN_INTERVAL, 174, 641, 642, 644
 - PREEN, 174
 - PRIORITY_HALFLIFE, 47, 212, 245, 247
 - PRIVATE_NETWORK_INTERFACE, 166, 634
 - PRIVATE_NETWORK_NAME, 165, 634
 - PRIVSEP_ENABLED, 232, 323, 628
 - PRIVSEP_SWITCHBOARD, 232, 323, 628
 - PROCD_ADDRESS, 216, 611
 - PROCD_LOG, 216
 - PROCD_MAX_SNAPSHOT_INTERVAL, 216
 - PUBLISH_OBITUARIES, 174
 - ParallelSchedulingGroup, 202, 411, 412
 - Process, 818
 - QUERY_TIMEOUT, 208
 - QUEUE_ALL_USERS_TRUSTED, 198
 - QUEUE_CLEAN_INTERVAL, 198
 - QUEUE_SUPER_USERS, 198
 - QUILL_ADDRESS_FILE, 241, 367
 - QUILL_ARGS, 239
 - QUILL_DB_SIZE_LIMIT, 241, 367
 - QUILL_DB_IP_ADDR, 240, 364, 366
 - QUILL_DB_NAME, 240, 366
 - QUILL_DB_QUERY_PASSWORD, 241, 367
 - QUILL_DB_TYPE, 240
 - QUILL_DB_USER, 240, 366
 - QUILL_ENABLED, 240, 893
-

-
- QUILL_IS_REMOTELY_QUERYABLE, 241, 367
 - QUILL_JOB_HISTORY_DURATION, 241, 366
 - QUILL_LOG, 240
 - QUILL_MAINTAIN_DB_CONN, 240, 367
 - QUILL_MANAGE_VACUUM, 241, 367
 - QUILL_NAME, 240, 366
 - QUILL_NOT_RESPONDING_TIMEOUT, 240
 - QUILL_POLLING_PERIOD, 240, 366
 - QUILL_RESOURCE_HISTORY_DURATION, 241, 366
 - QUILL_RUN_HISTORY_DURATION, 241, 366
 - QUILL_SHOULD_REINDEX, 241, 656
 - QUILL_USE_SQL_LOG, 240
 - QUILL, 239
 - Q-QUERY_TIMEOUT, 156
 - RANDOM_CHOICE (), 149
 - RANDOM_INTEGER (), 149, 555
 - RANK_FACTOR, 411
 - RANK, 179, 255, 271, 410, 411
 - RELEASE_DIR, 122, 151, 392
 - REMOTE_PRIO_FACTOR, 212, 245
 - REPLICATION_ARGS, 239
 - REPLICATION_DEBUG, 239
 - REPLICATION_INTERVAL, 238
 - REPLICATION_LIST, 238
 - REPLICATION_LOG, 239
 - REPLICATION, 239
 - REQUEST_CLAIM_TIMEOUT, 197
 - REQUIRE_LOCAL_CONFIG_FILE, 153
 - RESERVED_DISK, 154, 887
 - RESERVED_MEMORY, 183
 - RESERVED_SWAP, 154, 558
 - RESERVE_AFS_CACHE, 170
 - RUNBENCHMARKS, 183, 264, 270
 - Requirements, 195
 - SBIN, 151
 - SCHEDD_ADDRESS_FILE, 199
 - SCHEDD_ASSUME_NEGOTIATOR_GONE, 201
 - SCHEDD_ATTRS, 199
 - SCHEDD_BACKUP_SPOOL, 201
 - SCHEDD_CLUSTER_INCREMENT_VALUE, 202, 581
 - SCHEDD_CLUSTER_INITIAL_VALUE, 202, 581
 - SCHEDD_DAEMON_AD_FILE, 163, 633
 - SCHEDD_DEBUG, 199
 - SCHEDD_EXECUTE, 199
 - SCHEDD_HOST, 151
 - SCHEDD_INTERVAL_TIMESLICE, 196, 640
 - SCHEDD_INTERVAL, 101, 196
 - SCHEDD_LOCK, 198
 - SCHEDD_LOG, 558
 - SCHEDD_MIN_INTERVAL, 196
 - SCHEDD_NAME, 177, 199, 356, 610
 - SCHEDD_PREEMPTION_RANK, 202, 412
 - SCHEDD_PREEMPTION_REQUIREMENTS, 201, 411
 - SCHEDD_QUERY_WORKERS, 196
 - SCHEDD_ROUND_ATTR_<xxxx>, 201
 - SCHEDD_ROUND_ATTR, 662
 - SCHEDD_SEND_VACATE_VIA_TCP, 202
 - SCHED_UNIV_RENICE_INCREMENT, 198
 - SECONDARY_COLLECTOR_LIST, 178, 637, 640
 - SEC_*_AUTHENTICATION_METHODS, 229
 - SEC_*_AUTHENTICATION, 229
 - SEC_*_CRYPTO_METHODS, 229
 - SEC_*_ENCRYPTION, 229
 - SEC_*_INTEGRITY, 229
 - SEC_*_NEGOTIATION, 229
 - SEC_CLAIMTOBE_INCLUDE_DOMAIN, 668
 - SEC_DEFAULT_AUTHENTICATION_METHODS, 345
 - SEC_DEFAULT_AUTHENTICATION_TIMEOUT, 230, 602
 - SEC_DEFAULT_SESSION_DURATION, 230, 611
 - SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION, 231, 598
 - SEC_INVALIDATE_SESSIONS_VIA_TCP, 230, 598
 - SEC_PASSWORD_FILE, 231, 295
-

-
- SEC_TCP_SESSION_DEADLINE, 230, 581
 - SEC_TCP_SESSION_TIMEOUT, 230
 - SETTABLE_ATTRS..., 162
 - SETTABLE_ATTRS_PERMISSION-LEVEL, 314
 - SETTABLE_ATTRS, 162, 314
 - SHADOW_DEBUG, 202
 - SHADOW_JOB_CLEANUP_RETRY_DELAY, 203
 - SHADOW_LAZY_QUEUE_UPDATE, 203, 627
 - SHADOW_LOCK, 202
 - SHADOW_LOG, 42
 - SHADOW_MAX_JOB_CLEANUP_RETRIES, 203
 - SHADOW_QUEUE_UPDATE_INTERVAL, 202
 - SHADOW_RENICE_INCREMENT, 198
 - SHADOW_SIZE_ESTIMATE, 197
 - SHADOW, 195
 - SHELL, 779
 - SHUTDOWN_FAST_TIMEOUT, 175, 642
 - SHUTDOWN_GRACEFUL_TIMEOUT, 162, 642
 - SIGNIFICANT_ATTRIBUTES, 248
 - SLOTN_JOB_HOOK_KEYWORD, 193, 464
 - SLOTS_CONNECTED_TO_CONSOLE, 184, 403
 - SLOTS_CONNECTED_TO_KEYBOARD, 184, 403
 - SLOT_TYPE_<N>_PARTITIONABLE, 185, 407
 - SLOT_TYPE_<N>, 185, 400
 - SLOTx_EXECUTE, 152, 401
 - SLOTx_USER, 169, 319
 - SLOW_CKPT_SPEED, 203
 - SOAP_LEAVE_IN_QUEUE, 242, 471
 - SOAP_SSL_CA_DIR, 243
 - SOAP_SSL_CA_FILE, 243
 - SOAP_SSL_DH_FILE, 243
 - SOAP_SSL_SERVER_KEYFILE_PASSWORD, 243
 - SOAP_SSL_SERVER_KEYFILE, 243
 - SOFT_UID_DOMAIN, 169, 316
 - SPOOL, 152, 595
 - STARTD_ADDRESS_FILE, 182
 - STARTD_AD_REEVAL_EXPR, 609
 - STARTD_ATTRS, 163, 181, 315, 350, 406, 412
 - STARTD_AVAIL_CONFIDENCE, 192
 - STARTD_CLAIM_ID_FILE, 182
 - STARTD_COMPUTE_AVAIL_STATS, 191
 - STARTD_CRON_<ModuleName>_ARGS, 189
 - STARTD_CRON_<ModuleName>_CWD, 189
 - STARTD_CRON_<ModuleName>_ENV, 189
 - STARTD_CRON_<ModuleName>_EXECUTABLE, 187
 - STARTD_CRON_<ModuleName>_KILL, 188
 - STARTD_CRON_<ModuleName>_MODE, 188
 - STARTD_CRON_<ModuleName>_OPTIONS, 189
 - STARTD_CRON_<ModuleName>_PERIOD, 188
 - STARTD_CRON_<ModuleName>_PREFIX, 187
 - STARTD_CRON_<ModuleName>_RECONFIG, 188
 - STARTD_CRON_AUTOPUBLISH, 187
 - STARTD_CRON_CONFIG_VAL, 186
 - STARTD_CRON_JOBLIST, 187
 - STARTD_CRON_JOBS, 190
 - STARTD_CRON_NAME, 186
 - STARTD_DEBUG, 181, 315, 644
 - STARTD_EXPRS, 163
 - STARTD_HAS_BAD_UTMP, 181
 - STARTD_JOB_EXPRS, 181, 213
 - STARTD_JOB_HOOK_KEYWORD, 193, 464
 - STARTD_MAX_AVAIL_PERIOD_SAMPLES, 192
 - STARTD_NAME, 183
 - STARTD_NOCLAIM_SHUTDOWN, 183
 - STARTD_RESOURCE_PREFIX, 184, 629, 636
 - STARTD_SENDS_ALIVES, 182
 - STARTD_SHOULD_WRITE_CLAIM_ID_FILE, 182
-

-
- STARTD_SLOT_ATTRS, 185
 - STARTD_VM_ATTRS, 185
 - STARTD_VM_EXPRS, 185
 - STARTER_ALLOW_RUNAS_OWNER, 169, 318, 419, 639, 641
 - STARTER_CHOOSES_CKPT_SERVER, 172, 349
 - STARTER_DEBUG, 204
 - STARTER_INITIAL_UPDATE_INTERVAL, 463
 - STARTER_JOB_ENVIRONMENT, 205
 - STARTER_JOB_HOOK_KEYWORD, 464
 - STARTER_LOCAL_LOGGING, 204
 - STARTER_LOCAL, 195
 - STARTER_UPDATE_INTERVAL_TIMESLICE, 204, 602
 - STARTER_UPDATE_INTERVAL, 204, 463, 627
 - STARTER_UPLOAD_TIMEOUT, 205, 618, 638, 640
 - STARTER, 180
 - START_BACKFILL, 184, 265, 271, 413, 416
 - START_DAEMONS, 175
 - START_LOCAL_UNIVERSE, 195, 637, 640, 893
 - START_MASTER, 175
 - START_SCHEDULER_UNIVERSE, 195, 637, 640, 893
 - START, 179, 184, 253, 270, 410
 - STATE_FILE, 238
 - STORK_MAXDELAY_INMINUTES, 243
 - STORK_MAX_NUM_JOBS, 243
 - STORK_MAX_RETRY, 104, 243
 - STORK_MODULE_DIR, 243
 - STORK_TMP_CRED_DIR, 243
 - SUBMIT_DEBUG, 160
 - SUBMIT_EXPRS, 207
 - SUBMIT_MAX_PROCS_IN_CLUSTER, 207
 - SUBMIT_SEND_RESCHEDULE, 207
 - SUBMIT_SKIP_FILECHECKS, 207
 - SUBSYSTEM, 147
 - SUSPEND, 179, 270, 561
 - SYSAPI_GET_LOADAVG, 157
 - SYSTEM_PERIODIC_HOLD, 200, 620
 - SYSTEM_PERIODIC_RELEASE, 200, 620
 - SYSTEM_PERIODIC_REMOVE, 200, 620
 - StartIdleTime, 272
 - StateTimer, 272
 - TCP_UPDATE_COLLECTORS, 167
 - TILDE, 147
 - TOOL_DEBUG, 160
 - TOUCH_LOG_INTERVAL, 158
 - TRANSFERER_DEBUG, 239
 - TRANSFERER_LOG, 239
 - TRANSFERER, 239, 583
 - TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN, 161
 - TRUNC_<SUBSYS>_LOG_ON_OPEN, 158, 161
 - TRUST_UID_DOMAIN, 169
 - UID_DOMAIN, 149, 168, 316, 317, 331, 799
 - UNAME_ARCH, 148
 - UNAME_OPSYS, 148
 - UNICORE_GAHP, 220
 - UPDATE_COLLECTOR_WITH_TCP, 167, 209, 345
 - UPDATE_INTERVAL, 180, 187, 263
 - USERNAME, 149
 - USER_JOB_WRAPPER, 204, 569
 - USE_AFS, 171
 - USE_CKPT_SERVER, 172, 348, 349
 - USE_CLONE_TO_CREATE_PROCESSES, 164, 631, 637, 640
 - USE_GID_PROCESS_TRACKING, 420, 628
 - USE_NFS, 171
 - USE_PROCD, 216, 232, 323, 420, 639, 641
 - USE_PROCESS_GROUPS, 178, 645
 - USE_VISIBLE_DESKTOP, 205, 535, 595, 596
 - VALID_COD_USERS, 448
 - VALID_SPOOL_FILES, 207, 236, 355, 601, 612, 746
 - VMP_HOST_MACHINE, 235, 398
 - VMP_VM_LIST, 235, 398
 - VMWARE_BRIDGE_NETWORKING_TYPE, 234
 - VMWARE_LOCAL_SETTINGS_FILE, 235
 - VMWARE_NAT_NETWORKING_TYPE, 234
 - VMWARE_NETWORKING_TYPE, 234
-

- VMWARE_PERL, 234
- VMWARE_SCRIPT, 234
- VM_GAHP_LOG, 233, 592
- VM_GAHP_REQ_TIMEOUT, 233
- VM_GAHP_SERVER, 233
- VM_MAX_NUMBER, 233
- VM_MEMORY, 233
- VM_NETWORKING_DEFAULT_TYPE, 234
- VM_NETWORKING_TYPE, 234
- VM_NETWORKING, 234
- VM_RECHECK_INTERVAL, 233
- VM_SOFT_SUSPEND, 233
- VM_STATUS_INTERVAL, 233
- VM_TYPE, 233
- VM_UNIV_NOBODY_USER, 233
- VM_VERSION, 233
- WALL_CLOCK_CKPT_INTERVAL, 198
- WANT_SUSPEND, 180, 270
- WANT_UDP_COMMAND_SOCKET, 157, 212, 627
- WANT_VACATE, 180, 271
- WARN_ON_UNUSED_SUBMIT_FILE_MACROS, 207, 627, 820
- WEB_ROOT_DIR, 242
- WINDOWS_FIREWALL_FAILURE_RETRY, 178
- WorkHours, 277
- XEN_BOOTLOADER, 235
- XEN_BRIDGE_SCRIPT, 235
- XEN_DEFAULT_INITRD, 235, 425
- XEN_DEFAULT_KERNEL, 235, 425
- XEN_LOCAL_SETTINGS_FILE, 235
- XEN_SCRIPT, 235
- <https://ec2.amazonaws.com/>, 513
- ALLOW_ADMINISTRATOR, 302
- ALLOW_ADVERTISE_MASTER, 303
- ALLOW_ADVERTISE_SCHEDD, 303
- ALLOW_ADVERTISE_STARTD, 303
- ALLOW_CLIENT, 303
- ALLOW_CONFIG, 302
- ALLOW_DAEMON, 302
- ALLOW_NEGOTIATOR, 302
- ALLOW_OWNER, 302
- ALLOW_READ, 302
- ALLOW_SOAP, 302
- ALLOW_WRITE, 302
- COLLECTOR_ADDRESS_FILE, 163
- DENY_ADMINISTRATOR, 302
- DENY_ADVERTISE_MASTER, 303
- DENY_ADVERTISE_SCHEDD, 303
- DENY_ADVERTISE_STARTD, 303
- DENY_CLIENT, 303
- DENY_CONFIG, 302
- DENY_DAEMON, 302
- DENY_NEGOTIATOR, 302
- DENY_OWNER, 302
- DENY_READ, 302
- DENY_SOAP, 302
- DENY_WRITE, 302
- MAXJOBRETIREMENTTIME, 279
- NEGOTIATOR_ADDRESS_FILE, 163
- SEC_ADMINISTRATOR_AUTHENTICATION_METHODS, 288
- SEC_ADMINISTRATOR_AUTHENTICATION, 288
- SEC_ADMINISTRATOR_CRYPTOMETHODS, 300
- SEC_ADMINISTRATOR_ENCRYPTION, 299
- SEC_ADMINISTRATOR_INTEGRITY, 301
- SEC_ADVERTISE_MASTER_AUTHENTICATION_METHODS, 288
- SEC_ADVERTISE_MASTER_AUTHENTICATION, 288
- SEC_ADVERTISE_MASTER_CRYPTOMETHODS, 300
- SEC_ADVERTISE_MASTER_ENCRYPTION, 299
- SEC_ADVERTISE_MASTER_INTEGRITY, 301
- SEC_ADVERTISE_SCHEDD_AUTHENTICATION_METHODS, 288
- SEC_ADVERTISE_SCHEDD_AUTHENTICATION, 288
- SEC_ADVERTISE_SCHEDD_CRYPTOMETHODS, 300
- SEC_ADVERTISE_SCHEDD_ENCRYPTION, 299
- SEC_ADVERTISE_SCHEDD_INTEGRITY, 301

-
- SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS, 288
 - SEC_ADVERTISE_STARTD_AUTHENTICATION_SECTION, 288
 - SEC_ADVERTISE_STARTD_CRYPTOMETHODS, 300
 - SEC_ADVERTISE_STARTD_ENCRYPTION, 299
 - SEC_ADVERTISE_STARTD_INTEGRITY, 301
 - SEC_CLIENT_AUTHENTICATION_METHODS, 288
 - SEC_CLIENT_AUTHENTICATION, 287
 - SEC_CLIENT_CRYPTOMETHODS, 300
 - SEC_CLIENT_ENCRYPTION, 299
 - SEC_CLIENT_INTEGRITY, 301
 - SEC_CONFIG_AUTHENTICATION_METHODS, 288
 - SEC_CONFIG_AUTHENTICATION, 288
 - SEC_CONFIG_CRYPTOMETHODS, 300
 - SEC_CONFIG_ENCRYPTION, 299
 - SEC_CONFIG_INTEGRITY, 301
 - SEC_DAEMON_AUTHENTICATION_METHODS, 288
 - SEC_DAEMON_AUTHENTICATION, 288
 - SEC_DAEMON_CRYPTOMETHODS, 300
 - SEC_DAEMON_ENCRYPTION, 299
 - SEC_DAEMON_INTEGRITY, 301
 - SEC_DEFAULT_AUTHENTICATION_METHODS, 288
 - SEC_DEFAULT_AUTHENTICATION, 287, 288
 - SEC_DEFAULT_CRYPTOMETHODS, 300
 - SEC_DEFAULT_ENCRYPTION, 299
 - SEC_DEFAULT_INTEGRITY, 301
 - SEC_NEGOTIATOR_AUTHENTICATION_METHODS, 288
 - SEC_NEGOTIATOR_AUTHENTICATION, 288
 - SEC_NEGOTIATOR_CRYPTOMETHODS, 300
 - SEC_NEGOTIATOR_INTEGRITY, 301
 - SEC_OWNER_AUTHENTICATION_METHODS, 288
 - SEC_OWNER_AUTHENTICATION, 288
 - SEC_OWNER_CRYPTOMETHODS, 300
 - SEC_OWNER_ENCRYPTION, 299
 - SEC_OWNER_INTEGRITY, 301
 - SEC_READ_AUTHENTICATION_METHODS, 288
 - SEC_READ_AUTHENTICATION, 288
 - SEC_READ_CRYPTOMETHODS, 300
 - SEC_READ_ENCRYPTION, 299
 - SEC_READ_INTEGRITY, 301
 - SEC_WRITE_AUTHENTICATION_METHODS, 288
 - SEC_WRITE_AUTHENTICATION, 288
 - SEC_WRITE_CRYPTOMETHODS, 300
 - SEC_WRITE_ENCRYPTION, 299
 - SEC_WRITE_INTEGRITY, 301
 - Console activity, 131
 - CONSOLE_DEVICES macro, 125, 181, 393
 - CONTINUE macro, 179, 270
 - ContinueIdleTime macro, 272
 - contrib module
 - CondorView client, 138
 - COUNT_HYPERTHREAD_CPUS macro, 183
 - CPU activity, 131
 - CPUBusy macro, 272
 - CPUBusyTime macro, 273
 - CPUIdle macro, 272
 - CPUIsBusy macro, 272
 - crashes, 575
 - CREATE_CORE_FILES macro, 156
 - CRED_CHECK_INTERVAL macro, 244
 - CRED_INDEX_FILE macro, 244
 - CRED_STORE_DIR macro, 244
 - CRED_SUPER_USERS macro, 244
 - CREDD_CACHE_LOCALLY macro, 216
 - CREDD_HOST macro, 216, 655, 666
 - CronTab job scheduling, 98
 - crond program, 140
 - CumulativeSuspensionTime
 - job ClassAd attribute, 879
 - current working directory, 320
 - CurrentHosts
 - job ClassAd attribute, 879
 - cwd
 - of jobs, 320
 - D_COMMAND macro, 306
 - D_SECURITY macro, 306, 597
-

- daemon
 - condor_had*, 357
 - condor_ckpt_server*, 115
 - condor_collector*, 115
 - condor_credd*, 116, 216, 533
 - condor_dbmsd*, 115
 - condor_gridmanager*, 115
 - condor_had*, 116
 - condor_job_router*, 116, 521
 - condor_kbdd*, 115
 - condor_lease_manager*, 106, 116
 - condor_master*, 114
 - condor_negotiator*, 115
 - condor_procd*, 116
 - condor_quill*, 115
 - condor_replication*, 116
 - condor_schedd*, 114
 - condor_shadow*, 114
 - condor_startd*, 114, 252, 253
 - condor_starter*, 114
 - running as root, 109
 - stork_server*, 116
- DAEMON_LIST macro, 173, 348, 393, 633, 642, 657, 735
- DAEMON_SHUTDOWN macro, 164, 638, 898
- DAEMON_SHUTDOWN_FAST macro, 164, 638
- daemoncore, 351–354
 - command line arguments, 352
 - Unix signals, 352
- DAEMONNAME_ENVIRONMENT macro, 174
- daemons
 - descriptions, 114
- DAGMan, 64–91
 - ABORT-DAG-ON, 72
 - CONFIG, 76
 - DAG input file, 65
 - DAGs within DAGs, 78
 - describing dependencies, 69
 - dot, 90
 - example submit description file, 70
 - File Paths in DAGs, 89
 - job submission, 70
 - POST script, 67
 - PRE and POST scripts, 67
 - PRE script, 67
 - Rescue DAG, 86
 - RETRY of failed nodes, 72
 - Single submission of multiple, independent DAGs, 77
 - Splicing DAGs, 80
 - submit description file with, 69
 - VARs (macro for submit description file), 73
 - visualizing DAGs, 90
- DAGMan input file
 - ABORT-DAG-ON key word, 72
 - CATEGORY key word, 76
 - CONFIG key word, 76
 - DATA key word, 67
 - JOB key word, 66
 - MAXJOBS key word, 76
 - PARENT . . .CHILD key word, 69
 - PRIORITY key word, 75
 - RETRY key word, 72
 - SCRIPT key word, 67
 - SPLICE key word, 80
 - SUBDAG key word, 78
 - VARs key word, 73
- DAGMAN_ABORT_DUPLICATES macro, 227, 647
- DAGMAN_ABORT_ON_SCARY_SUBMIT macro, 227, 638, 640
- DAGMAN_ALLOW_EVENTS macro, 225
- DAGMAN_AUTO_RESCUE macro, 89, 228, 605
- DAGMAN_CONDOR_RM_EXE macro, 226
- DAGMAN_CONDOR_SUBMIT_EXE macro, 226
- DAGMAN_COPY_TO_SPOOL macro, 228, 587
- DAGMAN_DEBUG macro, 226
- DAGMAN_DEBUG_CACHE_ENABLE macro, 224
- DAGMAN_DEBUG_CACHE_SIZE macro, 224
- DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION macro, 225
- DAGMAN_INSERT_SUB_FILE macro, 228, 604, 605
- DAGMAN_LOG_ON_NFS_IS_ERROR macro, 227
- DAGMAN_MAX_JOBS_IDLE macro, 225
- DAGMAN_MAX_JOBS_SUBMITTED macro, 225
- DAGMAN_MAX_RESCUE_NUM macro, 228, 605
- DAGMAN_MAX_SUBMIT_ATTEMPTS macro, 224
- DAGMAN_MAX_SUBMITS_PER_INTERVAL macro, 224

-
- DAGMAN_MUNGE_NODE_NAMES macro, 78, 225
 - DAGMAN_OLD_RESCUE macro, 89, 228, 605
 - DAGMAN_ON_EXIT_REMOVE macro, 227, 643
 - DAGMAN_PENDING_REPORT_INTERVAL macro, 228
 - DAGMAN_PROHIBIT_MULTI_JOBS macro, 226
 - DAGMAN_RETRY_NODE_FIRST macro, 224, 227
 - DAGMAN_RETRY_SUBMIT_FIRST macro, 224
 - DAGMAN_STARTUP_CYCLE_DETECT macro, 224
 - DAGMAN_STORK_RM_EXE macro, 226
 - DAGMAN_STORK_SUBMIT_EXE macro, 226
 - DAGMAN_SUBMIT_DELAY macro, 224
 - DAGMAN_SUBMIT_DEPTH_FIRST macro, 227, 645
 - DATABASE_PURGE_INTERVAL macro, 240, 366
 - DATABASE_REINDEX_INTERVAL macro, 241, 367
 - DBMSD macro, 241, 634
 - DBMSD_ARGS macro, 242, 634
 - DBMSD_LOG macro, 242, 634
 - DBMSD_NOT_RESPONDING_TIMEOUT macro, 242
 - DC_DAEMON_LIST macro, 173, 633
 - DEAD_COLLECTOR_MAX_AVOIDANCE_TIME macro, 156, 621, 622
 - dedicated scheduling, 408
 - DEDICATED_EXECUTE_ACCOUNT_REGEX macro, 169, 319, 420, 639, 641
 - DEDICATED_SCHEDULER_USE_FIFO macro, 202, 643
 - DedicatedScheduler macro, 183, 409
 - DEFAULT_CRED_EXPIRE_THRESHOLD macro, 244
 - DEFAULT_DOMAIN_NAME macro, 155, 331, 569
 - DEFAULT_IO_BUFFER_BLOCK_SIZE macro, 206
 - DEFAULT_IO_BUFFER_SIZE macro, 206
 - DEFAULT_PRIO_FACTOR macro, 212
 - DEFAULT_RANK macro, 206
 - DEFAULT_RANK_STANDARD macro, 206
 - DEFAULT_RANK_VANILLA macro, 206
 - DEFAULT_UNIVERSE macro, 205, 589, 800
 - deferral time
 - of a job, 95
 - DELEGATE_JOB_GSI_CREDENTIALS macro, 230, 654
 - DENY_* macro, 575
 - DENY_ADMINISTRATOR macro, 302
 - DENY_ADVERTISE_MASTER macro, 303
 - DENY_ADVERTISE_SCHEDD macro, 303
 - DENY_ADVERTISE_STARTD macro, 303
 - DENY_CLIENT macro, 229
 - DENY_CLIENT macro, 303
 - DENY_CONFIG macro, 302
 - DENY_DAEMON macro, 302
 - DENY_NEGOTIATOR macro, 302
 - DENY_OWNER macro, 302
 - DENY_READ macro, 302
 - DENY_SOAP macro, 302
 - DENY_WRITE macro, 302
 - Deployment commands
 - cleanup_release, 673
 - condor_cold_start, 689
 - condor_cold_stop, 692
 - filelock_midwife, 852
 - filelock_undertaker, 854
 - install_release, 856
 - uniq_pid_midwife, 874
 - uniq_pid_undertaker, 876
 - deployment commands, 140
 - DEPLOYMENT_RECOMMENDED_DIRS macro, 141
 - DEPLOYMENT_RECOMMENDED_EXECS macro, 141
 - DEPLOYMENT_RELEASE_DIR macro, 141
 - DEPLOYMENT_REQUIRED_DIRS macro, 141
 - DEPLOYMENT_REQUIRED_EXECS macro, 141
 - directed acyclic graph (DAG), 64
 - Directed Acyclic Graph Manager (DAGMan), 64
 - DISCONNECTED_KEYBOARD_IDLE_BOOST macro, 184, 403
 - disk space requirement
 - execute directory, 120
 - log directory, 120
 - spool directory, 120
 - all versions, 123
-

- Condor files, 121
- DiskUsage
 - job ClassAd attribute, 879
- distributed ownership
 - of machines, 2
- Distributed Resource Management Application
 - API (DRMAA), 480
- dot, 90
- download, 117
- DRMAA (Distributed Resource Management Application API), 480
- dynamic *condor_startd* provisioning, 407
- dynamic deployment, 140
 - configuration, 141
 - relevance to grid computing, 520
- effective user priority (EUP), 245
- EMAIL_DOMAIN macro, 155
- EmailAttributes
 - job ClassAd attribute, 879
- ENABLE_BACKFILL macro, 183, 413, 416
- ENABLE_GRID_MONITOR macro, 223, 509
- ENABLE_HISTORY_ROTATION macro, 154, 663
- ENABLE_PERSISTENT_CONFIG macro, 162
- ENABLE_RUNTIME_CONFIG macro, 162
- ENABLE_SOAP macro, 242
- ENABLE_SOAP_SSL macro, 243
- ENABLE_USERLOG_LOCKING macro, 158
- ENABLE_WEB_SERVER macro, 242
- ENCRYPT_EXECUTE_DIRECTORY macro, 230
- EnteredCurrentStatus
 - job ClassAd attribute, 879
- ENV macro, 149
- environment variables, 33
 - _CONDOR_SCRATCH_DIR, 34
 - _CONDOR_SLOT, 34
 - CONDOR_CONFIG, 574
 - CONDOR_IDS, 120, 154
 - CONDOR_VM, 34
 - copying current environment, 798
 - in submit description file, 819
 - setting, for a job, 797
 - X509_USER_PROXY, 34
- EVENT_LOG macro, 161, 634
- EVENT_LOG_FSYNC macro, 161, 583
- EVENT_LOG_JOB_AD_INFORMATION_ATTRS
 - macro, 45, 161, 634
- EVENT_LOG_LOCKING macro, 161, 583
- EVENT_LOG_MAX_ROTATIONS macro, 161
- EVENT_LOG_MAX_SIZE macro, 161
- EVENT_LOG_ROTATION_LOCK macro, 161
- EVENT_LOG_USE_XML macro, 161, 634
- EVICT_BACKFILL macro, 184, 271, 414
- EXEC_TRANSFER_ATTEMPTS macro, 204
- ExecutableSize
 - job ClassAd attribute, 880
- execute machine, 113
- EXECUTE macro, 152, 322, 401, 425, 566, 613, 887
- EXECUTE_LOGIN_IS_DEDICATED macro, 170, 639, 641, 645
- execution environment, 33
- ExitBySignal
 - job ClassAd attribute, 880
- ExitCode
 - job ClassAd attribute, 880
- ExitSignal
 - job ClassAd attribute, 880
- ExitStatus
 - job ClassAd attribute, 880
- FAQ, 547–578
 - Condor on Windows machines, 566
 - installing Condor, 547
- FetchWorkDelay macro, 193, 461, 465
- file
 - locking, 4, 17
 - memory-mapped, 4, 17
 - read only, 4, 17
 - submit description, 19
 - write only, 4, 17
- file system
 - AFS, 108
 - NFS, 109
- file transfer
 - large files (>2Gbytes), 7
- file transfer mechanism, 26, 801
 - missing files, 565
- FILE_LOCK_VIA_MUTEX macro, 158
- filelock_midwife, 852
- filelock_undertaker, 854

- FILESYSTEM_DOMAIN macro, 149, 170, 331
- FLOCK_COLLECTOR_HOSTS macro, 199, 492
- FLOCK_FROM macro, 493
- FLOCK_NEGOTIATOR_HOSTS macro, 199, 493
- FLOCK_TO macro, 492
- flocking, 492
- Frequently Asked Questions, 547–578
- FS_REMOTE_DIR macro, 230, 297
- FULL_HOSTNAME macro, 147
- GAHP (Grid ASCII Helper Protocol), 494
- GAHP macro, 218
- GAHP_ARGS macro, 218
- GASS (Global Access to Secondary Storage), 499
- GCB (Generic Connection Brokering), 332
 - broker, 334
 - Condor client configuration, 339
 - GCB routing table configuration, 340
 - GCB routing table syntax and examples, 341
 - inagent, 334
 - security implications, 342
- GCB broker
 - configuration, 335
 - how to spawn the broker, 337
 - ports 65432 and 65430, 335
- GLEXEC macro, 223, 324
- GLEXEC_JOB macro, 223, 324, 595, 598
- GLEXEC_STARTER macro, 325, 595, 598, 616, 641, 645
- glidein, 519, 573
 - configuration, 519
- GLIDEIN_SERVER_URLS macro, 223, 519
- GLITE_LOCATION macro, 219, 511, 512
- Globus
 - gatekeeper errors, 574
- GLOBUS_GATEKEEPER_TIMEOUT macro, 218
- GRAM (Grid Resource Allocation and Management), 499
- green computing, 426–429
- grid computing
 - Condor-C, 494
 - FAQs, 573
 - glidein, 519
 - Grid Monitor, 508
 - matchmaking, 513
 - submitting jobs to Amazon EC2, 512
 - submitting jobs to gt2, 500
 - submitting jobs to gt4, 505
 - submitting jobs to NorduGrid, 510
 - submitting jobs to PBS, 511
 - submitting jobs to Platform LSF, 511
 - submitting jobs to Unicore, 510
- GRID_MONITOR macro, 223, 509
- GRID_MONITOR_HEARTBEAT_TIMEOUT macro, 223
- GRID_MONITOR_NO_STATUS_TIMEOUT macro, 223, 654
- GRID_MONITOR_RETRY_DURATION macro, 223
- GRIDFTP_URL_BASE macro, 219, 506
- GridJobStatus
 - job ClassAd attribute, 880
- GRIDMANAGER_CHECKPROXY_INTERVAL macro, 217
- GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT macro, 218
- GRIDMANAGER_CONTACT_SCHEDD_DELAY macro, 217
- GRIDMANAGER_DEBUG macro, 609
- GRIDMANAGER_EMPTY_RESOURCE_DELAY macro, 217
- GRIDMANAGER_GAHP_CALL_TIMEOUT macro, 218, 613
- GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT macro, 218
- GRIDMANAGER_JOB_PROBE_INTERVAL macro, 217
- GRIDMANAGER_LOG macro, 217, 631
- GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE macro, 218
- GRIDMANAGER_MAX_PENDING_REQUESTS macro, 218
- GRIDMANAGER_MAX_PENDING_SUBMITS macro, 218
- GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE macro, 218, 616
- GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE macro, 217, 616
- GRIDMANAGER_MAX_WS_DESTROYS_PER_RESOURCE macro, 218, 614

-
- GRIDMANAGER_MINIMUM_PROXY_TIME
 - macro, 217
 - GRIDMANAGER_RESOURCE_PROBE_DELAY
 - macro, 217
 - GRIDMANAGER_RESOURCE_PROBE_INTERVAL
 - macro, 217
 - GRIDMAP macro, 230, 292, 299
 - GridResource
 - job ClassAd attribute, 880
 - GROUP_AUTOREGROUP macro, 215, 251
 - GROUP_AUTOREGROUP_<groupname>
 - macro, 215
 - GROUP_AUTOREGROUP_chemistry macro, 251
 - GROUP_NAMES macro, 215, 251
 - GROUP_PRIO_FACTOR_<groupname>
 - macro, 215, 251
 - GROUP_QUOTA_<groupname> macro, 215
 - groups
 - accounting, 249
 - quotas, 250
 - GSI (Grid Security Infrastructure), 498
 - GSI_DAEMON_CERT macro, 229, 291
 - GSI_DAEMON_DIRECTORY macro, 229, 291, 292
 - GSI_DAEMON_KEY macro, 229, 291
 - GSI_DAEMON_NAME macro, 229, 605, 654
 - GSI_DAEMON_PROXY macro, 230, 291
 - GSI_DAEMON_TRUSTED_CA_DIR macro, 229, 291
 - GT2_GAHP macro, 219
 - GT4_GAHP macro, 219
 - HA_<SUBSYS>_LOCK_HOLD_TIME macro, 237
 - HA_<SUBSYS>_LOCK_URL macro, 236
 - HA_<SUBSYS>_POLL_PERIOD macro, 237
 - HA_LOCK_HOLD_TIME macro, 236
 - HA_LOCK_URL macro, 236
 - HA_POLL_PERIOD macro, 237
 - HAD macro, 238
 - HAD_ARGS macro, 238
 - HAD_CONNECTION_TIMEOUT macro, 237
 - HAD_DEBUG macro, 238
 - HAD_LIST macro, 237
 - HAD_LOG macro, 238
 - HAD_UPDATE_INTERVAL macro, 238
 - HAD_USE_PRIMARY macro, 237
 - HAD_USE_REPLICATION macro, 238, 358
 - heterogeneous pool
 - submitting a job to, 34
 - HIBERNATE macro, 193, 426, 601
 - HIBERNATE_CHECK_INTERVAL macro, 193, 426, 601
 - High Availability, 354
 - of central manager, 356
 - of job queue, 354
 - of job queue, with remote job submission, 355
 - sample configuration, 359
 - High-Performance Computing (HPC), 1
 - High-Throughput Computing (HTC), 1
 - HighLoad macro, 272
 - HIGHPORT macro, 166, 327
 - HISTORY macro, 154
 - HOLD_JOB_IF_CREDENTIAL_EXPIRES
 - macro, 217
 - HoldKillSig
 - job ClassAd attribute, 880
 - HoldReason
 - job ClassAd attribute, 880
 - HoldReasonCode
 - job ClassAd attribute, 880
 - HoldReasonSubCode
 - job ClassAd attribute, 880
 - HOOK_EVICT_CLAIM macro, 193, 462
 - HOOK_FETCH_WORK macro, 193, 461, 462, 464
 - HOOK_JOB_CLEANUP macro, 468
 - HOOK_JOB_EXIT macro, 463
 - HOOK_JOB_FINALIZE macro, 468
 - HOOK_PREPARE_JOB macro, 462, 881
 - HOOK_REPLY_CLAIM macro, 193
 - HOOK_REPLY_FETCH macro, 462
 - HOOK_TRANSLATE macro, 467
 - HOOK_UPDATE_JOB_INFO macro, 462, 464, 468, 585
 - host certificate, 290
 - HOSTALLOW macro, 162
 - HOSTALLOW. . . macro, 162
 - HOSTALLOW_* macro, 575
 - HOSTALLOW_ADMINISTRATOR macro, 133, 343
 - HOSTALLOW_CONFIG macro, 343
-

- HOSTALLOW_DAEMON macro, 653
- HOSTALLOW_NEGOTIATOR macro, 343
- HOSTALLOW_NEGOTIATOR_SCHEDD macro, 493
- HOSTALLOW_READ macro, 133, 313
- HOSTALLOW_WRITE macro, 119, 133, 137, 520, 668
- HOSTDENY macro, 162
- HOSTDENY_* macro, 575
- HOSTNAME macro, 147
- HPC (High-Performance Computing), 1
- HTC (High-Throughput Computing), 1
- `https://ec2.amazonaws.com/` macro, 513
- IGNORE_NFS_LOCK_ERRORS macro, 171
- ImageSize, 563
 - job ClassAd attribute, 880
- IN_HIGHPORT macro, 166, 327
- IN_LOWPORT macro, 166, 327
- INCLUDE macro, 151
- install_release, 856
- installation
 - checkpoint server, 346
 - CondorView Client, 139
 - download, 117
 - Java, 422
 - running as root, 119
 - using Linux RPMs, 137
 - vm universe, 424
 - Windows, 127–136
 - with *condor_configure*, 123
- INVALID_LOG_FILES macro, 208, 746
- IP_ADDRESS macro, 147
- IS_OWNER macro, 180, 261
- IS_VALID_CHECKPOINT_PLATFORM macro, 179, 254
- Java, 16, 48, 422
 - job example, 49
 - multiple class files, 51
 - using JAR files, 51
 - using packages, 52
- JAVA macro, 192, 422, 506
- Java Virtual Machine, 16, 48, 422
- JAVA_CLASSPATH_ARGUMENT macro, 192
- JAVA_CLASSPATH_DEFAULT macro, 192
- JAVA_CLASSPATH_SEPARATOR macro, 192
- JAVA_EXTRA_ARGUMENTS macro, 192, 423
- JAVA_MAXHEAP_ARGUMENT macro, 192, 423, 587
- job
 - analysis, 41
 - batch ready, 14
 - completion, 45
 - credential error on Windows, 567
 - dependencies within, 64
 - exiting with signal 9 *Unix*, 557
 - exiting with status 128 *NT*, 568
 - heterogeneous submit, 34
 - image size, 563
 - lease, 110
 - log events, 42
 - multiple data sets, 2
 - not running, 41
 - not running, why?, 557
 - preparation, 14
 - priority, 41, 47
 - state, 39, 40, 881
 - submission using a shared file system, 24
 - submission without a shared file system, 26
 - submitting, 19
 - universe, 882
- job deferral time, 95
- job execution
 - at a specific time, 95
- Job hooks, 460–468
 - Fetch Hooks
 - Job exit, 463
 - Update job info, 462
 - Evict a claim, 462
 - Fetch work, 461
 - Prepare job, 462
 - Reply to fetched work, 461
 - FetchWorkDelay, 465
 - Hooks, 461
 - Hooks invoked by Condor, 461
 - introduction, 460
 - Java Example, 466
 - Job Router, 467
 - Job Router Hooks
 - Job Cleanup, 468

- Job Finalize, 468
 - Translate, 467
 - Update Job Info, 467
- keywords, 464
- overview, 460
- job ID
 - cluster identifier, 818, 879
 - process identifier, 884
 - use in *condor_wait*, 849
- Job monitor, 107
- Job Router, 521
- Job Router commands
 - condor_router_history*, 775
- Job Router Routing Table ClassAd attribute
 - Copy_ ATTR_*, 527
 - Delete_ ATTR_*, 527
 - Eval_Set_ ATTR_*, 527
 - FailureRateThreshold*, 526
 - GridResource*, 526
 - JobFailureTest*, 526
 - JobShouldBeSandboxed*, 527
 - MaxIdleJobs*, 526
 - MaxJobs*, 526
 - Name*, 526
 - OverrideRoutingEntry*, 527
 - Requirements*, 526
 - Set_ ATTR_*, 527
 - SharedX509UserProxy*, 527
 - TargetUniverse*, 527
 - UseSharedX509UserProxy*, 527
- job scheduling
 - periodic, 98
- JOB_INHERITS_STARTER_ENVIRONMENT*
 - macro, 205, 599, 616, 658
- JOB_IS_FINISHED_INTERVAL* macro, 197
- JOB_PROXY_OVERRIDE_FILE* macro, 665
- JOB_RENICE_INCREMENT* macro, 204, 254
- JOB_ROUTER_DEFAULTS* macro, 220
- JOB_ROUTER_ENTRIES* macro, 220, 528
- JOB_ROUTER_ENTRIES_CMD* macro, 220, 528
- JOB_ROUTER_ENTRIES_FILE* macro, 220
- JOB_ROUTER_ENTRIES_REFRESH* macro, 220
- JOB_ROUTER_MAX_JOBS* macro, 221
- JOB_ROUTER_NAME* macro, 221
- JOB_ROUTER_POLLING_PERIOD* macro, 221, 468
- JOB_ROUTER_SOURCE_JOB_CONSTRAINT* macro, 221
- JOB_START_COUNT* macro, 196, 197
- JOB_START_DELAY* macro, 196, 197, 627
- JobLeaseDuration*
 - job ClassAd attribute, 110, 881
- JobPrio*
 - job ClassAd attribute, 881
- JobRunCount*
 - job ClassAd attribute, 881
- JobStartDate*
 - job ClassAd attribute, 881
- JobStatus*
 - job ClassAd attribute, 881
- JobUniverse*
 - job ClassAd attribute, 882
- JVM, 16, 48, 422
- KEEP_POOL_HISTORY* macro, 210, 396
- Kerberos authentication, 294
- KERBEROS_CLIENT_KEYTAB* macro, 232
- KERBEROS_MAP_FILE* macro, 294, 299
- KERBEROS_SERVER_KEYTAB* macro, 232
- KERBEROS_SERVER_PRINCIPAL* macro, 232, 294
- KERBEROS_SERVER_SERVICE* macro, 232
- KERBEROS_SERVER_USER* macro, 232
- KeyboardBusy* macro, 272
- KILL* macro, 179, 271
- KILLING_TIMEOUT* macro, 268, 271
- large file transfer support, 7
- LastCheckpointPlatform*
 - job ClassAd attribute, 882
- LastCkpt* macro, 272
- LastCkptServer*
 - job ClassAd attribute, 882
- LastCkptTime*
 - job ClassAd attribute, 882
- LastMatchTime*
 - job ClassAd attribute, 882
- LastRejMatchReason*
 - job ClassAd attribute, 882
- LastRejMatchTime*

- job ClassAd attribute, 882
- LastSuspensionTime
 - job ClassAd attribute, 882
- LastVacateTime
 - job ClassAd attribute, 883
- LeaseManager . CLASSAD_LOG macro, 222
- LeaseManager . DEBUG_ADS macro, 222
- LeaseManager . DEFAULT_MAX_LEASE_DURATION macro, 222
- LeaseManager . GETADS_INTERVAL macro, 221
- LeaseManager . MAX_LEASE_DURATION macro, 222
- LeaseManager . MAX_TOTAL_LEASE_DURATION macro, 222
- LeaseManager . PRUNE_INTERVAL macro, 222
- LeaseManager . QUERY_ADTYPE macro, 222
- LeaseManager . QUERY_CONSTRAINTS macro, 222
- LeaseManager . UPDATE_INTERVAL macro, 222
- LIB macro, 151
- LIBEXEC macro, 151, 201
- linking
 - dynamic, 5, 17
 - static, 5, 17
- Linux
 - keyboard and mouse activity, 530, 562
- LINUX_HIBERNATION_METHOD macro, 194
- local universe, 19
- LOCAL_CONFIG_DIR macro, 153
- LOCAL_CONFIG_FILE macro, 122, 147, 152, 391–393, 555
- LOCAL_CREDD macro, 533
- LOCAL_DIR macro, 120, 123, 151, 322
- LOCAL_QUEUE_BACKUP_DIR macro, 595
- LOCAL_XACT_BACKUP_FILTER macro, 595
- LocalSysCpu
 - job ClassAd attribute, 883
- LocalUserCpu
 - job ClassAd attribute, 883
- LOCK macro, 121, 154
- LOCK_FILE_UPDATE_INTERVAL macro, 165, 611
- log files
 - event descriptions, 42
- LOG macro, 152, 156, 182, 353
- LOG_ON_NFS_IS_ERROR macro, 207, 633
- LOGS_USE_TIMESTAMP macro, 158, 633
- LOWPORT macro, 166, 327
- LSF, 511
- LSF_GAHP macro, 219, 511
- machine
 - central manager, 113
 - checkpoint server, 113
 - execute, 113
 - owner, 112
 - submit, 113
- machine activity, 260
 - Backfill, 261
 - Benchmarking, 260
 - Busy, 260
 - Idle, 260
 - Killing, 261
 - Retiring, 260
 - Suspended, 260
 - transitions, 261–271
 - transitions summary, 270
 - Unclaimed, 260
 - Vacating, 261
- machine ClassAd, 13
- machine state, 256
 - Backfill, 257, 269
 - Claimed, 257, 265
 - claimed, the claim lease, 259
 - Matched, 257, 265
 - Owner, 256, 261
 - Preempting, 257, 268
 - transitions, 261–271
 - transitions summary, 270
 - Unclaimed, 256, 264
- machine state and activities figure, 261
- MachineBusy macro, 272
- macro
 - in configuration file, 143
 - in submit description file, 818
 - predefined, 56, 61
 - subsystem names, 147
- MAIL macro, 154, 392

- mailing lists, 9, 578
- MASTER_<name>_BACKOFF_CEILING macro, 176, 642
- MASTER_<name>_BACKOFF_CONSTANT macro, 175, 642
- MASTER_<name>_BACKOFF_FACTOR macro, 176
- MASTER_<name>_RECOVER_FACTOR macro, 176
- MASTER_<SUBSYS>_CONTROLLER macro, 237
- MASTER_ADDRESS_FILE macro, 178
- MASTER_ATTRS macro, 178
- MASTER_BACKOFF_CEILING macro, 176, 642
- MASTER_BACKOFF_CONSTANT macro, 175, 642
- MASTER_BACKOFF_FACTOR macro, 176
- MASTER_CHECK_INTERVAL macro, 208
- MASTER_CHECK_NEW_EXEC_INTERVAL macro, 175
- MASTER_DEBUG macro, 178
- MASTER_HA_LIST macro, 236, 355, 633
- MASTER_HAD_BACKOFF_CONSTANT macro, 358
- MASTER_INSTANCE_LOCK macro, 178
- MASTER_NAME macro, 177, 735
- MASTER_NEW_BINARY_DELAY macro, 175, 642
- MASTER_RECOVER_FACTOR macro, 176, 642
- MASTER_SHUTDOWN_<Name> macro, 175
- MASTER_UPDATE_INTERVAL macro, 175, 642
- MASTER_WAITS_FOR_GCB_BROKER macro, 168, 638, 643
- MATCH_TIMEOUT macro, 258, 265, 270
- matched state, 257, 265
- matchmaking, 2
 - negotiation algorithm, 247
 - on the Grid, 513
- MAX_<SUBSYS>_<LEVEL>_LOG macro, 160
- MAX_<SUBSYS>_LOG macro, 157, 160, 665
- MAX_ACCOUNTANT_DATABASE_SIZE macro, 212
- MAX_C_GAHP_LOG macro, 219
- MAX_CKPT_SERVER_LOG macro, 348
- MAX_CLAIM_ALIVES_MISSED macro, 181, 197
- MAX_CONCURRENT_DOWNLOADS macro, 196, 622, 627
- MAX_CONCURRENT_UPLOADS macro, 196, 622, 627
- MAX_DAGMAN_LOG macro, 71, 226
- MAX_DISCARDED_RUN_TIME macro, 172, 346
- MAX_EVENT_LOG macro, 161, 596, 634
- MAX_HAD_LOG macro, 238
- MAX_HISTORY_LOG macro, 155
- MAX_HISTORY_ROTATIONS macro, 155, 663
- MAX_JOB_MIRROR_UPDATE_LAG macro, 221
- MAX_JOB_QUEUE_LOG_ROTATIONS macro, 155
- max_job_retirement_time, 817
- MAX_JOBS_RUNNING macro, 39, 195, 328, 893
- MAX_JOBS_SUBMITTED macro, 196
- MAX_NEXT_JOB_START_DELAY macro, 197, 627, 806, 883
- MAX_NUM_CPUS macro, 182
- MAX_PENDING_STARTD_CONTACTS macro, 196, 598
- MAX_REPLICATION_LOG macro, 239
- MAX_SCHEDD_LOG macro, 315
- MAX_SHADOW_EXCEPTIONS macro, 196
- MAX_SLOT_TYPES macro, 185
- MAX_TRACKING_GID macro, 420, 628
- MAX_TRANSFERER_LIFETIME macro, 238
- MAX_TRANSFERER_LOG macro, 239
- MAX_VM_GAHP_LOG macro, 233
- MaxHosts
 - job ClassAd attribute, 883
- MaxJobRetirementTime
 - job ClassAd attribute, 883
- MAXJOBRETIREMENTTIME macro, 180, 271
- MAXJOBRETIREMENTTIME macro, 279
- MaxSuspendTime macro, 272
- MaxVacateTime macro, 272
- MEMORY macro, 183
- migration, 2, 3
- MIN_TRACKING_GID macro, 420, 628
- MinHosts
 - job ClassAd attribute, 883
- MPI application
 - under the dedicated scheduler, 58, 408
- MPI universe, 59–64
- MPI_CONDOR_RSH_PATH macro, 201

- multiple network interfaces, 329
- MY., ClassAd scope resolution prefix, 439
- MYPROXY_GET_DELEGATION macro, 242, 508
- NEGOTIATE_ALL_JOBS_IN_CLUSTER
 - macro, 199, 248
- negotiation, 247
 - by group, 250
- NEGOTIATOR_ADDRESS_FILE macro, 326
- NEGOTIATOR_ADDRESS_FILE macro, 163
- NEGOTIATOR_CONSIDER_PREEMPTION
 - macro, 215, 280, 608
- NEGOTIATOR_CYCLE_DELAY macro, 211
- NEGOTIATOR_DEBUG macro, 214
- NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES
 - macro, 212
- NEGOTIATOR_HOST macro, 150
- NEGOTIATOR_IGNORE_USER_PRIORITIES
 - macro, 517
- NEGOTIATOR_INFORM_STARTD macro, 212, 627
- NEGOTIATOR_INTERVAL macro, 211
- NEGOTIATOR_MATCH_EXPRS macro, 214, 639, 640
- NEGOTIATOR_MATCHLIST_CACHING macro, 517, 665
- NEGOTIATOR_MAX_TIME_PER_PIE
 - macro, 214
- NEGOTIATOR_MAX_TIME_PER_SUBMITTER
 - macro, 214
- NEGOTIATOR_POST_JOB_RANK macro, 213
- NEGOTIATOR_PRE_JOB_RANK macro, 213
- NEGOTIATOR_SOCKET_CACHE_SIZE macro, 212, 328
- NEGOTIATOR_TIMEOUT macro, 212
- NEGOTIATOR_USE_NONBLOCKING_STARTD_COMMANDS
 - macro, 167
- NET_REMAP_ENABLE macro, 167
- NET_REMAP_INAGENT macro, 167, 643
- NET_REMAP_ROUTE macro, 168
- NET_REMAP_SERVICE macro, 167
- network, 4, 17, 325
- network interfaces
 - multiple, 329
- NETWORK_INTERFACE macro, 165, 330, 332
- NETWORK_MAX_PENDING_CONNECTS macro, 157
- NextJobStartDelay
 - job ClassAd attribute, 883
- NFS
 - interaction with, 109
- nice job, 48
- NICE_USER_PRIO_FACTOR macro, 212, 245
- NiceUser
 - job ClassAd attribute, 883
- NICs, 329
- NIS
 - Condor must be dynamically linked, 564
- NO_DNS macro, 155
- NOES macro, 56, 61
- Node macro, 818
- NONBLOCKING_COLLECTOR_UPDATE macro, 167
- NonCondorLoadAvg macro, 272
- NorduGrid, 510
- NORDUGRID_GAHP macro, 220
- NOT_RESPONDING_TIMEOUT macro, 164
- NOT_RESPONDING_WANT_CORE macro, 165, 589
- NTDomain
 - job ClassAd attribute, 883
- NUM_CPUS macro, 182, 186, 402
- NUM_SLOTS macro, 186, 402
- NUM_SLOTS_TYPE_<N> macro, 186, 402
- NumCkpts
 - job ClassAd attribute, 883
- NumGlobusSubmits
 - job ClassAd attribute, 883
- NumJobMatches
 - job ClassAd attribute, 883
- NumJobReconnects
 - job ClassAd attribute, 883
- NumJobStarts
 - job ClassAd attribute, 883
- NumRestarts
 - job ClassAd attribute, 883
- NumShadowExceptions
 - job ClassAd attribute, 883
- NumShadowStarts
 - job ClassAd attribute, 884
- NumSystemHolds

- job ClassAd attribute, 884
- OBITUARY_LOG_LENGTH macro, 174
- OFFLINE_EXPIRE_ADS_AFTER macro, 194, 428, 584
- OFFLINE_LOG macro, 194, 428, 584
- opportunistic scheduling, 408
- OPSYS macro, 148, 392
- OUT_HIGHPORT macro, 167, 327
- OUT_LOWPORT macro, 166, 327
- overview, 1–4
- Owner
 - job ClassAd attribute, 884
- owner
 - of directories, 120
- owner state, 256, 261
- parallel universe, 19, 55–64
 - running MPI applications, 58
- ParallelSchedulingGroup macro, 202, 411, 412
- ParallelShutdownPolicy
 - job ClassAd attribute, 884
- PASSWD_CACHE_REFRESH macro, 156
- PBS (Portable Batch System), 511
- PBS_GAHP macro, 219, 511
- PER_JOB_HISTORY_DIR macro, 202, 647
- PERIODIC_CHECKPOINT macro, 179, 443, 555
- PERIODIC_EXPR_INTERVAL macro, 200, 637
- PERIODIC_EXPR_TIMESLICE macro, 200, 640
- PERIODIC_MEMORY_SYNC macro, 203
- Perl module, 482
 - examples, 485
- permission denied, 575
- PERSISTENT_CONFIG_DIR macro, 162
- Personal Condor, 129, 550
- PID macro, 148
- pie slice, 248
- pie spin, 248
- platform-specific information
 - AIX, 545
 - AIX 5.1L, 545
 - AIX 5.2L, 545
 - Linux, 529
 - Linux keyboard and mouse activity, 530
 - Macintosh OS X, 545
 - Red Hat 9.x, 530
 - Red Hat Fedora 1, 2, and 3, 530
 - Windows, 531–544
- platforms supported, 5
- policy
 - at UW-Madison, 274
 - default with Condor, 272
 - desktop/non-desktop, 277
 - disabling preemption, 279
 - suspending jobs instead of evicting them, 280
 - time of day, 276
- POLLING_INTERVAL macro, 180, 266
- pool of machines, 112
- POOL_HISTORY_DIR macro, 210, 396
- POOL_HISTORY_MAX_STORAGE macro, 210, 396
- POOL_HISTORY_SAMPLING_INTERVAL macro, 210
- port usage, 325
 - conflicts, 329
 - FAQ on communication errors, 551
 - firewalls, 327
 - multiple collectors, 328
 - nonstandard ports for central managers, 326
- power management, 426–429
 - entering a low power state, 426
 - leaving a low power state, 427
 - Linux platform details, 428
 - Windows platform troubleshooting, 428
- PPID macro, 149
- PREEMPT macro, 179, 270, 463
- preempting state, 257, 268
- preemption
 - desktop/non-desktop, 277
 - disabling, 279
 - priority, 47, 245
 - vacate, 48
- PREEMPTION_RANK macro, 214, 618
- PREEMPTION_RANK_STABLE macro, 214, 246, 584, 618
- PREEMPTION_REQUIREMENTS macro, 47, 213, 215, 246, 618, 753

-
- PREEMPTION_REQUIREMENTS_STABLE
 macro, 213, 246, 584, 618
 PREEN macro, 174
 PREEN_ADMIN macro, 207, 746
 PREEN_ARGS macro, 174
 PREEN_INTERVAL macro, 174, 641, 642, 644
 priority
 by group, 249
 in machine allocation, 244
 nice job, 48
 of a job, 41, 47
 of a user, 47
 PRIORITY_HALFLIFE macro, 47, 212, 245, 247
 PRIVATE_NETWORK_INTERFACE macro, 166, 634
 PRIVATE_NETWORK_NAME macro, 165, 634
 privilege separation, 320
 PrivSep (privilege separation), 320
 PRIVSEP_ENABLED macro, 232, 323, 628
 PRIVSEP_SWITCHBOARD macro, 232, 323, 628
 PROCD_ADDRESS macro, 216, 611
 PROCD_LOG macro, 216
 PROCD_MAX_SNAPSHOT_INTERVAL macro, 216
 process
 definition for a submitted job, 884
 Process macro, 818
 ProcId
 job ClassAd attribute, 884
 proxy, 500
 renewal with *MyProxy*, 506
 PUBLISH_OBITUARIES macro, 174

 Q_QUERY_TIMEOUT macro, 156
 QDate
 job ClassAd attribute, 884
 QUERY_TIMEOUT macro, 208
 QUEUE_ALL_USERS_TRUSTED macro, 198
 QUEUE_CLEAN_INTERVAL macro, 198
 QUEUE_SUPER_USERS macro, 198
 Quill, 362
 QUILL macro, 239
 QUILL_ADDRESS_FILE macro, 241, 367
 QUILL_ARGS macro, 239
 QUILL_DB_IP_ADDR macro, 240, 364, 366
 QUILL_DB_NAME macro, 240, 366
 QUILL_DB_QUERY_PASSWORD macro, 241, 367
 QUILL_DB_TYPE macro, 240
 QUILL_DB_USER macro, 240, 366
 QUILL_DB_SIZE_LIMIT macro, 241, 367
 QUILL_ENABLED macro, 240, 893
 QUILL_IS_REMOTELY_QUERYABLE macro, 241, 367
 QUILL_JOB_HISTORY_DURATION macro, 241, 366
 QUILL_LOG macro, 240
 QUILL_MAINTAIN_DB_CONN macro, 240, 367
 QUILL_MANAGE_VACUUM macro, 241, 367
 QUILL_NAME macro, 240, 366
 QUILL_NOT_RESPONDING_TIMEOUT macro, 240
 QUILL_POLLING_PERIOD macro, 240, 366
 QUILL_RESOURCE_HISTORY_DURATION macro, 241, 366
 QUILL_RUN_HISTORY_DURATION macro, 241, 366
 QUILL_SHOULD_REINDEX macro, 241, 656
 QUILL_USE_SQL_LOG macro, 240
 quotas
 for a group, 250

 RANDOM_CHOICE() macro
 use in submit description file, 820
 RANDOM_CHOICE () macro, 149
 RANDOM_INTEGER () macro, 149, 555
 rank attribute, 22
 examples, 23, 441
 RANK macro, 179, 255, 271, 410, 411
 RANK_FACTOR macro, 411
 real user priority (RUP), 244
 recovery from crashes, 575
 RELEASE_DIR macro, 122, 151, 392
 ReleaseReason
 job ClassAd attribute, 884
 remote system call, 2, 3, 16
 condor_shadow, 16, 39, 109
 REMOTE_PRIO_FACTOR macro, 212, 245
 RemoteIwd
 job ClassAd attribute, 884
-

- RemoteSysCpu
 - job ClassAd attribute, 884
- RemoteUserCpu
 - job ClassAd attribute, 884
- RemoteWallClockTime
 - job ClassAd attribute, 884
- RemoveKillSig
 - job ClassAd attribute, 885
- REPLICATION macro, 239
- REPLICATION_ARGS macro, 239
- REPLICATION_DEBUG macro, 239
- REPLICATION_INTERVAL macro, 238
- REPLICATION_LIST macro, 238
- REPLICATION_LOG macro, 239
- REQUEST_CLAIM_TIMEOUT macro, 197
- REQUIRE_LOCAL_CONFIG_FILE macro, 153
- requirements attribute, 22, 441
 - automatic extensions, 558
- Requirements macro, 195
- RESERVE_AFS_CACHE macro, 170
- RESERVED_DISK macro, 154, 887
- RESERVED_MEMORY macro, 183
- RESERVED_SWAP macro, 154, 558
- resource
 - management, 2
 - offer, 3
 - owner, 112
 - request, 3
- RPM installation on Linux, 137
- RUNBENCHMARKS macro, 183, 264, 270
- running a job
 - at certain times of day, 555
 - on a different architecture, 34
 - on only certain machines, 553
 - only at night, 555
- running multiple programs, 21
- SBIN macro, 151
- scalability
 - using the Grid Monitor, 508
- SCHED_UNIV_RENICE_INCREMENT macro, 198
- SCHEDD_ADDRESS_FILE macro, 199
- SCHEDD_ASSUME_NEGOTIATOR_GONE macro, 201
- SCHEDD_ATTRS macro, 199
- SCHEDD_BACKUP_SPOOL macro, 201
- SCHEDD_CLUSTER_INCREMENT_VALUE macro, 202, 581
- SCHEDD_CLUSTER_INITIAL_VALUE macro, 202, 581
- SCHEDD_DAEMON_AD_FILE macro, 163, 633
- SCHEDD_DEBUG macro, 199
- SCHEDD_EXECUTE macro, 199
- SCHEDD_HOST macro, 151
- SCHEDD_INTERVAL macro, 101, 196
- SCHEDD_INTERVAL_TIMESLICE macro, 196, 640
- SCHEDD_LOCK macro, 198
- SCHEDD_LOG macro, 558
- SCHEDD_MIN_INTERVAL macro, 196
- SCHEDD_NAME macro, 177, 199, 356, 610
- SCHEDD_PREEMPTION_RANK macro, 202, 412
- SCHEDD_PREEMPTION_REQUIREMENTS macro, 201, 411
- SCHEDD_QUERY_WORKERS macro, 196
- SCHEDD_ROUND_ATTR macro, 662
- SCHEDD_ROUND_ATTR_<xxxx> macro, 201
- SCHEDD_SEND_VACATE_VIA_TCP macro, 202
- scheduler universe, 18
- scheduling
 - dedicated, 55, 60, 408
 - opportunistic, 408
 - pie slice, 248
 - pie spin, 248
- scheduling jobs
 - to execute at a specific time, 95
 - to execute periodically, 98
- SDK
 - Chirp, 53
- SEC_*_AUTHENTICATION macro, 229
- SEC_*_AUTHENTICATION_METHODS macro, 229
- SEC_*_CRYPTO_METHODS macro, 229
- SEC_*_ENCRYPTION macro, 229
- SEC_*_INTEGRITY macro, 229
- SEC_*_NEGOTIATION macro, 229
- SEC_ADMINISTRATOR_AUTHENTICATION macro, 288
- SEC_ADMINISTRATOR_AUTHENTICATION_METHODS macro, 288

-
- | | |
|--|---|
| SEC_ADMINISTRATOR_CRYPTOMETHODS
macro, 300 | SEC_CONFIG_AUTHENTICATIONMETHODS
macro, 288 |
| SEC_ADMINISTRATOR_ENCRYPTION macro,
299 | SEC_CONFIG_CRYPTOMETHODS macro, 300 |
| SEC_ADMINISTRATOR_INTEGRITY macro,
301 | SEC_CONFIG_ENCRYPTION macro, 299 |
| SEC_ADVERTISE_MASTER_AUTHENTICATION
macro, 288 | SEC_CONFIG_INTEGRITY macro, 301 |
| SEC_ADVERTISE_MASTER_AUTHENTICATIONMETHODS
macro, 288 | SEC_DAEMON_AUTHENTICATION macro, 288 |
| SEC_ADVERTISE_MASTER_CRYPTOMETHODS
macro, 300 | SEC_DAEMON_AUTHENTICATIONMETHODS
macro, 288 |
| SEC_ADVERTISE_MASTER_ENCRYPTION
macro, 299 | SEC_DAEMON_CRYPTOMETHODS macro, 300 |
| SEC_ADVERTISE_MASTER_INTEGRITY
macro, 301 | SEC_DAEMON_ENCRYPTION macro, 299 |
| SEC_ADVERTISE_SCHEDD_AUTHENTICATION
macro, 288 | SEC_DAEMON_INTEGRITY macro, 301 |
| SEC_ADVERTISE_SCHEDD_AUTHENTICATIONMETHODS
macro, 288 | SEC_DEFAULT_AUTHENTICATION macro,
287, 288 |
| SEC_ADVERTISE_SCHEDD_CRYPTOMETHODS
macro, 300 | SEC_DEFAULT_AUTHENTICATIONMETHODS
macro, 345 |
| SEC_ADVERTISE_SCHEDD_ENCRYPTION
macro, 299 | SEC_DEFAULT_AUTHENTICATIONMETHODS
macro, 288 |
| SEC_ADVERTISE_SCHEDD_INTEGRITY
macro, 301 | SEC_DEFAULT_AUTHENTICATION.TIMEOUT
macro, 230, 602 |
| SEC_ADVERTISE_STARTD_AUTHENTICATION
macro, 288 | SEC_DEFAULT_CRYPTOMETHODS macro,
300 |
| SEC_ADVERTISE_STARTD_AUTHENTICATIONMETHODS
macro, 288 | SEC_DEFAULT_ENCRYPTION macro, 299 |
| SEC_ADVERTISE_STARTD_CRYPTOMETHODS
macro, 300 | SEC_DEFAULT_INTEGRITY macro, 301 |
| SEC_ADVERTISE_STARTD_ENCRYPTION
macro, 299 | SEC_DEFAULT_SESSION_DURATION macro,
230, 611 |
| SEC_ADVERTISE_STARTD_INTEGRITY
macro, 301 | SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION
macro, 231, 598 |
| SEC_CLAIMTOBE_INCLUDE_DOMAIN macro,
668 | SEC_INVALIDATE_SESSIONS.VIA_TCP
macro, 230, 598 |
| SEC_CLIENT_AUTHENTICATION macro, 287 | SEC_NEGOTIATOR_AUTHENTICATION
macro, 288 |
| SEC_CLIENT_AUTHENTICATIONMETHODS
macro, 288 | SEC_NEGOTIATOR_AUTHENTICATIONMETHODS
macro, 288 |
| SEC_CLIENT_CRYPTOMETHODS macro, 300 | SEC_NEGOTIATOR_CRYPTOMETHODS
macro, 300 |
| SEC_CLIENT_ENCRYPTION macro, 299 | SEC_NEGOTIATOR_ENCRYPTION macro, 299 |
| SEC_CLIENT_INTEGRITY macro, 301 | SEC_NEGOTIATOR_INTEGRITY macro, 301 |
| SEC_CONFIG_AUTHENTICATION macro, 288 | SEC_OWNER_AUTHENTICATION macro, 288 |
| | SEC_OWNER_AUTHENTICATIONMETHODS
macro, 288 |
| | SEC_OWNER_CRYPTOMETHODS macro, 300 |
| | SEC_OWNER_ENCRYPTION macro, 299 |
| | SEC_OWNER_INTEGRITY macro, 301 |
| | SEC_PASSWORD_FILE macro, 231, 295 |
| | SEC_READ_AUTHENTICATION macro, 288 |
-

-
- SEC_READ_AUTHENTICATION_METHODS
macro, 288
 - SEC_READ_CRYPTOMETHODS macro, 300
 - SEC_READ_ENCRYPTION macro, 299
 - SEC_READ_INTEGRITY macro, 301
 - SEC_TCP_SESSION_DEADLINE macro, 230, 581
 - SEC_TCP_SESSION_TIMEOUT macro, 230
 - SEC_WRITE_AUTHENTICATION macro, 288
 - SEC_WRITE_AUTHENTICATION_METHODS
macro, 288
 - SEC_WRITE_CRYPTOMETHODS macro, 300
 - SEC_WRITE_ENCRYPTION macro, 299
 - SEC_WRITE_INTEGRITY macro, 301
 - SECONDARY_COLLECTOR_LIST macro, 178, 637, 640
 - security
 - in Condor, 281–325
 - access levels, 283
 - authentication, 287
 - authorization, 302
 - based on user authorization, 302
 - encryption, 299
 - host-based, 307
 - integrity, 301
 - running jobs as user nobody, 318
 - sample configuration using pool password, 296
 - sample configuration using pool password for startd advertisement, 297
 - sessions, 306
 - unified map file, 298
 - sessions, 306
 - SETTABLE_ATTRS macro, 162, 314
 - SETTABLE_ATTRS... macro, 162
 - SETTABLE_ATTRS_PERMISSION-LEVEL
macro, 314
 - shadow, 16
 - SHADOW macro, 195
 - SHADOW_DEBUG macro, 202
 - SHADOW_JOB_CLEANUP_RETRY_DELAY
macro, 203
 - SHADOW_LAZY_QUEUE_UPDATE macro, 203, 627
 - SHADOW_LOCK macro, 202
 - SHADOW_LOG macro, 42
 - SHADOW_MAX_JOB_CLEANUP_RETRIES
macro, 203
 - SHADOW_QUEUE_UPDATE_INTERVAL macro, 202
 - SHADOW_RENICE_INCREMENT macro, 198
 - SHADOW_SIZE_ESTIMATE macro, 197
 - shared file system
 - submission of jobs, 24
 - submission of jobs without one, 26
 - SHELL macro, 779
 - SHUTDOWN_FAST_TIMEOUT macro, 175, 642
 - SHUTDOWN_GRACEFUL_TIMEOUT macro, 162, 642
 - signal, 4, 17
 - SIGTSTP, 4, 17
 - SIGUSR2, 4, 17
 - SIGNIFICANT_ATTRIBUTES macro, 248
 - Simple Object Access Protocol(SOAP), 468
 - skew in timing information, 564
 - SLOT_TYPE_<N> macro, 185, 400
 - SLOT_TYPE_<N>_PARTITIONABLE macro, 185, 407
 - SLOTN_JOB_HOOK_KEYWORD macro, 193, 464
 - slots
 - dynamic *condor_startd* provisioning, 407
 - subdividing slots, 407
 - SLOTS_CONNECTED_TO_CONSOLE macro, 184, 403
 - SLOTS_CONNECTED_TO_KEYBOARD macro, 184, 403
 - SLOTx_EXECUTE macro, 152, 401
 - SLOTx_USER macro, 169, 319
 - SLOW_CKPT_SPEED macro, 203
 - SMP machines
 - configuration, 399–408
 - SOAP
 - Web Service API, 468
 - SOAP_LEAVE_IN_QUEUE macro, 242, 471
 - SOAP_SSL_CA_DIR macro, 243
 - SOAP_SSL_CA_FILE macro, 243
 - SOAP_SSL_DH_FILE macro, 243
 - SOAP_SSL_SERVER_KEYFILE macro, 243
 - SOAP_SSL_SERVER_KEYFILE_PASSWORD
macro, 243
 - SOFT_UID_DOMAIN macro, 169, 316
 - Software Developer's Kit
-

- Chirp, 53
- Solaris26, 559
- Solaris27, 559
- SPOOL macro, 152, 595
- START macro, 179, 184, 253, 270, 410
- START_BACKFILL macro, 184, 265, 271, 413, 416
- START_DAEMONS macro, 175
- START_LOCAL_UNIVERSE macro, 195, 637, 640, 893
- START_MASTER macro, 175
- START_SCHEDULER_UNIVERSE macro, 195, 637, 640, 893
- startd
 - configuration, 252
- STARTD_AD_REEVAL_EXPR macro, 609
- STARTD_ADDRESS_FILE macro, 182
- STARTD_ATTRS macro, 163, 181, 315, 350, 406, 412
- STARTD_AVAIL_CONFIDENCE macro, 192
- STARTD_CLAIM_ID_FILE macro, 182
- STARTD_COMPUTE_AVAIL_STATS macro, 191
- STARTD_CRON_<ModuleName>_ARGS macro, 189
- STARTD_CRON_<ModuleName>_CWD macro, 189
- STARTD_CRON_<ModuleName>_ENV macro, 189
- STARTD_CRON_<ModuleName>_EXECUTABLE macro, 187
- STARTD_CRON_<ModuleName>_KILL macro, 188
- STARTD_CRON_<ModuleName>_MODE macro, 188
- STARTD_CRON_<ModuleName>_OPTIONS macro, 189
- STARTD_CRON_<ModuleName>_PERIOD macro, 188
- STARTD_CRON_<ModuleName>_PREFIX macro, 187
- STARTD_CRON_<ModuleName>_RECONFIG macro, 188
- STARTD_CRON_AUTOPUBLISH macro, 187
- STARTD_CRON_CONFIG_VAL macro, 186
- STARTD_CRON_JOBLIST macro, 187
- STARTD_CRON_JOBS macro, 190
- STARTD_CRON_NAME macro, 186
- STARTD_DEBUG macro, 181, 315, 644
- STARTD_EXPRS macro, 163
- STARTD_HAS_BAD_UTMP macro, 181
- STARTD_JOB_EXPRS macro, 181, 213
- STARTD_JOB_HOOK_KEYWORD macro, 193, 464
- STARTD_MAX_AVAIL_PERIOD_SAMPLES macro, 192
- STARTD_NAME macro, 183
- STARTD_NOCLAIM_SHUTDOWN macro, 183
- STARTD_RESOURCE_PREFIX macro, 184, 629, 636
- STARTD_SENDS_ALIVES macro, 182
- STARTD_SHOULD_WRITE_CLAIM_ID_FILE macro, 182
- STARTD_SLOT_ATTRS macro, 185
- STARTD_VM_ATTRS macro, 185
- STARTD_VM_EXPRS macro, 185
- STARTER macro, 180
- STARTER_ALLOW_RUNAS_OWNER macro, 169, 318, 419, 639, 641
- STARTER_CHOOSSES_CKPT_SERVER macro, 172, 349
- STARTER_DEBUG macro, 204
- STARTER_INITIAL_UPDATE_INTERVAL macro, 463
- STARTER_JOB_ENVIRONMENT macro, 205
- STARTER_JOB_HOOK_KEYWORD macro, 464
- STARTER_LOCAL macro, 195
- STARTER_LOCAL_LOGGING macro, 204
- STARTER_UPDATE_INTERVAL macro, 204, 463, 627
- STARTER_UPDATE_INTERVAL_TIMESLICE macro, 204, 602
- STARTER_UPLOAD_TIMEOUT macro, 205, 618, 638, 640
- StartIdleTime macro, 272
- state
 - of a machine, 256
 - transitions, 261–271
 - transitions summary, 270
- state and activities figure, 261
- STATE_FILE macro, 238
- StateTimer macro, 272

- status
 - of queued jobs, 39
- Stork, 102–106
 - condor_lease_manager*, 106
 - jobs under DAGMan, 106
 - submit description file, 102
- Stork commands
 - stork_list_cred*, 860
 - stork_q*, 858
 - stork_rm*, 862
 - stork_rm_cred*, 864
 - stork_status*, 868
 - stork_store_cred*, 866
 - stork_submit*, 870
- Stork submit commands
 - alt_protocols*, 872
 - cred_name*, 872
 - dap_type*, 870
 - dest_url*, 871
 - src_url*, 871
 - x509proxy*, 872
- stork_list_cred* command, 860
- STORK_MAX_NUM_JOBS macro, 243
- STORK_MAX_RETRY macro, 104, 243
- STORK_MAXDELAY_INMINUTES macro, 243
- STORK_MODULE_DIR macro, 243
- stork_q* command, 858
- stork_rm* command, 862
- stork_rm_cred* command, 864
- stork_server* daemon, 116
- stork_status* command, 868
- stork_store_cred* command, 866
- stork_submit* command, 870
- STORK_TMP_CRED_DIR macro, 243
- StreamErr
 - job ClassAd attribute, 885
- StreamOut
 - job ClassAd attribute, 885
- submit commands
 - \$ENV macro, 819
 - \$RANDOM_CHOICE() macro, 820
 - allow_startup_script*, 806
 - amazon_ami_id*, 809
 - amazon_instance_type*, 809
 - amazon_keypair_file*, 809
 - amazon_private_key*, 809
 - amazon_public_key*, 809
 - amazon_security_groups*, 809
 - amazon_user_data*, 809
 - amazon_user_data_file*, 809
 - append_files*, 806
 - arguments*, 796
 - buffer_block_size*, 806
 - buffer_files*, 806
 - buffer_size*, 806
 - compress_files*, 807
 - concurrency_limits*, 814
 - copy_to_spool*, 814
 - coresize*, 814
 - cron_day_of_month*, 814
 - cron_day_of_week*, 814
 - cron_hour*, 814
 - cron_minute*, 814
 - cron_month*, 815
 - cron_prep_time*, 815
 - cron_window*, 815
 - deferral_prep_time*, 815
 - deferral_time*, 815
 - deferral_window*, 815
 - email_attributes*, 815
 - environment*, 797
 - error*, 798
 - executable*, 798
 - fetch_files*, 807
 - file_remaps*, 807
 - getenv*, 798
 - globus_rematch*, 809
 - globus_resubmit*, 809
 - globus_rsl*, 809
 - globus_xml*, 810
 - grid_resource*, 501, 505, 810
 - hold*, 803
 - hold_kill_sig*, 812
 - image_size*, 815
 - initialdir*, 815
 - input*, 799
 - jar_files*, 812
 - java_vm_args*, 812
 - job_lease_duration*, 816
 - keystore_alias*, 810
 - keystore_file*, 810
 - keystore_passphrase_file*, 810

- kill_sig, 816
- leave_in_queue, 803
- load_profile, 816
- local_files, 808
- log, 799, 871
- log_xml, 799
- machine_count, 812
- match_list_length, 816
- max_job_retirement_time, 817
- MyProxyCredentialName, 810
- MyProxyHost, 810
- MyProxyNewProxyLifetime, 811
- MyProxyPassword, 811
- MyProxyRefreshThreshold, 811
- MyProxyServerDN, 811
- next_job_start_delay, 805
- nice_user, 817
- noop_job, 817
- noop_job_exit_code, 817
- noop_job_exit_signal, 817
- nordugrid_rsl, 811
- notification, 799
- notify_user, 799
- on_exit_hold, 803
- on_exit_remove, 804
- output, 799
- periodic_hold, 805
- periodic_release, 805
- periodic_remove, 805
- priority, 800
- queue, 800
- rank, 23, 33, 800
- remote_initialdir, 817
- remove_kill_sig, 812
- rendezvousdir, 818
- request_cpus, 818
- request_disk, 818
- request_memory, 818
- requirements, 22, 32, 800
- should_transfer_files, 26, 801
- stream_error, 802
- stream_input, 802
- stream_output, 802
- transfer_error, 811
- transfer_executable, 802
- transfer_input, 811
- transfer_input_files, 802
- transfer_output, 811
- transfer_output_files, 802
- transfer_output_remaps, 803
- universe, 800
- vm_cdrom_files, 812
- vm_checkpoint, 812
- vm_macaddr, 812
- vm_memory, 813
- vm_networking, 813
- vm_networking_type, 813
- vm_no_output_vm, 813
- vm_should_transfer_cdrom_files, 813
- vm_type, 813
- vmware_dir, 813
- vmware_should_transfer_files, 813
- vmware_snapshot_disk, 813
- want_remote_io, 808
- when_to_transfer_output, 26, 565, 803
- x509userproxy, 811
- xen_cdrom_device, 813
- xen_disk, 813
- xen_initrd, 814
- xen_kernel, 814
- xen_kernel_params, 814
- xen_root, 814
- xen_transfer_files, 814
- submit description file, 19
 - contents of, 19
 - examples, 20–22
 - grid universe, 501
- submit machine, 113
- SUBMIT_DEBUG macro, 160
- SUBMIT_EXPRS macro, 207
- SUBMIT_MAX_PROCS_IN_CLUSTER macro, 207
- SUBMIT_SEND_RESCHEDULE macro, 207
- SUBMIT_SKIP_FILECHECKS macro, 207
- substitution macro
 - in submit description file, 818
- <SUBSYS> macro, 173
- <SUBSYS>_ADDRESS_FILE macro, 163
- <SUBSYS>_ARGS macro, 174
- <SUBSYS>_ATTRS macro, 163
- <SUBSYS>_DAEMON_AD_FILE macro, 163
- <SUBSYS>_DEBUG macro, 158

- <SUBSYS>_DEBUG macro levels
 - D_ACCOUNTANT, 160
 - D_ALL, 158
 - D_CKPT, 159
 - D_COMMAND, 159
 - D_DAEMONCORE, 159
 - D_FDS, 160
 - D_FULLDEBUG, 158
 - D_HOSTNAME, 159
 - D_JOB, 159
 - D_KEYBOARD, 159
 - D_LOAD, 159
 - D_MACHINE, 159
 - D_MATCH, 159
 - D_NETWORK, 159
 - D_PID, 160
 - D_PRIV, 159
 - D_PROCFAMILY, 160
 - D_PROTOCOL, 160
 - D_SECURITY, 160
 - D_SYSCALLS, 159
- <SUBSYS>_ENABLE_SOAP_SSL macro, 242
- <SUBSYS>_EXPRS macro, 163
- <SUBSYS>_<LEVEL>_LOG macro, 160
- <SUBSYS>_LOCK macro, 158
- <SUBSYS>_LOG macro, 157
- <SUBSYS>_SOAP_SSL_PORT macro, 243
- <SUBSYS>_TIMEOUT_MULTIPLIER macro, 167
- SUBSYSTEM macro, 147
- subsystem names, 147
- supported platforms, 5
- SUSPEND macro, 179, 270, 561
- SYSAPI_GET_LOADAVG macro, 157
- SYSTEM_PERIODIC_HOLD macro, 200, 620
- SYSTEM_PERIODIC_RELEASE macro, 200, 620
- SYSTEM_PERIODIC_REMOVE macro, 200, 620
- TARGET., ClassAd scope resolution prefix, 439
- TCP, 345
 - sending updates, 345
- TCP_UPDATE_COLLECTORS macro, 167
- thread
 - kernel-level, 4, 17
 - user-level, 4, 17
- TILDE macro, 147
- timing information
 - incorrect, 564
- TOOL_DEBUG macro, 160
- TotalSuspensions
 - job ClassAd attribute, 885
- TOUCH_LOG_INTERVAL macro, 158
- TRANSFERER macro, 239, 583
- TRANSFERER_DEBUG macro, 239
- TRANSFERER_LOG macro, 239
- TransferErr
 - job ClassAd attribute, 885
- TransferExecutable
 - job ClassAd attribute, 885
- TransferIn
 - job ClassAd attribute, 885
- TransferOut
 - job ClassAd attribute, 885
- transferring files, 26, 565
- TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN macro, 161
- TRUNC_<SUBSYS>_LOG_ON_OPEN macro, 158, 161
- TRUST_UID_DOMAIN macro, 169
- UDP, 345
 - lost datagrams, 345
- UID
 - effective, 315
 - potential risk running jobs as user nobody, 318
 - real, 315
- UID_DOMAIN macro, 149, 168, 316, 317, 331, 799
- UIDs in Condor, 315–320
- UNAME_ARCH macro, 148
- UNAME_OPSYS macro, 148
- unclaimed state, 256, 264
- Unicore, 510
- UNICORE_GAHP macro, 220
- uniq_pid_midwife, 874
- uniq_pid_undertaker, 876
- universe, 15
 - Grid, 16, 18
 - grid, 495, 519
 - grid, grid type gt2, 500

- grid, grid type gt4, 505
- Java, 18
- java, 16
- job attribute definitions, 882
- local, 19
- MPI, 16
- parallel, 16, 19
- scheduler, 18
- standard, 16
- vanilla, 16, 18
- vm, 16, 19, 91
- Unix
 - alarm, 4, 17
 - exec, 4, 17
 - flock, 4, 17
 - fork, 4, 17
 - large files, 5, 18
 - lockf, 4, 17
 - mmap, 4, 17
 - pipe, 4, 17
 - semaphore, 4, 17
 - shared memory, 4, 17
 - sleep, 4, 17
 - socket, 4, 17
 - system, 4, 17
 - timer, 4, 17
- Unix administrator, 119
- Unix daemon
 - running as root, 109
- Unix directory
 - execute, 120
 - lock, 121
 - log, 120
 - spool, 120
- Unix installation
 - download, 117
- Unix user
 - condor, 120
 - root, 119
- UPDATE_COLLECTOR_WITH_TCP macro, 167, 209, 345
- UPDATE_INTERVAL macro, 180, 187, 263
- upgrade
 - version 6.4.x to version 6.6.x, 548
- upgrading
 - items to be aware of, 580
- USE_AFS macro, 171
- USE_CKPT_SERVER macro, 172, 348, 349
- USE_CLONE_TO_CREATE_PROCESSES
 - macro, 164, 631, 637, 640
- USE_GID_PROCESS_TRACKING macro, 420, 628
- USE_NFS macro, 171
- USE_PROCD macro, 216, 232, 323, 420, 639, 641
- USE_PROCESS_GROUPS macro, 178, 645
- USE_VISIBLE_DESKTOP macro, 205, 535, 595, 596
- user
 - priority, 47
- user condor
 - home directory not found, 564
- user manual, 11–111
- user nobody
 - potential security risk with jobs, 318
- user priority, 244
 - effective (EUP), 245
 - real (RUP), 244
- USER_JOB_WRAPPER macro, 204, 569
- USERNAME macro, 149
- vacate, 48
- VALID_COD_USERS macro, 448
- VALID_SPOOL_FILES macro, 207, 236, 355, 601, 612, 746
- vanilla jobs
 - cycling between suspended and unsuspended, 560
- VARS, 73
- viewing
 - log files, 107
- virtual machine
 - configuration, 398
 - running Condor jobs under, 398
- virtual machine universe, 91–95
- virtual machines, 424
- vm universe, 19, 91
 - submit commands specific to VMware, 93
- VM_GAHP_LOG macro, 233, 592
- VM_GAHP_REQ_TIMEOUT macro, 233
- VM_GAHP_SERVER macro, 233
- VM_MAX_NUMBER macro, 233

- VM_MEMORY macro, 233
- VM_NETWORKING macro, 234
- VM_NETWORKING_DEFAULT_TYPE macro, 234
- VM_NETWORKING_TYPE macro, 234
- VM_RECHECK_INTERVAL macro, 233
- VM_SOFT_SUSPEND macro, 233
- VM_STATUS_INTERVAL macro, 233
- VM_TYPE macro, 233
- VM_UNIV_NOBODY_USER macro, 233
- VM_VERSION macro, 233
- VMP_HOST_MACHINE macro, 235, 398
- VMP_VM_LIST macro, 235, 398
- VMWARE_BRIDGE_NETWORKING_TYPE macro, 234
- VMWARE_LOCAL_SETTINGS_FILE macro, 235
- VMWARE_NAT_NETWORKING_TYPE macro, 234
- VMWARE_NETWORKING_TYPE macro, 234
- VMWARE_PERL macro, 234
- VMWARE_SCRIPT macro, 234
- WALL_CLOCK_CHKPT_INTERVAL macro, 198
- WANT_SUSPEND macro, 180, 270
- WANT_UDP_COMMAND_SOCKET macro, 157, 212, 627
- WANT_VACATE macro, 180, 271
- WARN_ON_UNUSED_SUBMIT_FILE_MACROS macro, 207, 627, 820
- Web Service API, 468
 - condor_schedd* daemon command port, 471
 - file transfer, 470
 - job submission, 469
 - transactions, 469
- WEB_ROOT_DIR macro, 242
- Windows
 - Condor daemon names, 136
 - installation, 127–136
 - initial file size, 127
 - location of files, 130
 - Personal Condor, 129
 - preparation, 128
 - required disk space, 128
 - unattended install, 132
 - loading account profile, 534
 - manual install, 135
 - out of desktop heap, 570
 - release notes, 531
 - starting the Condor service, 136
- WINDOWS_FIREWALL_FAILURE_RETRY macro, 178
- WorkHours macro, 277
- X509_USER_PROXY, 34
- XEN_BOOTLOADER macro, 235
- XEN_BRIDGE_SCRIPT macro, 235
- XEN_DEFAULT_INITRD macro, 235, 425
- XEN_DEFAULT_KERNEL macro, 235, 425
- XEN_LOCAL_SETTINGS_FILE macro, 235
- XEN_SCRIPT macro, 235