HTCondor™ Version 8.6.10 Manual

Center for High Throughput Computing, University of Wisconsin–Madison

March 13, 2018

Contents

1 Overview **1**

1.1 High-Throughput Computing (HTC) and its Requirements 1

1.2 HTCondor's Power 2

1.3 Exceptional Features 3

1.4 Current Limitations 4

1.5 Availability 5

1.6 Contributions and Acknowledgments 5

1.7 Contact Information 7

1.8 Privacy Notice 7

2 Users' Manual **9**

2.1 Welcome to HTCondor 9

2.2 Introduction 9

2.3 Matchmaking with ClassAds 10

2.3.1 Inspecting Machine ClassAds with condor_status 10

2.4 Running a Job: the Steps To Take 12

2.4.1 Choosing an HTCondor Universe 13

2.5 Submitting a Job 16

2.5.1 Sample submit description files 17

2.5.2 Using the Power and Flexibility of the Queue Command 20

2.5.3 Variables in the Submit Description File 22

2.5.4 Including Submit Commands Defined Elsewhere . . . 23
2.5.5 Using Conditionals in the Submit Description File . . . 23
2.5.6 Function Macros in the Submit Description File . . . 26
2.5.7 About Requirements and Rank . . . 29
2.5.8 Submitting Jobs Using a Shared File System . . . 31
2.5.9 Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism . . . 32
2.5.10 Environment Variables . . . 42
2.5.11 Heterogeneous Submit: Execution on Differing Architectures . . . 43
2.5.12 Jobs That Require GPUs . . . 47
2.5.13 Interactive Jobs . . . 48
2.6 Managing a Job . . . 49
2.6.1 Checking on the progress of jobs . . . 49
2.6.2 Removing a job from the queue . . . 51
2.6.3 Placing a job on hold . . . 52
2.6.4 Changing the priority of jobs . . . 52
2.6.5 Why is the job not running? . . . 53
2.6.6 Job in the Hold State . . . 55
2.6.7 In the Job Event Log File . . . 55
2.6.8 Job Completion . . . 59
2.7 Priorities and Preemption . . . 59
2.7.1 Job Priority . . . 60
2.7.2 User priority . . . 60
2.7.3 Details About How HTCondor Jobs Vacate Machines . . . 61
2.8 Java Applications . . . 61
2.8.1 A Simple Example Java Application . . . 62
2.8.2 Less Simple Java Specifications . . . 63
2.8.3 Chirp I/O . . . 66
2.9 Parallel Applications (Including MPI Applications) . . . 68

2.9.1 How Parallel Jobs Run . . . 68
2.9.2 Parallel Jobs and the Dedicated Scheduler . . . 69
2.9.3 Submission Examples . . . 69
2.9.4 MPI Applications Within HTCondor's Vanilla Universe . . . 74
2.10 DAGMan Applications . . . 75
2.10.1 DAGMan Terminology . . . 76
2.10.2 The DAG Input File: Basic Commands . . . 77
2.10.3 Command Order . . . 83
2.10.4 Node Job Submit File Contents . . . 84
2.10.5 DAG Submission . . . 84
2.10.6 File Paths in DAGs . . . 86
2.10.7 DAG Monitoring and DAG Removal . . . 88
2.10.8 Suspending a Running DAG . . . 88
2.10.9 Advanced Features of DAGMan . . . 89
2.10.10 The Rescue DAG . . . 122
2.10.11 DAG Recovery . . . 125
2.10.12 Visualizing DAGs with *dot* . . . 126
2.10.13 Capturing the Status of Nodes in a File . . . 127
2.10.14 A Machine-Readable Event History, the jobstate.log File . . . 129
2.10.15 Status Information for the DAG in a ClassAd . . . 133
2.10.16 Utilizing the Power of DAGMan for Large Numbers of Jobs . . . 133
2.10.17 Workflow Metrics . . . 136
2.10.18 DAGMan and Accounting Groups . . . 139
2.11 Virtual Machine Applications . . . 139
2.11.1 The Submit Description File . . . 139
2.11.2 Checkpoints . . . 142
2.11.3 Disk Images . . . 143
2.11.4 Job Completion in the vm Universe . . . 143

2.11.5 Failures to Launch . . . 144
2.12 Docker Universe Applications . . . 146
2.13 Time Scheduling for Job Execution . . . 147
2.13.1 Job Deferral . . . 148
2.13.2 CronTab Scheduling . . . 150
2.14 Special Environment Considerations . . . 154
2.14.1 AFS . . . 154
2.14.2 NFS . . . 154
2.14.3 HTCondor Daemons That Do Not Run as root . . . 155
2.14.4 Job Leases . . . 156
2.15 Potential Problems . . . 156
2.15.1 Renaming of argv[0] . . . 156

3 Administrators' Manual 157
3.1 Introduction . . . 157
3.1.1 The Different Roles a Machine Can Play . . . 157
3.1.2 The HTCondor Daemons . . . 158
3.2 Installation, Start Up, Shut Down, and Reconfiguration . . . 161
3.2.1 Obtaining the HTCondor Software . . . 161
3.2.2 Installation on Unix . . . 161
3.2.3 Installation on Windows . . . 172
3.2.4 Upgrading – Installing a New Version on an Existing Pool . . . 181
3.2.5 Shutting Down and Restarting an HTCondor Pool . . . 182
3.2.6 Reconfiguring an HTCondor Pool . . . 183
3.3 Introduction to Configuration . . . 184
3.3.1 HTCondor Configuration Files . . . 184
3.3.2 Ordered Evaluation to Set the Configuration . . . 185
3.3.3 Configuration File Macros . . . 186

3.3.4 Comments and Line Continuations . . . 190
3.3.5 Multi-Line Values . . . 191
3.3.6 Executing a Program to Produce Configuration Macros . . . 191
3.3.7 Including Configuration from Elsewhere . . . 192
3.3.8 Reporting Errors and Warnings . . . 193
3.3.9 Conditionals in Configuration . . . 194
3.3.10 Function Macros in Configuration . . . 196
3.3.11 Macros That Will Require a Restart When Changed . . . 198
3.3.12 Pre-Defined Macros . . . 198
3.4 Configuration Templates . . . 201
3.4.1 Configuration Templates: Using Predefined Sets of Configuration . . . 201
3.4.2 Available Configuration Templates . . . 202
3.4.3 Configuration Template Transition Syntax . . . 205
3.4.4 Configuration Template Examples . . . 206
3.5 Configuration Macros . . . 206
3.5.1 HTCondor-wide Configuration File Entries . . . 206
3.5.2 Daemon Logging Configuration File Entries . . . 218
3.5.3 DaemonCore Configuration File Entries . . . 225
3.5.4 Network-Related Configuration File Entries . . . 228
3.5.5 Shared File System Configuration File Macros . . . 233
3.5.6 Checkpoint Server Configuration File Macros . . . 237
3.5.7 condor_master Configuration File Macros . . . 238
3.5.8 condor_startd Configuration File Macros . . . 244
3.5.9 condor_schedd Configuration File Entries . . . 261
3.5.10 condor_shadow Configuration File Entries . . . 276
3.5.11 condor_starter Configuration File Entries . . . 278
3.5.12 condor_submit Configuration File Entries . . . 283
3.5.13 condor_preen Configuration File Entries . . . 286

3.5.14 condor_collector Configuration File Entries . . . 286
3.5.15 condor_negotiator Configuration File Entries . . . 291
3.5.16 condor_procd Configuration File Macros . . . 298
3.5.17 condor_credd Configuration File Macros . . . 299
3.5.18 condor_gridmanager Configuration File Entries . . . 300
3.5.19 condor_job_router Configuration File Entries . . . 303
3.5.20 condor_lease_manager Configuration File Entries . . . 306
3.5.21 Grid Monitor Configuration File Entries . . . 307
3.5.22 Configuration File Entries Relating to Grid Usage . . . 307
3.5.23 Configuration File Entries for DAGMan . . . 308
3.5.24 Configuration File Entries Relating to Security . . . 317
3.5.25 Configuration File Entries Relating to Virtual Machines . . . 322
3.5.26 Configuration File Entries Relating to High Availability . . . 325
3.5.27 MyProxy Configuration File Macros . . . 328
3.5.28 Configuration File Macros Affecting APIs . . . 329
3.5.29 Configuration File Entries Relating to condor_ssh_to_job . . . 330
3.5.30 condor_rooster Configuration File Macros . . . 331
3.5.31 condor_shared_port Configuration File Macros . . . 332
3.5.32 Configuration File Entries Relating to Hooks . . . 334
3.5.33 Configuration File Entries Only for Windows Platforms . . . 339
3.5.34 condor_defrag Configuration File Macros . . . 339
3.5.35 *condor_gangliad* Configuration File Macros . . . 341
3.6 User Priorities and Negotiation . . . 343
3.6.1 Real User Priority (RUP) . . . 343
3.6.2 Effective User Priority (EUP) . . . 343
3.6.3 Priorities in Negotiation and Preemption . . . 344
3.6.4 Priority Calculation . . . 345
3.6.5 Negotiation . . . 346

3.6.6 The Layperson's Description of the Pie Spin and Pie Slice . . . 347
3.6.7 Group Accounting . . . 347
3.6.8 Accounting Groups with Hierarchical Group Quotas . . . 349
3.7 Policy Configuration for Execute Hosts and for Submit Hosts . . . 352
3.7.1 *condor_startd* Policy Configuration . . . 352
3.7.2 *condor_schedd* Policy Configuration . . . 392
3.8 Security . . . 394
3.8.1 HTCondor's Security Model . . . 395
3.8.2 Security Negotiation . . . 398
3.8.3 Authentication . . . 401
3.8.4 The Unified Map File for Authentication . . . 411
3.8.5 Encryption . . . 412
3.8.6 Integrity . . . 414
3.8.7 Authorization . . . 415
3.8.8 Security Sessions . . . 420
3.8.9 Host-Based Security in HTCondor . . . 421
3.8.10 Examples of Security Configuration . . . 423
3.8.11 Changing the Security Configuration . . . 425
3.8.12 Using HTCondor w/ Firewalls, Private Networks, and NATs . . . 426
3.8.13 User Accounts in HTCondor on Unix Platforms . . . 426
3.9 Networking (includes sections on Port Usage and CCB) . . . 431
3.9.1 Port Usage in HTCondor . . . 432
3.9.2 Reducing Port Usage with the *condor_shared_port* Daemon . . . 435
3.9.3 Configuring HTCondor for Machines With Multiple Network Interfaces . . . 437
3.9.4 HTCondor Connection Brokering (CCB) . . . 440
3.9.5 Using TCP to Send Updates to the *condor_collector* . . . 442
3.9.6 Running HTCondor on an IPv6 Network Stack . . . 443
3.10 The Checkpoint Server . . . 445

3.10.1 Preparing to Install a Checkpoint Server . . . 445
3.10.2 Installing the Checkpoint Server Module . . . 446
3.10.3 Configuring the Pool to Use Multiple Checkpoint Servers . . . 447
3.10.4 Checkpoint Server Domains . . . 448
3.11 DaemonCore . . . 449
3.11.1 DaemonCore and Unix signals . . . 450
3.11.2 DaemonCore and Command-line Arguments . . . 451
3.12 Monitoring . . . 452
3.12.1 Ganglia . . . 453
3.12.2 Absent ClassAds . . . 455
3.13 The High Availability of Daemons . . . 456
3.13.1 High Availability of the Job Queue . . . 456
3.13.2 High Availability of the Central Manager . . . 458
3.14 Setting Up for Special Environments . . . 464
3.14.1 Using HTCondor with AFS . . . 464
3.14.2 Enabling the Transfer of Files Specified by a URL . . . 465
3.14.3 Configuring HTCondor for Multiple Platforms . . . 467
3.14.4 Full Installation of condor_compile . . . 469
3.14.5 The *condor_kbdd* . . . 470
3.14.6 Configuring The HTCondorView Server . . . 472
3.14.7 Running HTCondor Jobs within a Virtual Machine . . . 474
3.14.8 HTCondor's Dedicated Scheduling . . . 475
3.14.9 Configuring HTCondor for Running Backfill Jobs . . . 479
3.14.10 Per Job PID Namespaces . . . 485
3.14.11 Group ID-Based Process Tracking . . . 485
3.14.12 Cgroup-Based Process Tracking . . . 486
3.14.13 Limiting Resource Usage with a User Job Wrapper . . . 487
3.14.14 Limiting Resource Usage Using Cgroups . . . 489

3.14.15 Concurrency Limits . . . 490
3.15 Java Support Installation . . . 493
3.16 Setting Up the VM and Docker Universes . . . 494
3.16.1 The VM Universe . . . 494
3.16.2 The Docker Universe . . . 497
3.17 Singularity Support . . . 500
3.18 Power Management . . . 501
3.18.1 Entering a Low Power State . . . 501
3.18.2 Returning From a Low Power State . . . 502
3.18.3 Keeping a ClassAd for a Hibernating Machine . . . 502
3.18.4 Linux Platform Details . . . 502
3.18.5 Windows Platform Details . . . 503

4 Miscellaneous Concepts 504

4.1 HTCondor's ClassAd Mechanism . . . 504
4.1.1 ClassAds: Old and New . . . 505
4.1.2 Old ClassAd Syntax . . . 506
4.1.3 Old ClassAd Evaluation Semantics . . . 516
4.1.4 Old ClassAds in the HTCondor System . . . 520
4.1.5 Extending ClassAds with User-written Functions . . . 522
4.2 HTCondor's Checkpoint Mechanism . . . 523
4.2.1 Standalone Checkpoint Mechanism . . . 524
4.2.2 Checkpoint Safety . . . 525
4.2.3 Checkpoint Warnings . . . 525
4.2.4 Checkpoint Library Interface . . . 526
4.3 Computing On Demand (COD) . . . 527
4.3.1 Overview of How COD Works . . . 528
4.3.2 Authorizing Users to Create and Manage COD Claims . . . 528

4.3.3 Defining a COD Application . . . 528
4.3.4 Managing COD Resource Claims . . . 532
4.3.5 Limitations of COD Support in HTCondor . . . 538
4.4 Hooks . . . 539
4.4.1 Job Hooks That Fetch Work . . . 539
4.4.2 Hooks for a Job Router . . . 546
4.4.3 Daemon ClassAd Hooks . . . 548
4.5 Logging in HTCondor . . . 550
4.5.1 Job and Daemon Logs . . . 550
4.5.2 DAGMan Logs . . . 553

5 Grid Computing 554
5.1 Introduction . . . 554
5.2 Connecting HTCondor Pools with Flocking . . . 555
5.2.1 Flocking Configuration . . . 555
5.2.2 Job Considerations . . . 556
5.3 The Grid Universe . . . 556
5.3.1 HTCondor-C, The condor Grid Type . . . 556
5.3.2 HTCondor-G, the gt2, and gt5 Grid Types . . . 560
5.3.3 The nordugrid Grid Type . . . 568
5.3.4 The unicore Grid Type . . . 569
5.3.5 The batch Grid Type (for PBS, LSF, SGE, and SLURM) . . . 569
5.3.6 The EC2 Grid Type . . . 571
5.3.7 The GCE Grid Type . . . 575
5.3.8 The cream Grid Type . . . 577
5.3.9 The BOINC Grid Type . . . 578
5.3.10 Matchmaking in the Grid Universe . . . 579
5.4 The HTCondor Job Router . . . 584

5.4.1 Routing Mechanism . . . 584
5.4.2 Job Submission with Job Routing Capability . . . 585
5.4.3 An Example Configuration . . . 587
5.4.4 Routing Table Entry ClassAd Attributes . . . 588
5.4.5 Example: constructing the routing table from ReSS . . . 590

6 Application Programming Interfaces (APIs) 591

6.1 Web Service . . . 591
6.1.1 Transactions . . . 591
6.1.2 Job Submission . . . 592
6.1.3 File Transfer . . . 593
6.1.4 Implementation Details . . . 594
6.1.5 Get These Items Correct . . . 595
6.1.6 Methods for Transaction Management . . . 595
6.1.7 Methods for Job Submission . . . 596
6.1.8 Methods for File Transfer . . . 597
6.1.9 Methods for Job Management . . . 598
6.1.10 Methods for ClassAd Management . . . 601
6.1.11 Methods for Version Information . . . 602
6.1.12 Common Data Structures . . . 602
6.2 The DRMAA API . . . 603
6.2.1 Implementation Details . . . 603
6.3 The HTCondor User and Job Log Reader API . . . 604
6.3.1 Constants and Enumerated Types . . . 605
6.3.2 Constructors and Destructors . . . 605
6.3.3 Initializers . . . 607
6.3.4 Primary Methods . . . 608
6.3.5 Accessors . . . 609

6.3.6 Methods for saving and restoring persistent reader state . . . 609
6.3.7 Save state to persistent storage . . . 609
6.3.8 Restore state from persistent storage . . . 610
6.3.9 API Reference . . . 610
6.3.10 Access to the persistent state data . . . 611
6.3.11 Future persistence API . . . 613
6.4 Chirp . . . 614
6.5 The Command Line Interface . . . 614
6.6 The HTCondor Perl Module . . . 614
6.6.1 Subroutines . . . 615
6.6.2 Examples . . . 617
6.7 Python Bindings . . . 622
6.7.1 `htcondor` Module . . . 622
6.7.2 Sample Code using the `htcondor` Python Module . . . 633
6.7.3 ClassAd Module . . . 634
6.7.4 Sample Code using the `classad` Module . . . 638

7 Platform-Specific Information 641

7.1 Linux . . . 641
7.1.1 Linux Address Space Randomization . . . 642
7.2 Microsoft Windows . . . 642
7.2.1 Limitations under Windows . . . 642
7.2.2 Supported Features under Windows . . . 643
7.2.3 Secure Password Storage . . . 644
7.2.4 Executing Jobs as the Submitting User . . . 644
7.2.5 The condor_credd Daemon . . . 644
7.2.6 Executing Jobs with the User's Profile Loaded . . . 646
7.2.7 Using Windows Scripts as Job Executables . . . 646

7.2.8 How HTCondor for Windows Starts and Stops a Job . . . 648
7.2.9 Security Considerations in HTCondor for Windows . . . 649
7.2.10 Network files and HTCondor . . . 650
7.2.11 Interoperability between HTCondor for Unix and HTCondor for Windows . . . 652
7.2.12 Some differences between HTCondor for Unix -vs- HTCondor for Windows . . . 652
7.3 Macintosh OS X . . . 653

8 Frequently Asked Questions (FAQ) 654

9 Contrib and Source Modules 655
9.1 Introduction . . . 655
9.2 Using HTCondor with the Hadoop File System . . . 655
9.2.1 condor_hdfs Configuration File Entries . . . 656
9.3 Quill . . . 657
9.3.1 Installation and Configuration . . . 658
9.3.2 Four Usage Examples . . . 664
9.3.3 Quill and Security . . . 664
9.3.4 Quill and Its RDBMS Schema . . . 665
9.4 The HTCondorView Client Contrib Module . . . 684
9.4.1 Step-by-Step Installation of the HTCondorView Client . . . 684
9.5 Job Monitor/Log Viewer . . . 686
9.5.1 Transition States . . . 687
9.5.2 Events . . . 687
9.5.3 Selecting Jobs . . . 687
9.5.4 Zooming . . . 687
9.5.5 Keyboard and Mouse Shortcuts . . . 687

10 Version History and Release Notes 689
10.1 Introduction to HTCondor Versions . . . 689

10.1.1 HTCondor Version Number Scheme . . . 689
10.1.2 The Stable Release Series . . . 690
10.1.3 The Development Release Series . . . 690
10.2 Upgrading from the 8.4 series to the 8.6 series of HTCondor . . . 690
10.3 Stable Release Series 8.6 . . . 692
10.4 Development Release Series 8.5 . . . 705
10.5 Stable Release Series 8.4 . . . 719

11 Command Reference Manual (man pages) 734

bosco_cluster . . . 735
bosco_findplatform . . . 737
bosco_install . . . 738
bosco_ssh_start . . . 739
bosco_start . . . 740
bosco_stop . . . 741
bosco_uninstall . . . 742
condor_advertise . . . 743
condor_check_userlogs . . . 747
condor_checkpoint . . . 748
condor_chirp . . . 751
condor_cod . . . 755
condor_compile . . . 758
condor_config_val . . . 760
condor_configure . . . 765
condor_continue . . . 770
condor_convert_history . . . 772
condor_dagman . . . 774
condor_dagman_metrics_reporter . . . 780

condor_drain . . . 783
condor_fetchlog . . . 785
condor_findhost . . . 788
condor_gather_info . . . 790
condor_gpu_discovery . . . 793
condor_history . . . 796
condor_hold . . . 799
condor_install . . . 802
condor_job_router_info . . . 807
condor_master . . . 809
condor_off . . . 810
condor_on . . . 813
condor_ping . . . 816
condor_pool_job_report . . . 819
condor_power . . . 820
condor_preen . . . 822
condor_prio . . . 824
condor_procd . . . 826
condor_q . . . 829
condor_qedit . . . 844
condor_qsub . . . 846
condor_reconfig . . . 851
condor_release . . . 854
condor_reschedule . . . 856
condor_restart . . . 858
condor_rm . . . 861
condor_rmdir . . . 864
condor_router_history . . . 866

condor_router_q . . . 868
condor_router_rm . . . 870
condor_run . . . 872
condor_set_shutdown . . . 875
condor_ssh_to_job . . . 877
condor_sos . . . 881
condor_stats . . . 883
condor_status . . . 886
condor_store_cred . . . 894
condor_submit . . . 896
condor_submit_dag . . . 938
condor_suspend . . . 945
condor_tail . . . 947
condor_transfer_data . . . 949
condor_transform_ads . . . 951
condor_update_machine_ad . . . 954
condor_updates_stats . . . 956
condor_urlfetch . . . 959
condor_userlog . . . 961
condor_userprio . . . 964
condor_vacate . . . 969
condor_vacate_job . . . 971
condor_version . . . 974
condor_wait . . . 976
condor_who . . . 979
gidd_alloc . . . 983
procd_ctl . . . 984

12 Appendix A: ClassAd Attributes **986**

13 Appendix B: Codes and Other Needed Values **1042**

Index **1046**

LICENSING AND COPYRIGHT

HTCondor is released under the Apache License, Version 2.0.

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

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 (50outstanding 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

Chapter 1

Overview

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 HTCondor's Power

HTCondor 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. HTCondor 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 HTCondor. HTCondor finds an available machine on the network and begins running the job on that machine. HTCondor has the capability to detect that a machine running a HTCondor 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. HTCondor continues the job on the new machine from precisely where it left off.

In those cases where HTCondor can checkpoint and migrate a job, HTCondor 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.

HTCondor 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 HTCondor. Depending upon the number of machines in the HTCondor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment.

HTCondor does not require an account (login) on machines where it runs a job. HTCondor 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.

HTCondor 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. HTCondor 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 HTCondor 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. HTCondor 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, HTCondor 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 HTCondor libraries, users of HTCondor may be assured that their jobs will eventually complete, even in the ever changing environment that HTCondor utilizes. As a machine running a job submitted to HTCondor becomes unavailable, the job can be check pointed. The job may continue after migrating to another machine. HTCondor's 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 HTCondor 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 HTCondor executes their programs there. The program behaves under HTCondor 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 HTCondor. HTCondor is able to run non-interactive programs. The checkpoint and migration of programs by HTCondor 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 HTCondor that allows jobs submitted within a first pool of HTCondor 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 HTCondor following the dependencies given by the graph.

HTCondor Enables Grid Computing. As grid computing becomes a reality, HTCondor is already there. The technique of glidein allows jobs submitted to HTCondor to be executed on grid machines in various locations worldwide. As the details of grid computing evolve, so does HTCondor'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. HTCondor handles this automatically.

ClassAds. The ClassAd mechanism in HTCondor 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 HTCondor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in HTCondor's adaptation to nearly any desired policy.

1.4 Current Limitations

Limitations on Jobs which can Checkpointed Although HTCondor can schedule and run any type of process, HTCondor 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. HTCondor 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, the job must be statically linked. *condor_compile* does this by default.
12. Reading to or writing from files larger than 2 GBytes is only supported when the submit side *condor_shadow* and the standard universe user job application itself are both 64-bit executables.

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

Security Implications. HTCondor does a significant amount of work to prevent security hazards, but loopholes are known to exist. HTCondor 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. HTCondor 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, HTCondor requires that the jobs be re-linked with the HTCondor 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 source and/or object code available. HTCondor's other services are still available for these commercial packages.

1.5 Availability

HTCondor is currently available as a free download from the Internet via the World Wide Web at URL http://htcondor.org/downloads/. Binary distributions of this HTCondor Version 8.6.10 release are available for the platforms detailed in Table 1.1. A platform is an architecture/operating system combination.

In the table, *clipped* means that HTCondor does not support checkpointing or remote system calls on the given platform. This means that *standard* universe jobs are not supported. Some clipped platforms will have further limitations with respect to supported universes. See section 2.4.1 on page 13 for more details on job universes within HTCondor and their abilities and limitations.

The HTCondor source code is available for public download alongside the binary distributions.

Architecture	*Operating System*
Intel x86	- RedHat Enterprise Linux 6
	- All versions Windows Vista or greater (clipped)
x86_64	- Red Hat Enterprise Linux 6
	- Red Hat Enterprise Linux 7
	- Debian Linux 7.0 (wheezy)
	- Debian Linux 8.0 (jessie)
	- Macintosh OS X 10.7 through 10.10 (clipped)
	- Ubuntu 12.04; Precise Pangolin (clipped)
	- Ubuntu 14.04; Trusty Tahr

Table 1.1: Supported platforms in HTCondor Version 8.6.10

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

For more platform-specific information about HTCondor's support for various operating systems, see Chapter 7 on page 641.

Jobs submitted to the standard universe utilize *condor_compile* to relink programs with libraries provided by HTCondor. Table 1.2 lists supported compilers by platform for this Version 8.6.10 release. Other compilers may work, but are not supported.

1.6 Contributions and Acknowledgments

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

Platform	Compiler	Notes
Red Hat Enterprise Linux 6 on x86_64	gcc, g++, and g77	as shipped
Red Hat Enterprise Linux 7 on x86_64	gcc, g++, and g77	as shipped
Debian Linux 7.0 (wheezy) on x86_64	gcc, g++, gfortran	as shipped
Debian Linux 8.0 (jessie) on x86_64	gcc, g++, gfortran	as shipped
Ubuntu 14.04 on x86_64	gcc, g++, gfortran	as shipped

Table 1.2: Supported compilers in HTCondor Version 8.6.10

- The Globus Alliance (http://www.globus.org), for code and assistance in developing HTCondor-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 HTCondor's High Availability. The *condor_had* daemon allows one of multiple machines to function as the central manager for a HTCondor 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 HTCondor 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.

The HTCondor project wishes to acknowledge the following:

- This material is based upon work supported by the National Science Foundation under Grant Numbers MCS-8105904, OCI-0437810, and OCI-0850745. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1.7 Contact Information

The latest software releases, publications/papers regarding HTCondor and other High-Throughput Computing research can be found at the official web site for HTCondor at http://htcondor.org/.

In addition, there is an e-mail list at **htcondor-world@cs.wisc.edu**. The HTCondor Team uses this e-mail list to announce new releases of HTCondor and other major HTCondor-related news items. To subscribe or unsubscribe from the the list, follow the instructions at http://htcondor.org/mail-lists/. Because many of us receive too much e-mail as it is, you will be happy to know that the HTCondor 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 (**htcondor-users@cs.wisc.edu**) targeted at solving problems with HTCondor. HTCondor team members attempt to monitor traffic to htcondor-users, responding as they can. Follow the instructions at http://htcondor.org/mail-lists/.

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

1.8 Privacy Notice

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

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

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

We hope that you will contribute to the development of HTCondor 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 3.5.14 of this manual.

Example of data reported:

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

```
This Collector has the following IDs:
    HTCondor: 6.6.0 Nov 12 2003
    HTCondor: INTEL-LINUX-GLIBC22

                     Machines Owner Claimed Unclaimed Matched Preempting

          INTEL/LINUX      810    52     716        37       0          5
        INTEL/WINDOWS      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
```

Chapter 2

Users' Manual

2.1 Welcome to HTCondor

HTCondor is developed by the Center for High Throughput Computing 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 15 years ago. HTCondor pools have 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 research size, complexity, and scope. Over the years, the project, and now the Center for High Throughput Computing have established collaborations with scientists from around the world, and have provided them with access to many cycles. One scientist consumed 100 CPU years!

2.2 Introduction

In a nutshell, HTCondor is a specialized batch system for managing compute-intensive jobs. Like most batch systems, HTCondor provides a queuing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to HTCondor, HTCondor 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. HTCondor can schedule jobs on dedicated machines. But unlike traditional batch systems, HTCondor 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, low load average, etc.), HTCondor 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.

HTCondor 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 HTCondor allocates resources. Understanding the unique framework by which HTCondor matches submitted jobs with machines is the key to getting the most from HTCondor's scheduling algorithm.

HTCondor simplifies job submission by acting as a matchmaker of ClassAds. HTCondor'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 HTCondor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a HTCondor pool advertise their attributes, such as available 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 HTCondor 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 HTCondor 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 HTCondor to rank available machines based upon floating point performance. Or, perhaps you care only that the machine has a minimum of 128 MiB of RAM. Or, perhaps you will take any machine you can get! These job attributes and requirements are bundled up into a job ClassAd.

HTCondor 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. HTCondor makes certain that all requirements in both ClassAds are satisfied.

2.3.1 Inspecting Machine ClassAds with condor_status

Once HTCondor 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               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime
```

```
amul.cs.wisc.edu   LINUX      INTEL  Claimed   Busy     0.990  1896  0+00:07:04
slot1@amundsen.cs. LINUX      INTEL  Owner     Idle     0.000  1456  0+00:21:58
slot2@amundsen.cs. LINUX      INTEL  Owner     Idle     0.110  1456  0+00:21:59
angus.cs.wisc.edu  LINUX      INTEL  Claimed   Busy     0.940   873  0+00:02:54
anhai.cs.wisc.edu  LINUX      INTEL  Claimed   Busy     1.400  1896  0+00:03:03
apollo.cs.wisc.edu LINUX      INTEL  Unclaimed Idle     1.000  3032  0+00:00:04
arragon.cs.wisc.ed LINUX      INTEL  Claimed   Busy     0.980   873  0+00:04:29
bamba.cs.wisc.edu  LINUX      INTEL  Owner     Idle     0.040  3032 15+20:10:19
```

...

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 -long lists the machine ClassAds for all machines in the pool.

Refer to the *condor_status* command reference page located on page 886 for a complete description of the *condor_status* command.

The following shows a portion of a machine ClassAd for a single machine: turunmaa.cs.wisc.edu. Some of the listed attributes are used by HTCondor for scheduling. Other attributes are for information purposes. An important point is that *any* of the attributes in a machine ClassAd 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.

```
Machine = "turunmaa.cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
Name = "turunmaa.cs.wisc.edu"
CondorPlatform = "$CondorPlatform: x86_rhap_5 $"
Cpus = 1
IsValidCheckpointPlatform = ( ( ( TARGET.JobUniverse == 1 ) == false ) ||
 ( ( MY.CheckpointPlatform =!= undefined ) &&
 ( ( TARGET.LastCheckpointPlatform =?= MY.CheckpointPlatform ) ||
 ( TARGET.NumCkpts == 0 ) ) ) )
CondorVersion = "$CondorVersion: 7.6.3 Aug 18 2011 BuildID: 361356 $"
Requirements = ( START ) && ( IsValidCheckpointPlatform )
EnteredCurrentActivity = 1316094896
MyAddress = "<128.105.175.125:58026>"
EnteredCurrentState = 1316094896
Memory = 1897
CkptServer = "pitcher.cs.wisc.edu"
OpSys = "LINUX"
State = "Owner"
START = true
Arch = "INTEL"
Mips = 2634
Activity = "Idle"
StartdIpAddr = "<128.105.175.125:58026>"
TargetType = "Job"
LoadAvg = 0.210000
```

```
CheckpointPlatform = "LINUX INTEL 2.6.x normal 0x40000000"
Disk = 92309744
VirtualMemory = 2069476
TotalSlots = 1
UidDomain = "cs.wisc.edu"
MyType = "Machine"
```

2.4 Running a Job: the Steps To Take

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

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

Code Preparation. A job run under HTCondor must be able to run as a background batch job. HTCondor 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. HTCondor can redirect console output (`stdout` and `stderr`) and keyboard input (`stdin`) to and from files for the program. Create any needed files that contain the proper keystrokes needed for program input. Make certain the program will run correctly with the files.

The HTCondor Universe. HTCondor 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 HTCondor: the standard universe and the vanilla universe. The standard universe allows a job running under HTCondor 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 HTCondor library using the *condor_compile* command. The manual page for *condor_compile* on page 758 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 HTCondor's File Transfer mechanism.

Choose a universe under which to run the HTCondor 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 in place of `stdin` and `stdout`, and the platform type required to run the program. The number of times to run a program may be included; 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 for guidance.

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

Once submitted, HTCondor 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 HTCondor will run your jobs with *condor_prio*. If desired, HTCondor 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, HTCondor 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 an HTCondor Universe

A *universe* in HTCondor defines an execution environment. HTCondor Version 8.6.10 supports several different universes for user jobs:

- standard
- vanilla
- grid
- java
- scheduler
- local
- parallel
- vm
- docker

The **universe** under which a job runs is specified in the submit description file. If a universe is not specified, the default is vanilla, unless your HTCondor administrator has changed the default. However, we strongly encourage you to specify the universe, since the default can be changed by your HTCondor administrator, and the default that ships with HTCondor has changed.

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 grid universe allows users to submit jobs using HTCondor'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 program known as a 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. The docker universe runs a Docker container as an HTCondor job.

Standard Universe

In the standard universe, HTCondor 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.

HTCondor 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, HTCondor 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, HTCondor 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 HTCondor 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 HTCondor 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. HTCondor 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, the job must be statically linked. *condor_compile* does this by default.
12. Reading to or writing from files larger than 2 GBytes is only supported when the submit side *condor_shadow* and the standard universe user job application itself are both 64-bit executables.

Vanilla Universe

The vanilla universe in HTCondor 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. HTCondor 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 HTCondor'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 HTCondor to rely on a shared file system, such as NFS or AFS. Alternatively, HTCondor has a mechanism for transferring files on behalf of the user. In this case, HTCondor 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, HTCondor presumes a shared file system for vanilla jobs. However, if a shared file system is unavailable, a user can enable the HTCondor 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 2.5.8 on page 31. For details on using the HTCondor File Transfer mechanism, see section 2.5.9 on page 32.

Grid Universe

The Grid universe in HTCondor is intended to provide the standard HTCondor interface to users who wish to start jobs intended for remote management systems. Section 5.3 on page 556 has details on using the Grid universe. The manual page for *condor_submit* on page 896 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. HTCondor 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 an HTCondor 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 HTCondor environment. Please see section 2.9 for more details.

VM Universe

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

Please see section 2.11 for details.

Docker Universe

The docker universe runs a docker container on an execute host as a job. Please see section 2.12 for details.

2.5 Submitting a Job

A job is submitted for execution to HTCondor 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, HTCondor 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 HTCondor works toward running the job.

The contents of a submit description file have been designed to save time for HTCondor users. It is easy to submit multiple runs of a program to HTCondor with a single submit description file. To run the same program many times on different input data sets, arrange the 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, files mapped for `stdin`, `stdout`, `stderr`, command-line arguments, and shell environment; these are all specified in the submit description file. 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 file, given by file name, every time will need to use a separate subdirectory for the output of each run.

The *condor_submit* manual page is on page 896 and contains a complete and full description of how to use *condor_submit*. It also includes descriptions 899 of all of the many commands that may be placed into a submit description file. In addition, the index lists entries for each command under the heading of Submit Commands.

Note that job ClassAd attributes can be set directly in a submit file using the **+<attribute> = <value>** syntax (see 928 for details.)

2.5.1 Sample submit description files

In addition to the examples of submit description files given here, there are more in the *condor_submit* manual page (see 896).

Example 1

Example 1 is one of the simplest submit description files possible. It queues up the program *myexe* for execution somewhere in the pool. Use of the vanilla universe is implied, as that is the default when not specified in the submit description file.

An executable is compiled to run on a specific platform. Since this submit description file does not specify a platform, HTCondor will use its default, which is to run the job on a machine which has the same architecture and operating system as the machine where *condor_submit* is run to submit the job.

Standard input for this job will come from the file `inputfile`, as specified by the **input** command, and standard output for this job will go to the file `outputfile`, as specified by the **output** command. HTCondor expects to find `inputfile` in the current working directory when this job is submitted, and the system will take care of getting the input file to where it needs to be when the job is executed, as well as bringing back the output results (to the current working directory) after job execution.

A log file, `myexe.log`, will also be produced that contains events the job had during its lifetime inside of HTCondor. When the job finishes, its exit conditions will be noted in the log file. This file's contents are an excellent way to figure out what happened to submitted jobs.

```
####################
#
# Example 1
# Simple HTCondor submit description file
#
####################

Executable   = myexe
Log          = myexe.log
Input        = inputfile
Output       = outputfile
Queue
```

Example 2

Example 2 queues up one copy of the program *foo* (which had been created by *condor_compile*) for execution by HTCondor. No **input**, **output**, or **error** commands are given in the submit description file, so `stdin`, `stdout`, and `stderr` will all refer to `/dev/null`. The program may produce output by explicitly opening a file and writing to it.

```
####################
#
# Example 2
# Standard universe submit description file
#
####################

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

Example 3

Example 3 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` due to the **initialdir** command. For each copy, `stdin` will be `test.data`, `stdout` will be `loop.out`, and `stderr` will be `loop.error`. Each run will read input and write output files within its own directory. Placing data files in separate directories is a convenient way to organize data when a large group of HTCondor jobs is to run. The example file shows program submission of *mathematica* as a vanilla universe job. The vanilla universe is most often the right choice of universe when the source and/or object code is not available.

The **request_memory** command is included to ensure that the *mathematica* jobs match with and then execute on pool machines that provide at least 1 GByte of memory.

```
####################
```

```
#
# Example 3: demonstrate use of multiple
# directories for data organization.
#
####################

executable     = mathematica
universe       = vanilla
input          = test.data
output         = loop.out
error          = loop.error
log            = loop.log
request_memory = 1 GB

initialdir     = run_1
queue

initialdir     = run_2
queue
```

Example 4

The submit description file for Example 4 queues 150 runs of program *foo* which has been compiled and linked for Linux running on a 32-bit Intel processor. This job requires HTCondor to run the program on machines which have greater than 32 MiB of physical memory, and the **rank** command expresses a preference to run each instance of the program on machines with more than 64 MiB. It also advises HTCondor that this standard universe job will use up to 28000 KiB 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 HTCondor runs, checkpoints, and migrates processes for all the 150 queued programs will be written into the single file `foo.log`.

```
####################
#
# Example 4: Show off some fancy features including
# the use of pre-defined macros.
#
####################

Executable     = foo
Universe       = standard
requirements   = OpSys == "LINUX" && Arch =="INTEL"
rank           = Memory >= 64
image_size     = 28000
request_memory = 32
```

```
error   = err.$(Process)
input   = in.$(Process)
output  = out.$(Process)
log     = foo.log

queue 150
```

2.5.2 Using the Power and Flexibility of the Queue Command

A wide variety of job submissions can be specified with extra information to the **queue** submit command. This flexibility eliminates the need for a job wrapper or Perl script for many submissions.

The form of the **queue** command defines variables and expands values, identifying a set of jobs. Square brackets identify an optional item.

queue [**<int expr>**]

queue [**<int expr>**] [**<varname>**] **in** [**slice**] **<list of items>**

queue [**<int expr>**] [**<varname>**] **matching** [**files | dirs**] [**slice**] **<list of items with file globbing>**

queue [**<int expr>**] [**<list of varnames>**] **from** [**slice**] **<file name> | <list of items>**

All optional items have defaults:

- If `<int expr>` is not specified, it defaults to the value 1.
- If `<varname>` or `<list of varnames>` is not specified, it defaults to the single variable called `ITEM`.
- If `slice` is not specified, it defaults to all elements within the list. This is the Python slice `[::]`, with a step value of 1.
- If neither `files` nor `dirs` is specified in a specification using the **from** key word, then both files and directories are considered when globbing.

The list of items uses syntax in one of two forms. One form is a comma and/or space separated list; the items are placed on the same line as the **queue** command. The second form separates items by placing each list item on its own line, and delimits the list with parentheses. The opening parenthesis goes on the same line as the **queue** command. The closing parenthesis goes on its own line. The **queue** command specified with the key word **from** will always use the second form of this syntax. Example 3 below uses this second form of syntax.

The optional `slice` specifies a subset of the list of items using the Python syntax for a slice. Negative step values are not permitted.

Here are a set of examples.

Example 1

```
transfer_input_files = $(filename)
arguments            = -infile $(filename)
queue filename matching files *.dat
```

The use of file globbing expands the list of items to be all files in the current directory that end in `.dat`. Only files, and not directories are considered due to the specification of `files`. One job is queued for each file in the list of items. For this example, assume that the three files `initial.dat`, `middle.dat`, and `ending.dat` form the list of items after expansion; macro `filename` is assigned the value of one of these file names for each job queued. That macro value is then substituted into the **arguments** and **transfer_input_files** commands. The **queue** command expands to

```
transfer_input_files = initial.dat
arguments            = -infile initial.dat
queue
transfer_input_files = middle.dat
arguments            = -infile middle.dat
queue
transfer_input_files = ending.dat
arguments            = -infile ending.dat
queue
```

Example 2

```
queue 1 input in A, B, C
```

Variable `input` is set to each of the 3 items in the list, and one job is queued for each. For this example the **queue** command expands to

```
input = A
queue
input = B
queue
input = C
queue
```

Example 3

```
queue input,arguments from (
  file1, -a -b 26
  file2, -c -d 92
)
```

Using the `from` form of the options, each of the two variables specified is given a value from the list of items. For this example the **queue** command expands to

```
input = file1
arguments = -a -b 26
queue
input = file2
arguments = -c -d 92
queue
```

2.5.3 Variables in the Submit Description File

There are automatic variables for use within the submit description file.

$(Cluster) or $(ClusterId) Each set of queued jobs from a specific user, submitted from a single submit host, sharing an executable have the same value of `$(Cluster)` or `$(ClusterId)`. The first cluster of jobs are assigned to cluster 0, and the value is incremented by one for each new cluster of jobs. `$(Cluster)` or `$(ClusterId)` will have the same value as the job ClassAd attribute `ClusterId`.

$(Process) or $(ProcId) Within a cluster of jobs, each takes on its own unique `$(Process)` or `$(ProcId)` value. The first job has value 0. `$(Process)` or `$(ProcId)` will have the same value as the job ClassAd attribute `ProcId`.

$(Item) The default name of the variable when no `<varname>` is provided in a **queue** command.

$(ItemIndex) Represents an index within a list of items. When no slice is specified, the first `$(ItemIndex)` is 0. When a slice is specified, `$(ItemIndex)` is the index of the item within the original list.

$(Step) For the `<int expr>` specified, `$(Step)` counts, starting at 0.

$(Row) When a list of items is specified by placing each item on its own line in the submit description file, `$(Row)` identifies which line the item is on. The first item (first line of the list) is `$(Row)` 0. The second item (second line of the list) is `$(Row)` 1. When a list of items are specified with all items on the same line, `$(Row)` is the same as `$(ItemIndex)`.

Here is an example of a **queue** command for which the values of these automatic variables are identified.

Example 1

This example queues six jobs.

```
queue 3 in (A, B)
```

- `$(Process)` takes on the six values 0, 1, 2, 3, 4, and 5.
- Because there is no specification for the `<varname>` within this **queue** command, variable `$(Item)` is defined. It has the value `A` for the first three jobs queued, and it has the value `B` for the second three jobs queued.

- `$(Step)` takes on the three values 0, 1, and 2 for the three jobs with `$(Item)=A`, and it takes on the same three values 0, 1, and 2 for the three jobs with `$(Item)=B`.
- `$(ItemIndex)` is 0 for all three jobs with `$(Item)=A`, and it is 1 for all three jobs with `$(Item)=B`.
- `$(Row)` has the same value as `$(ItemIndex)` for this example.

2.5.4 Including Submit Commands Defined Elsewhere

Externally defined submit commands can be incorporated into the submit description file using the syntax

```
include : <what-to-include>
```

The `<what-to-include>` specification may specify a single file, where the contents of the file will be incorporated into the submit description file at the point within the file where the **include** is. Or, `<what-to-include>` may cause a program to be executed, where the output of the program is incorporated into the submit description file. The specification of `<what-to-include>` has the bar character (`|`) following the name of the program to be executed.

The **include** key word is case insensitive. There are *no* requirements for white space characters surrounding the colon character.

Included submit commands may contain further nested **include** specifications, which are also parsed, evaluated, and incorporated. Levels of nesting on included files are limited, such that infinite nesting is discovered and thwarted, while still permitting nesting.

Consider the example

```
include : list-infiles.sh |
```

In this example, the bar character at the end of the line causes the script `list-infiles.sh` to be invoked, and the output of the script is parsed and incorporated into the submit description file. If this bash script contains

```
echo "transfer_input_files = `ls -m infiles/*.dat`"
```

then the output of this script has specified the set of input files to transfer to the execute host. For example, if directory `infiles` contains the three files `A.dat`, `B.dat`, and `C.dat`, then the submit command

```
transfer_input_files = infiles/A.dat, infiles/B.dat, infiles/C.dat
```

is incorporated into the submit description file.

2.5.5 Using Conditionals in the Submit Description File

Conditional `if`/`else` semantics are available in a limited form. The syntax:

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   <statement>
   . . .
   <statement>
endif
```

An `else` key word and statements are not required, such that simple `if` semantics are implemented. The `<simple condition>` does not permit compound conditions. It optionally contains the exclamation point character (`!`) to represent the not operation, followed by

- the `defined` keyword followed by the name of a variable. If the variable is defined, the statement(s) are incorporated into the expanded input. If the variable is *not* defined, the statement(s) are not incorporated into the expanded input. As an example,

  ```
  if defined MY_UNDEFINED_VARIABLE
     X = 12
  else
     X = -1
  endif
  ```

  results in `X = -1`, when `MY_UNDEFINED_VARIABLE` is *not* yet defined.

- the `version` keyword, representing the version number of of the daemon or tool currently reading this conditional. This keyword is followed by an HTCondor version number. That version number can be of the form `x.y.z` or `x.y`. The version of the daemon or tool is compared to the specified version number. The comparison operators are

  - == for equality. Current version 8.2.3 is equal to 8.2.
  - >= to see if the current version number is greater than or equal to. Current version 8.2.3 is greater than 8.2.2, and current version 8.2.3 is greater than or equal to 8.2.
  - <= to see if the current version number is less than or equal to. Current version 8.2.0 is less than 8.2.2, and current version 8.2.3 is less than or equal to 8.2.

  As an example,

  ```
  if version >= 8.1.6
     DO_X = True
  else
     DO_Y = True
  endif
  ```

  results in defining `DO_X` as `True` if the current version of the daemon or tool reading this if statement is 8.1.6 or a more recent version.

- `True` or `yes` or the value 1. The statement(s) are incorporated.
- `False` or `no` or the value 0 The statement(s) are *not* incorporated.
- `$(<variable>)` may be used where the immediately evaluated value is a simple boolean value. A value that evaluates to the empty string is considered `False`, otherwise a value that does not evaluate to a simple boolean value is a syntax error.

The syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
elif <simple condition>
   <statement>
   . . .
   <statement>
endif
```

is the same as syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   if <simple condition>
      <statement>
      . . .
      <statement>
   endif
endif
```

Here is an example use of a conditional in the submit description file. A portion of the `sample.sub` submit description file uses the if/else syntax to define command line arguments in one of two ways:

```
if defined X
  arguments = -n $(X)
else
  arguments = -n 1 -debug
endif
```

Submit variable `X` is defined on the *condor_submit* command line with

```
condor_submit  X=3  sample.sub
```

This command line incorporates the submit command `X = 3` into the submission before parsing the submit description file. For this submission, the command line arguments of the submitted job become

```
-n 3
```

If the job were instead submitted with the command line

```
condor_submit  sample.sub
```

then the command line arguments of the submitted job become

```
-n 1 -debug
```

2.5.6 Function Macros in the Submit Description File

A set of predefined functions increase flexibility. Both submit description files and configuration files are read using the same parser, so these functions may be used in both submit description files and configuration files.

Case is significant in the function's name, so use the same letter case as given in these definitions.

$CHOICE(index, listname) or $CHOICE(index, item1, item2, ...) An item within the list is returned. The list is represented by a parameter name, or the list items are the parameters. The `index` parameter determines which item. The first item in the list is at index 0. If the index is out of bounds for the list contents, an error occurs.

$ENV(environment-variable-name[:default-value]) Evaluates to the value of environment variable `environment-variable-name`. If there is no environment variable with that name, Evaluates to UNDEFINED unless the optional `:default-value` is used; in which case it evaluates to default-value. For example,

```
A = $ENV(HOME)
```

binds `A` to the value of the `HOME` environment variable.

$F[fpduwnxbqa](filename) One or more of the lower case letters may be combined to form the function name and thus, its functionality. Each letter operates on the `filename` in its own way.

- `f` convert relative path to full path by prefixing the current working directory to it. This option works only in *condor_submit* files.
- `p` refers to the entire directory portion of `filename`, with a trailing slash or backslash character. Whether a slash or backslash is used depends on the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified.

- d refers to the last portion of the directory within the path, if specified. It will have a trailing slash or backslash, as appropriate to the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified unless u or w is used. if b is used the trailing slash or backslash will be omitted.
- u convert path separators to Unix style slash characters
- w convert path separators to Windows style backslash characters
- n refers to the file name at the end of any path, but without any file name extension. As an example, the return value from `$Fn(/tmp/simulate.exe)` will be `simulate` (without the `.exe` extension).
- x refers to a file name extension, with the associated period (`.`). As an example, the return value from `$Fn(/tmp/simulate.exe)` will be `.exe`.
- b when combined with the d option, causes the trailing slash or backslash to be omitted. When combined with the x option, causes the leading period (`.`) to be omitted.
- q causes the return value to be enclosed within quotes. Double quote marks are used unless a is also specified.
- a When combined with the q option, causes the return value to be enclosed within single quotes.

$DIRNAME(filename) is the same as $Fp(filename)

$BASENAME(filename) is the same as $Fnx(filename)

$INT(item-to-convert) or $INT(item-to-convert, format-specifier) Expands, evaluates, and returns a string version of `item-to-convert`. The `format-specifier` has the same syntax as a C language or Perl format specifier. If no `format-specifier` is specified, `"%d"` is used as the format specifier.

$RANDOM_CHOICE(choice1, choice2, choice3, ...) A random choice of one of the parameters in the list of parameters is made. For example, if one of the integers 0-8 (inclusive) should be randomly chosen:

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

$RANDOM_INTEGER(min, max [, step]) A random integer within the range `min` and `max`, inclusive, is selected. 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):

```
$RANDOM_INTEGER(0, 8, 2)
```

$REAL(item-to-convert) or $REAL(item-to-convert, format-specifier) Expands, evaluates, and returns a string version of `item-to-convert` for a floating point type. The `format-specifier` is a C language or Perl format specifier. If no `format-specifier` is specified, `"%16G"` is used as a format specifier.

$SUBSTR(name, start-index) or $SUBSTR(name, start-index, length) Expands `name` and returns a substring of it. The first character of the string is at index 0. The first character of the substring is at index `start-index`. If the optional `length` is not specified, then the substring includes characters up to the end of the string. A negative value of `start-index` works back from the end of the string. A negative value of `length` eliminates use of characters from the end of the string. Here are some examples that all assume

```
Name = abcdef
```

- `$SUBSTR(Name, 2)` is `cdef`.
- `$SUBSTR(Name, 0, -2)` is `abcd`.
- `$SUBSTR(Name, 1, 3)` is `bcd`.
- `$SUBSTR(Name, -1)` is `f`.
- `$SUBSTR(Name, 4, -3)` is the empty string, as there are no characters in the substring for this request.

Here are example uses of the function macros in a submit description file. Note that these are not complete submit description files, but only the portions that promote understanding of use cases of the function macros.

Example 1

Generate a range of numerical values for a set of jobs, where values other than those given by `$(Process)` are desired.

```
MyIndex     = $(Process) + 1
initial_dir = run-$INT(MyIndex, %04d)
```

Assuming that there are three jobs queued, such that `$(Process)` becomes 0, 1, and 2, `initial_dir` will evaluate to the directories `run-0001`, `run-0002`, and `run-0003`.

Example 2

This variation on Example 1 generates a file name extension which is a 3-digit integer value.

```
Values     = $(Process) * 10
Extension  = $INT(Values, %03d)
input      = X.$(Extension)
```

Assuming that there are four jobs queued, such that `$(Process)` becomes 0, 1, 2, and 3, `Extension` will evaluate to 000, 010, 020, and 030, leading to files defined for **input** of `X.000`, `X.010`, `X.020`, and `X.030`.

Example 3

This example uses both the file globbing of the **queue** command and a macro function to specify a job input file that is within a subdirectory on the submit host, but will be placed into a single, flat directory on the execute host.

```
arguments            = $Fnx(FILE)
transfer_input_files = $(FILE)
queue  FILE  MATCHING (
     samplerun/*.dat
     )
```

Assume that two files that end in `.dat`, `A.dat` and `B.dat`, are within the directory `samplerun`. Macro `FILE` expands to `samplerun/A.dat` and `samplerun/B.dat` for the two jobs queued. The input files transferred are `samplerun/A.dat` and `samplerun/B.dat` on the submit host. The `$Fnx()` function macro expands to the complete file name with any leading directory specification stripped, such that the command line argument for one of the jobs will be `A.dat` and the command line argument for the other job will be `B.dat`.

2.5.7 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 HTCondor 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 504.

All of the commands in the submit description file are case insensitive, *except* for the ClassAd attribute string values. ClassAd 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 HTCondor 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 987. A list of defined ClassAd attributes for machine ClassAds is given in the unnumbered Appendix on page 1005.

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 HTCondor 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 ranking 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 HTCondor 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.

2.5.8 Submitting Jobs Using a Shared File System

If vanilla, java, or parallel universe jobs are submitted without using the File Transfer mechanism, HTCondor 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.

HTCondor 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 HTCondor 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 HTCondor 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, HTCondor 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 `UidDomain`s and/or `FileSystemDomain`s, 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 HTCondor 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 HTCondor 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 HTCondor's File Transfer mechanism, the job will *only* run on the machine from which it was submitted.

2.5.9 Submitting Jobs Without a Shared File System: HTCondor's File Transfer Mechanism

HTCondor works well without a shared file system. The HTCondor file transfer mechanism permits the user to select which files are transferred and under which circumstances. HTCondor can transfer any files needed by a job from the machine where the job was submitted into a remote scratch directory on the machine where the job is to be executed. HTCondor executes the job and transfers output back to the submitting machine. The user specifies which files and directories 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.

Specifying If and When to Transfer Files

To enable the file transfer mechanism, place two commands in the job's submit description file: **should_transfer_files** and **when_to_transfer_output**. By default, they will be:

```
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
```

Setting the **should_transfer_files** command explicitly enables or disables the file transfer mechanism. The command takes on one of three possible values:

1. YES: HTCondor transfers both the executable and the file defined by the **input** command from the machine where the job is submitted to the remote machine where the job is to be executed. The file defined by the **output** command as well as any files created by the execution of the job are transferred back to the machine where the job was submitted. When they are transferred and the directory location of the files is determined by the command **when_to_transfer_output**.
2. IF_NEEDED: HTCondor 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, the same as if should_transfer_files = YES. If the job is matched with a machine in the local FileSystemDomain, HTCondor will not transfer files and relies on the shared file system.
3. NO: HTCondor's file transfer mechanism is disabled.

The **when_to_transfer_output** command tells HTCondor when output files are to be transferred back to the submit machine. The command takes on one of two possible values:

1. ON_EXIT: HTCondor transfers the file defined by the **output** command, as well as any other files in the remote scratch directory created by the job, back to the submit machine only when the job exits on its own.
2. ON_EXIT_OR_EVICT: HTCondor behaves the same as described for the value ON_EXIT when the job exits on its own. However, if, and each time the job is evicted from a machine, *files are transferred back at eviction time*. The files that are transferred back at eviction time may include intermediate files that are not part of the final output of the job. When **transfer_output_files** is specified, its list governs which are transferred back at eviction time. Before the job starts running again, all of the files that were stored when the job was last evicted are copied to the job's new remote scratch directory.

   The purpose of saving files at eviction time is to allow the job to resume from where it left off. This is similar to using the checkpoint feature of the standard universe, but just specifying ON_EXIT_OR_EVICT is not enough to make a job capable of producing or utilizing checkpoints. The job must be designed to save and restore its state using the files that are saved at eviction time.

   The files that are transferred back at eviction time are not stored in the location where the job's final output will be written when the job exits. HTCondor manages these files automatically, so usually the only reason for a user to worry about them is to make sure that there is enough space to store them. The files are stored on the submit machine in a temporary directory within the directory defined by the configuration variable SPOOL. The directory is named using the ClusterId and ProcId job ClassAd attributes. The directory name takes the form:

   ```
   <X mod 10000>/<Y mod 10000>/cluster<X>.proc<Y>.subproc0
   ```

   where <X> is the value of ClusterId, and <Y> is the value of ProcId. As an example, if job 735.0 is evicted, it will produce the directory

   ```
   $(SPOOL)/735/0/cluster735.proc0.subproc0
   ```

The default values for these two submit commands make sense as used together. If only **should_transfer_files** is set, and set to the value NO, then no output files will be transferred, and the value of **when_to_transfer_output** is irrelevant. If only **when_to_transfer_output** is set, and set to the value ON_EXIT_OR_EVICT, then the default value for an unspecified **should_transfer_files** will be YES.

Note that 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*.

Specifying What Files to Transfer

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

1. the executable, as defined with the **executable** command
2. the input, as defined with the **input** command
3. any jar files, for the **java** universe, as defined with the **jar_files** command

If the job requires other input files, the submit description file should utilize the **transfer_input_files** command. This comma-separated list specifies any other files or directories that HTCondor is to transfer to the remote scratch directory, to set up the execution environment for the job before it is run. These files are placed in the same directory as the job's executable. For example:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = file1,file2
```

This example explicitly enables the file transfer mechanism, and it transfers the executable, the file specified by the **input** command, any jar files specified by the **jar_files** command, and files `file1` and `file2`.

If the file transfer mechanism is enabled, HTCondor will transfer the following files from the execute machine back to the submit machine after the job exits.

1. the output file, as defined with the **output** command
2. the error file, as defined with the **error** command
3. any files created by the job in the remote scratch directory; this only occurs for jobs other than **grid** universe, and for HTCondor-C **grid** universe jobs; directories created by the job within the remote scratch directory are ignored for this automatic detection of files to be transferred.

A path given for **output** and **error** commands represents a path on the submit machine. If no path is specified, the directory specified with **initialdir** is used, and if that is not specified, the directory from which the job was submitted is used. At the time the job is submitted, zero-length files are created on the submit machine, at the given path for the files defined by the **output** and **error** commands. This permits job submission failure, if these files cannot be written by HTCondor.

To *restrict* the output files or permit entire directory contents to be transferred, specify the exact list with **transfer_output_files**. Delimit the list of file names, directory names, or paths with commas. When this list is defined, and any of the files or directories do not exist as the job exits, HTCondor considers this an error, and places the job on hold. Setting **transfer_output_files** to the empty string (`""`) means no files are to be transferred. When this list is defined, automatic detection of output files created by the job is disabled. Paths specified in this list refer to locations on the execute machine. The naming and placement of files and directories relies on the term *base name*. By example, the path `a/b/c` has the base name `c`. It is the file name or directory name with all directories leading up to that name stripped off. On the submit machine, the transferred files or directories are named using only the base name. Therefore, each output file or directory must have a different name, even if they originate from different paths.

For **grid** universe jobs other than than HTCondor-C grid jobs, files to be transferred (other than standard output and standard error) must be specified using **transfer_output_files** in the submit description file, because automatic detection of new files created by the job does not take place.

Here are examples to promote understanding of what files and directories are transferred, and how they are named after transfer. Assume that the job produces the following structure within the remote scratch directory:

```
o1
o2
d1 (directory)
    o3
    o4
```

If the submit description file sets

```
transfer_output_files = o1,o2,d1
```

then transferred back to the submit machine will be

```
o1
o2
d1 (directory)
    o3
    o4
```

Note that the directory `d1` and all its contents are specified, and therefore transferred. If the directory `d1` is not created by the job before exit, then the job is placed on hold. If the directory `d1` is created by the job before exit, but is empty, this is not an error.

If, instead, the submit description file sets

```
transfer_output_files = o1,o2,d1/o3
```

then transferred back to the submit machine will be

```
o1
o2
o3
```

Note that only the base name is used in the naming and placement of the file specified with `d1/o3`.

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 job, 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 current working directory as the job is submitted, 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, HTCondor 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 remote scratch directory on the execute machine, in which the program runs. Therefore, the executing program must access input files relative to its working directory. Because all files and directories listed for transfer are placed into a single, flat directory, inputs must be uniquely named to avoid collision when transferred. A collision causes the last file in the list to overwrite the earlier one.

Both relative and absolute paths may be used in **transfer_output_files**. Relative paths are relative to the job's remote scratch directory on the execute machine. When the files and directories are copied back to the submit machine, they are placed in the job's initial working directory as the base name of the original path. An alternate name or path may be specified by using **transfer_output_remaps**.

A job may create files outside the remote scratch directory but within 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, HTCondor will not automatically transfer such files back after execution completes, nor will it clean up these files.

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`.

Here is the directory tree as it exists on the submit machine, for all the 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 first example explicitly transfers input files. These input files to be transferred are specified relative to the directory where the job is submitted. An output file specified in the **arguments** command, `out1`, is created when the job is executed. It will be transferred back into the directory `/scratch/test`.

```
# file name:  my_program.condor
# HTCondor 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
```

The log file is written on the submit machine, and is not involved with the file transfer mechanism.

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
# HTCondor 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**, **error**, **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
# HTCondor 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 the single, flat, remote scratch directory on the execute machine.

```
# file name:  my_program.condor
# HTCondor 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
# HTCondor 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 This example illustrates a case where the executing program creates an output file in a directory other than within the remote scratch 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.

The output file `/tmp/out1` is transferred back to the job's initial working directory as `/scratch/test/out1`.

```
# file name:  my_program.condor
# HTCondor 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
```

Behavior for Error Cases

This section describes HTCondor's behavior for some error cases in dealing with the transfer of files.

Disk Full on Execute Machine When transferring any files from the submit machine to the remote scratch directory, if the disk is full on the execute machine, then the job is place on hold.

Error Creating Zero-Length Files on Submit Machine As a job is submitted, HTCondor creates zero-length files as placeholders on the submit machine for the files defined by **output** and **error**. If these files cannot be created, then job submission fails.

This job submission failure avoids having the job run to completion, only to be unable to transfer the job's output due to permission errors.

Error When Transferring Files from Execute Machine to Submit Machine When a job exits, or potentially when a job is evicted from an execute machine, one or more files may be transferred from the execute machine back to the machine on which the job was submitted.

During transfer, if any of the following three similar types of errors occur, the job is put on hold as the error occurs.

1. If the file cannot be opened on the submit machine, for example because the system is out of inodes.
2. If the file cannot be written on the submit machine, for example because the permissions do not permit it.
3. If the write of the file on the submit machine fails, for example because the system is out of disk space.

File Transfer Using a URL

Instead of file transfer that goes only between the submit machine and the execute machine, HTCondor has the ability to transfer files from a location specified by a URL for a job's input file, or from the execute machine to a location specified by a URL for a job's output file(s). This capability requires administrative set up, as described in section 3.14.2.

The transfer of an input file is restricted to vanilla and vm universe jobs only. HTCondor's file transfer mechanism must be enabled. Therefore, the submit description file for the job will define both **should_transfer_files** and **when_to_transfer_output**. In addition, the URL for any files specified with a URL are given in the **transfer_input_files** command. An example portion of the submit description file for a job that has a single file specified with a URL:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = http://www.full.url/path/to/filename
```

The destination file is given by the file name within the URL.

For the transfer of the entire contents of the output sandbox, which are all files that the job creates or modifies, HTCondor's file transfer mechanism must be enabled. In this sample portion of the submit description file, the first two commands explicitly enable file transfer, and the added **output_destination** command provides both the protocol to be used and the destination of the transfer.

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
output_destination = urltype://path/to/destination/directory
```

Note that with this feature, no files are transferred back to the submit machine. This does not interfere with the streaming of output.

If only a subset of the output sandbox should be transferred, the subset is specified by further adding a submit command of the form:

```
transfer_output_files = file1, file2
```

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.FileSystem` In addition, HTCondor automatically adds the `FileSystemDomain` attribute to the job ClassAd, 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, HTCondor 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 HTCondor 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, HTCondor automatically adds the `DiskUsage` job attribute. This attribute includes the total size of the job's executable and all input files to be transferred. HTCondor 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 is 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 value, 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 floating point speed among the machines in the file system domain, weight the part of the expression that is matching the file system domains. For example:

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

2.5.10 Environment Variables

The environment under which a job executes often contains information that is potentially useful to the job. HTCondor 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 within the *condor_submit* manual page at section 11 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, as described at section 11.

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.

HTCondor 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 HTCondor 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 3.7.1 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 HTCondor version 6.9.3, this environment variable is no longer used. It will only be defined if the `ALLOW_VM_CRUFT` configuration variable is set to `True`.
- `X509_USER_PROXY` gives the full path to the X.509 user proxy file if one is associated with the job. Typically, a user will specify **x509userproxy** in the submit description file. This setting is currently available in the local, java, and vanilla universes.
- `_CONDOR_JOB_AD` is the path to a file in the job's scratch directory which contains the job ad for the currently running job. The job ad is current as of the start of the job, but is not updated during the running of the job. The job may read attributes and their values out of this file as it runs, but any changes will not be acted on in any way by HTCondor. The format is the same as the output of the *condor_q* **-l** command. This environment variable may be particularly useful in a USER_JOB_WRAPPER.
- `_CONDOR_MACHINE_AD` is the path to a file in the job's scratch directory which contains the machine ad for the slot the currently running job is using. The machine ad is current as of the start of the job, but is not updated during the running of the job. The format is the same as the output of the *condor_status* **-l** command.

- _CONDOR_JOB_IWD is the path to the initial working directory the job was born with.
- _CONDOR_WRAPPER_ERROR_FILE is only set when the administrator has installed a USER_JOB_WRAPPER. If this file exists, HTCondor assumes that the job wrapper has failed and copies the contents of the file to the StarterLog for the administrator to debug the problem.
- CONDOR_IDS overrides the value of configuration variable CONDOR_IDS, when set in the environment.
- CONDOR_ID is set for scheduler universe jobs to be the same as the ClusterId attribute.

2.5.11 Heterogeneous Submit: Execution on Differing Architectures

If executables are available for the different platforms of machines in the HTCondor pool, HTCondor 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 32-bit Intel processor running Windows Vista, submitted from an Intel architecture running Linux would add the requirement

```
requirements = Arch == "INTEL" && OpSys == "WINDOWS"
```

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 5.3.10 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 HTCondor 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 HTCondor 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 two platforms: RHEL 3 on both 32-bit and 64-bit Intel processors. 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.LINUX.X86_64
```

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 HTCondor. 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 == "X86_64" && OpSys =="LINUX")

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 HTCondor 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, HTCondor reports 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 HTCondor, the author of the heterogeneous submission file would need to write extra policy expressions in the `requirements` expression to force HTCondor 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. HTCondor 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 HTCondor
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)

# HTCondor automatically adds the correct expressions to insure that the
# checkpointed jobs will restart on the correct platform types.
Requirements = ( (Arch == "INTEL" && OpSys == "LINUX") || \
                 (Arch == "X86_64" && OpSys == "LINUX") )

Arguments    = +W1024 +H768 +Iimage1.pov
Queue

Arguments    = +W1024 +H768 +Iimage2.pov
Queue

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

Vanilla Universe Example for Execution on Differing Operating Systems

The addition of several related OpSys attributes assists in selection of specific operating systems and versions in heterogeneous pools.

```
####################
#
# Example of submission targeting RedHat platforms in a heterogeneous Linux pool
#
####################

universe     = vanilla
Executable   = /bin/date
Log          = distro.log
Output       = distro.out
Error        = distro.err

Requirements = (OpSysName == "RedHat")

Queue


####################
#
# Example of submission targeting RedHat 6 platforms in a heterogeneous Linux pool
#
####################
```

```
universe     = vanilla
Executable   = /bin/date
Log          = distro.log
Output       = distro.out
Error        = distro.err

Requirements = ( OpSysName == "RedHat" && OpSysMajorVer == 6)

Queue
```

Here is a more compact way to specify a RedHat 6 platform.

```
####################
#
# Example of submission targeting RedHat 6 platforms in a heterogeneous Linux pool
#
####################

universe     = vanilla
Executable   = /bin/date
Log          = distro.log
Output       = distro.out
Error        = distro.err

Requirements = ( OpSysAndVer == "RedHat6")

Queue
```

2.5.12 Jobs That Require GPUs

A job that needs GPUs to run identifies the number of GPUs needed in the submit description file by adding the submit command

```
request_GPUs = <n>
```

where <n> is replaced by the integer quantity of GPUs required for the job. For example, a job that needs 1 GPU uses

```
request_GPUs = 1
```

Because there are different capabilities among GPUs, the job might need to further qualify which GPU of available ones is required. Do this by specifying or adding a clause to an existing **Requirements** submit command. As an example, assume that the job needs a speed and capacity of a CUDA GPU that meets or exceeds the value 1.2. In the submit description file, place

```
request_GPUs = 1
requirements = (CUDACapability >= 1.2) && $(requirements:True)
```

Access to GPU resources by an HTCondor job needs special configuration of the machines that offer GPUs. Details of how to set up the configuration are in section 3.7.1.

2.5.13 Interactive Jobs

An *interactive job* is a Condor job that is provisioned and scheduled like any other vanilla universe Condor job onto an execute machine within the pool. The result of a running interactive job is a shell prompt issued on the execute machine where the job runs. The user that submitted the interactive job may then use the shell as desired, perhaps to interactively run an instance of what is to become a Condor job. This might aid in checking that the set up and execution environment are correct, or it might provide information on the RAM or disk space needed. This job (shell) continues until the user logs out or any other policy implementation causes the job to stop running. A useful feature of the interactive job is that the users and jobs are accounted for within Condor's scheduling and priority system.

Neither the submit nor the execute host for interactive jobs may be on Windows platforms.

The current working directory of the shell will be the initial working directory of the running job. The shell type will be the default for the user that submits the job. At the shell prompt, X11 forwarding is enabled.

Each interactive job will have a job ClassAd attribute of

```
InteractiveJob = True
```

Submission of an interactive job specifies the option **-interactive** on the *condor_submit* command line.

A submit description file may be specified for this interactive job. Within this submit description file, a specification of these 5 commands will be either ignored or altered:

1. **executable**
2. **transfer_executable**
3. **arguments**
4. **universe**. The interactive job is a vanilla universe job.
5. **queue <n>**. In this case the value of **<n>** is ignored; exactly one interactive job is queued.

The submit description file may specify anything else needed for the interactive job, such as files to transfer.

If *no* submit description file is specified for the job, a default one is utilized as identified by the value of the configuration variable `INTERACTIVE_SUBMIT_FILE`.

Here are examples of situations where interactive jobs may be of benefit.

- An application that cannot be batch processed might be run as an interactive job. Where input or output cannot be captured in a file and the executable may not be modified, the interactive nature of the job may still be run on a pool machine, and within the purview of Condor.
- A pool machine with specialized hardware that requires interactive handling can be scheduled with an interactive job that utilizes the hardware.
- The debugging and set up of complex jobs or environments may benefit from an interactive session. This interactive session provides the opportunity to run scripts or applications, and as errors are identified, they can be corrected on the spot.
- Development may have an interactive nature, and proceed more quickly when done on a pool machine. It may also be that the development platforms required reside within Condor's purview as execute hosts.

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 11 beginning on page 734) for more information.

When jobs are submitted, HTCondor 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: submit.chtc.wisc.edu : <128.104.55.9:32772> : submit.chtc.wisc.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
711197.0   aragorn         1/15 19:18   0+04:29:33 H  0   0.0  script.sh
894381.0   frodo           3/16 09:06  82+17:08:51 R  0   439.5 elk elk.in
894386.0   frodo           3/16 09:06  82+20:21:28 R  0   219.7 elk elk.in
894388.0   frodo           3/16 09:06  81+17:22:10 R  0   439.5 elk elk.in
1086870.0   gollum          4/27 09:07   0+00:10:14 I  0   7.3  condor_dagman
1086874.0   gollum          4/27 09:08   0+00:00:01 H  0   0.0  RunDC.bat
1297254.0   legolas         5/31 18:05  14+17:40:01 R  0   7.3  condor_dagman
1297255.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297256.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297259.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1297261.0   legolas         5/31 18:05  14+17:39:55 R  0   7.3  condor_dagman
1302278.0   legolas         6/4  12:22   1+00:05:37 I  0   390.6 mdrun_1.sh
1304740.0   legolas         6/5  00:14   1+00:03:43 I  0   390.6 mdrun_1.sh
1304967.0   legolas         6/5  05:08   0+00:00:00 I  0   0.0  mdrun_1.sh

14 jobs; 4 idle, 8 running, 2 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:

`R`: The job is currently running.

`I`: The job is idle. It is not running right now, because it is waiting for a machine to become available.

`H`: The job is the hold state. In the hold state, the job will not be scheduled to run until it is released. See the *condor_hold* manual page located on page 799 and the *condor_release* manual page located on page 854.

The `RUN_TIME` time reported for a job is the time that has been committed to the job.

Another useful method of tracking the progress of jobs is through the job event log. The specification of a `log` in the submit description file causes the progress of the job to be logged in a file. Follow the events by viewing the job event 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 a job begins to run, HTCondor 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 *condor_shadow* processes 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 there is degraded performance, limit the number of jobs that can run simultaneously by reducing the `MAX_JOBS_RUNNING` configuration variable.

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    LINUX        Claimed    Busy       0.980  64    0+07:10:02
biron.cs.w INTEL    LINUX        Claimed    Busy       1.000  128   0+01:10:00
cambridge. INTEL    LINUX        Claimed    Busy       0.988  64    0+00:15:00
falcons.cs INTEL    LINUX        Claimed    Busy       0.996  32    0+02:05:03
happy.cs.w INTEL    LINUX        Claimed    Busy       0.988  128   0+03:05:00
istat03.st INTEL    LINUX        Claimed    Busy       0.883  64    0+06:45:01
istat04.st INTEL    LINUX        Claimed    Busy       0.988  64    0+00:10:00
istat09.st INTEL    LINUX        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    LINUX        0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    LINUX        0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi X86_64   LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
anfrom.cs. X86_64   LINUX        1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    LINUX        0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    LINUX        1.000  nice-user.condor@cs. chevre.cs.wisc.
aura.cs.wi X86_64   WINDOWS      0.996  nice-user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    WINDOWS      1.000  nice-user.condor@cs. chevre.cs.wisc.
bamba.cs.w INTEL    LINUX        1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    LINUX        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 799 and the manual page for *condor_release* on page 854 contain usage details.

2.6.4 Changing the priority of jobs

In addition to the priorities assigned to each user, HTCondor 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 HTCondor. 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 is the job not running?

Users occasionally find that their jobs do not run. There are many possible reasons why a specific job is not running. The following prose attempts to identify some of the potential issues behind why a job is not running.

At the most basic level, the user knows the status of a job by using *condor_q* to see that the job is not running. By far, the most common reason (to the novice HTCondor job submitter) why the job is not running is that HTCondor has not yet been through its periodic negotiation cycle, in which queued jobs are assigned to machines within the pool and begin their execution. This periodic event occurs by default once every 5 minutes, implying that the user ought to wait a few minutes before searching for reasons why the job is not running.

Further inquiries are dependent on whether the job has never run at all, or has run for at least a little bit.

For jobs that have never run, many problems can be diagnosed by using the **-analyze** option of the *condor_q* command. Here is an example; running *condor_q*'s analyzer provided the following information:

```
$ condor_q -analyze 27497829

-- Submitter: submit-1.chtc.wisc.edu : <128.104.100.43:9618?sock=5557_e660_3> : submit-1.chtc.wisc.edu
User priority for einstein@submit.chtc.wisc.edu is not available, attempting to analyze without it.
---
27497829.000:  Run analysis summary.  Of 5257 machines,
   5257 are rejected by your job's requirements
      0 reject your job because of their own requirements
      0 match and are already running your jobs
      0 match but are serving other users
      0 are available to run your job
        No successful match recorded.
        Last failed match: Tue Jun 18 14:36:25 2013

        Reason for last match failure: no match found

WARNING:  Be advised:
   No resources matched request's constraints

The Requirements expression for your job is:

    ( OpSys == "OSX" ) && ( TARGET.Arch == "X86_64" ) &&
    ( TARGET.Disk >= RequestDisk ) && ( TARGET.Memory >= RequestMemory ) &&
    ( ( TARGET.HasFileTransfer ) || ( TARGET.FileSystemDomain == MY.FileSystemDomain ) )


Suggestions:
    Condition                         Machines Matched Suggestion
    ---------                         ---------------- ----------
1   ( target.OpSys == "OSX" )         0                MODIFY TO "LINUX"
2   ( TARGET.Arch == "X86_64" )       5190
3   ( TARGET.Disk >= 1 )              5257
4   ( TARGET.Memory >= ifthenelse(MemoryUsage isnt undefined,MemoryUsage,1) )
                                      5257
5   ( ( TARGET.HasFileTransfer ) || ( TARGET.FileSystemDomain == "submit-1.chtc.wisc.edu" ) )
                                      5257
```

This example also shows that the job does not run because the platform requested, Mac OS X, is not available on

any of the machines in the pool. Recall that unless informed otherwise in the **Requirements** expression in the submit description file, the platform requested for an execute machine will be the same as the platform where *condor_submit* is run to submit the job. And, while Mac OS X is a Unix-type operating system, it is not the same as Linux, and thus will not match with machines running Linux.

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 has not run. In most of these situations, the delay is transient, and the job will run following the next negotiation cycle.

A second class of problems represents jobs that do or did run, for at least a short while, but are no longer running. The first issue is identifying whether the job is in this category. The *condor_q* command is not enough; it only tells the current state of the job. The needed information will be in the **log** file or the **error** file, as defined in the submit description file for the job. If these files are not defined, then there is little hope of determining if the job ran at all. For a job that ran, even for the briefest amount of time, the **log** file will contain an event of type 1, which will contain the string `Job executing on host`.

A job may run for a short time, before failing due to a file permission problem. The log file used by the *condor_shadow* daemon will contain more information if this is the problem. This log file is associated with the machine on which the job was submitted. The location and name of this log file may be discovered on the submitting machine, using the command

```
%  condor_config_val SHADOW_LOG
```

Memory and swap space problems may be identified by looking at the log file used by the *condor_schedd* daemon. The location and name of this log file may be discovered on the submitting machine, using the command

```
%  condor_config_val SCHEDD_LOG
```

A swap space problem will show in the log with the following message:

```
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
```

As an explanation, HTCondor computes the total swap space on the 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, and 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. HTCondor does not consider the physical memory size, so the situation occurs where HTCondor thinks it has no swap space to work with, and it will not run the submitted jobs.

To see how much swap space HTCondor 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 HTCondor. There are two ways to fix the problem:

1. Configure the machine with some real swap space.
2. Disable this check within HTCondor. 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.

2.6.6 Job in the Hold State

A variety of errors and unusual conditions may cause a job to be placed into the Hold state. The job will stay in this state and in the job queue until conditions are corrected and *condor_release* is invoked.

A table listing the reasons why a job may be held is at section 12. A string identifying the reason that a particular job is in the Hold state may be displayed by invoking *condor_q*. For the example job ID 16.0, use:

```
condor_q -hold 16.0
```

This command prints information about the job, including the job ClassAd attribute `HoldReason`.

2.6.7 In the Job Event Log File

In a job event log file are a listing of events in chronological order that occurred during the life of one or more jobs. 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 job ClassAd attributes of `ClusterId` value, `ProcId` value, and the node number for parallel universe jobs or a set of zeros (for jobs run under all other universes), 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 could not 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: An informational event, to update the amount of 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, such as 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 job has transitioned to the hold state. This might happen if the user applies the *condor_hold*

command to the job.

Event Number: 013
Event Name: Job was released
Event Description: The job was in the hold state and is to be re-run.

Event Number: 014
Event Name: Parallel node executed
Event Description: A parallel universe program is running on a node.

Event Number: 015
Event Name: Parallel node terminated
Event Description: A parallel universe 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). This event is no longer used.

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. This event is no longer used.

Event Number: 020
Event Name: Detected Down Globus Resource
Event Description: The Globus resource that a job wants to run on has become unavailable. This event is no longer used.

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 ClassAd attributes are noted. This event is written as a supplement to other events when the configuration parameter `EVENT_LOG_JOB_AD_INFORMATION_ATTRS` is set.

Event Number: 029
Event Name: The job's remote status is unknown
Event Description: No updates of the job's remote status have been received for 15 minutes.

Event Number: 030
Event Name: The job's remote status is known again
Event Description: An update has been received for a job whose remote status was previous logged as unknown.

Event Number: 031
Event Name: Job stage in
Event Description: A grid universe job is doing the stage in of input files.

Event Number: 032
Event Name: Job stage out
Event Description: A grid universe job is doing the stage out of output files.

Event Number: 033
Event Name: Job ClassAd attribute update
Event Description: A Job ClassAd attribute is changed due to action by the *condor_schedd* daemon. This includes changes by *condor_prio*.

Event Number: 034
Event Name: Pre Skip event
Event Description: For DAGMan, this event is logged if a PRE SCRIPT exits with the defined PRE_SKIP value in the DAG input file. This makes it possible for DAGMan to do recovery in a workflow that has such an event, as it would otherwise not have any event for the DAGMan node to which the script belongs, and in recovery, DAGMan's internal tables would become corrupted.

2.6.8 Job Completion

When an HTCondor job completes, either through normal means or by abnormal termination by signal, HTCondor will remove it from the job queue. That is, the job will no longer appear in the output of *condor_q*, and the job will be inserted into the job history file. Examine the job history file with the *condor_history* command. If there is a log file specified in the submit description file for the job, then the job exit status will be recorded there as well.

By default, HTCondor does not send an email message when the job completes. Modify this behavior with the **notification** command in the submit description file. The message will include the exit status of the job, which is the argument that the job passed to the exit system call when it completed, or it will be notification that the job was killed by a signal. Notification will also include the following statistics (as appropriate) about the job:

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

Completed at: when the job completed

Real Time: the elapsed time between when the job was submitted and when it completed, given in a form of `<days> <hours>:<minutes>:<seconds>`

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

Statistics about just the last time the job ran:

Run Time: total time the job was running, given in the form `<days> <hours>:<minutes>:<seconds>`

Remote User Time: total CPU time the job spent executing in user mode on remote machines; this does not count time spent on run attempts that were evicted without a checkpoint. Given in the form `<days> <hours>:<minutes>:<seconds>`

Remote System Time: total CPU time the job spent executing in system mode (the time spent at system calls); this does not count time spent on run attempts that were evicted without a checkpoint. Given in the form `<days> <hours>:<minutes>:<seconds>`

The Run Time accumulated by all run attempts are summarized with the time given in the form `<days> <hours>:<minutes>:<seconds>`.

And, statistics about the bytes sent and received by the last run of the job and summed over all attempts at running the job are given.

2.7 Priorities and Preemption

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

2.7.1 Job Priority

Job priorities allow a user to assign a priority level to each of their own submitted HTCondor jobs, in order to control the order of job execution. This handles the situation in which a user has more jobs queued, waiting to be executed, than there are machines available. Setting a job priority identifies the ordering in which that user's jobs are executed; a higher job priority job is matched and executed before a lower priority job. A job priority can be any integer, and larger values are of higher priority. So, 0 is a higher job priority than -3, and 6 is a higher job priority than 5.

For the simple case, each job can be given a distinct priority. For an already queued job, its priority may be set with the *condor_prio* command; see the example in section 2.6.4, or the *condor_prio* manual page 824 for details. This sets the value of job ClassAd attribute `JobPrio`.

A fine-grained categorization of jobs and their ordering is available for experts by using the job ClassAd attributes: `PreJobPrio1`, `PreJobPrio2`, `JobPrio`, `PostJobPrio1`, or `PostJobPrio2`.

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 HTCondor can be examined with the *condor_userprio* command (see page 964). HTCondor administrators can set and change individual user priorities with the same utility.

HTCondor 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 then 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 HTCondor adjusts the priorities is controlled with the configuration variable `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.

HTCondor 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, HTCondor will immediately take a checkpoint and vacate jobs belonging to the lower priority user. This will free up machines that HTCondor will then give over to the higher priority user. HTCondor 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 HTCondor site administrator can define a `PREEMPTION_REQUIREMENTS` expression in HTCondor's configuration. The default expression that ships with HTCondor 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 a fair share of machines. For a general discussion of limiting preemption, please see section 3.7.1 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 HTCondor 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 927). Nice jobs artificially boost the user priority by ten million just for the nice job. This effectively means that nice jobs will only run on machines that no other HTCondor job (that is, non-niced job) wants. In a similar fashion, an HTCondor administrator could set the user priority of any specific HTCondor 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 HTCondor Jobs Vacate Machines

When HTCondor 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 HTCondor 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, HTCondor 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 HTCondor libraries:

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

`ckpt_and_exit()()` Checkpoints and exits; HTCondor 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

HTCondor 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 HTCondor Java universe brings together these two features to create a distributed, homogeneous computing environment.

Compiled Java programs can be submitted to HTCondor, and HTCondor 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 HTCondor can use the Java Virtual Machine.

```
% condor_status -java

Name               JavaVendor Ver    State     Activity LoadAv  Mem  ActvtyTime

adelie01.cs.wisc.e Sun Micros 1.6.0_ Claimed   Busy     0.090   873  0+00:02:46
adelie02.cs.wisc.e Sun Micros 1.6.0_ Owner     Idle     0.210   873  0+03:19:32
slot10@bio.cs.wisc Sun Micros 1.6.0_ Unclaimed Idle     0.000   118  7+03:13:28
slot2@bio.cs.wisc. Sun Micros 1.6.0_ Unclaimed Idle     0.000   118  7+03:13:28
...
```

If there is no output from the *condor_status* command, then HTCondor 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 3.15 for more information on getting HTCondor 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 HTCondor 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 2
# 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 HTCondor's file transfer mechanisms, see section 2.5.9.

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
```

HTCondor 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, HTCondor 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 HTCondor 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
```

Discovering the main class within a JAR file. As of Java version 1.4, Java virtual machines have a **-jar** option, which takes a single JAR file as an argument. With this option, the Java virtual machine discovers the main class to run from the contents of the Manifest file, which is bundled within the JAR file. HTCondor's **java** universe does not support this discovery, so before submitting the job, the name of the main class must be identified.

For a Java application which is run on the command line with

```
java -jar OneJarFile.jar
```

the equivalent version after discovery might look like

```
java -classpath OneJarFile.jar TheMainClass
```

The specified value for `TheMainClass` can be discovered by unjarring the JAR file, and looking for the MainClass definition in the Manifest file. Use that definition in the HTCondor submit description file. Partial contents of that file Java universe submit file will appear as

```
universe   = java
executable =  OneJarFile.jar
jar_files = OneJarFile.jar
Arguments = TheMainClass More-Arguments
queue
```

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 HTCondor must only run on those machines within the pool that run the needed JVM. Inform HTCondor 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 an HTCondor pool will execute a benchmark to determine its speed. The benchmark is taken when HTCondor 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 HTCondor 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 -Xmx1024m
```

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 HTCondor 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 HTCondor'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 HTCondor 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 HTCondor 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 HTCondor 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
+WantIOProxy = True
queue
```

2.9 Parallel Applications (Including MPI Applications)

HTCondor's parallel universe supports jobs that span multiple machines, where the multiple processes within a job must be running concurrently on these multiple machines, perhaps communicating with each other. The parallel universe provides machine scheduling, but does not enforce a particular programming paradigm for the underlying applications. Thus, parallel universe jobs may run under various MPI implementations as well as under other programming environments.

The parallel universe supersedes the mpi universe. The mpi universe eventually will be removed from HTCondor.

2.9.1 How Parallel Jobs Run

Parallel universe jobs are submitted from the machine running the dedicated scheduler. The dedicated scheduler matches and claims a fixed number of machines (slots) for the parallel universe job, and when a sufficient number of machines are claimed, the parallel job is started on each claimed slot.

Each invocation of *condor_submit* assigns a single `ClusterId` for what is considered the single parallel job submitted. The **machine_count** submit command identifies how many machines (slots) are to be allocated. Each instance of the **queue** submit command acquires and claims the number of slots specified by **machine_count**. Each of these slots shares a common job ClassAd and will have the same `ProcId` job ClassAd attribute value.

Once the correct number of machines are claimed, the **executable** is started at more or less the same time on all machines. If desired, a monotonically increasing integer value that starts at 0 may be provided to each of these machines. The macro `$(Node)` is similar to the MPI *rank* construct. This macro may be used within the submit description file in either the **arguments** or **environment** command. Thus, as the executable runs, it may discover its own `$(Node)` value.

Node 0 has special meaning and consequences for the parallel job. The completion of a parallel job is implied and taken to be when the Node 0 executable exits. All other nodes that are part of the parallel job and that have not yet exited on their own are killed. This default behavior may be altered by placing the line

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

in the submit description file. It causes HTCondor to wait until every node in the parallel job has completed to consider the job finished.

2.9.2 Parallel Jobs and the Dedicated Scheduler

To run parallel universe jobs, HTCondor must be configured such that machines running parallel jobs are *dedicated*. Note that dedicated has a very specific meaning in HTCondor: while dedicated machines can run serial jobs, they prefer to run parallel jobs, and dedicated machines never preempt a parallel job once it starts running.

A machine becomes a dedicated machine when an administrator configures it to accept parallel jobs from one specific dedicated scheduler. Note the difference between parallel and serial jobs. While any scheduler in a pool can send serial jobs to any machine, only the designated dedicated scheduler may send parallel universe jobs to a dedicated machine. Dedicated machines must be specially configured. See section 3.14.8 for a description of the necessary configuration, as well as examples. Usually, a single dedicated scheduler is configured for a pool which can run parallel universe jobs, and this *condor_schedd* daemon becomes the single machine from which parallel universe jobs are submitted.

The following command line will list the execute machines in the local pool which have been configured to use a dedicated scheduler, also printing the name of that dedicated scheduler. In order to run parallel jobs, this name will be defined to be the string `"DedicatedScheduler@"`, prepended to the name of the scheduler host.

```
  condor_status -const '!isUndefined(DedicatedScheduler)' \
-format "%s\t" Machine -format "%s\n" DedicatedScheduler

execute1.example.com DedicatedScheduler@submit.example.com
execute2.example.com DedicatedScheduler@submit.example.com
```

If this command emits no lines of output, then then pool is not correctly configured to run parallel jobs. Make sure that the name of the scheduler is correct. The string after the `@` sign should match the name of the *condor_schedd* daemon, as returned by the command

```
  condor_status -schedd
```

2.9.3 Submission Examples

Simplest Example

Here is a submit description file for a parallel universe job example that is as simple as possible:

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

```
log = log
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
queue
```

This job specifies the **universe** as **parallel**, letting HTCondor know that dedicated resources are required. The **machine_count** command identifies that eight machines are required for this job.

Because no **requirements** are specified, the dedicated scheduler claims eight machines with the same architecture and operating system as the submit machine. When all the machines are ready, it invokes the */bin/sleep* command, with a command line argument of 30 on each of the eight machines more or less simultaneously. Job events are written to the log specified in the **log** command.

The file transfer mechanism is enabled for this parallel job, such that if any of the eight claimed execute machines does not share a file system with the submit machine, HTCondor will correctly transfer the executable. This */bin/sleep* example implies that the submit machine is running a Unix operating system, and the default assumption for submission from a Unix machine would be that there is a shared file system.

Example with Operating System Requirements

Assume that the pool contains Linux machines installed with either a RedHat or an Ubuntu operating system. If the job should run only on RedHat platforms, the requirements expression may specify this:

```
##############################################
##  submit description file for a parallel program
##  targeting RedHat machines
##############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
log = log
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
requirements = (OpSysName == "RedHat")
queue
```

The machine selection may be further narrowed, instead using the `OpSysAndVer` attribute.

```
##############################################
##  submit description file for a parallel program
##  targeting RedHat 6 machines
##############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
log = log
should_transfer_files = IF_NEEDED
```

```
when_to_transfer_output = ON_EXIT
requirements = (OpSysAndVer == "RedHat6")
queue
```

Using the `$(Node)` Macro

```
######################################
## submit description file for a parallel program
## showing the $(Node) macro
######################################
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(Node)
output = outfile.$(Node)
error = errfile.$(Node)
machine_count = 4
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
queue
```

The `$(Node)` macro is expanded to values of 0-3 as the job instances are about to be started. This assigns unique names to the input and output files to be transferred or accessed from the shared file system. The `$(Node)` value is fixed for the entire length of the job.

Differing Requirements for the Machines

Sometimes one machine's part in a parallel job will have specialized needs. These can be handled with a **Requirements** submit command that also specifies the number of needed machines.

```
######################################
## Example submit description file
## with 4 total machines and differing requirements
######################################
universe = parallel
executable = special.exe
machine_count = 1
requirements = ( machine == "machine1@example.com")
queue

machine_count = 3
requirements = ( machine =!= "machine1@example.com")
queue
```

The dedicated scheduler acquires and claims four machines. All four share the same value of `ClusterId`, as this value is associated with this single parallel job. The existence of a second **queue** command causes a total of two `ProcId` values to be assigned for this parallel job. The `ProcId` values are assigned based on ordering within the submit description file. Value 0 will be assigned for the single executable that must be executed on machine1@example.com, and the value 1 will be assigned for the other three that must be executed elsewhere.

Requesting multiple cores per slot

If the parallel program has a structure that benefits from running on multiple cores within the same slot, multi-core slots may be specified.

```
######################################
## submit description file for a parallel program
## that needs 8-core slots
######################################
universe = parallel
executable = foo.sh
log = logfile
input = infile.$(Node)
output = outfile.$(Node)
error = errfile.$(Node)
machine_count = 2
request_cpus = 8
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT
queue
```

This parallel job causes the scheduler to match and claim two machines, where each of the machines (slots) has eight cores. The parallel job is assigned a single `ClusterId` and a single `ProcId`, meaning that there is a single job ClassAd for this job.

The executable, `foo.sh`, is started at the same time on a single core within each of the two machines (slots). It is presumed that the executable will take care of invoking processes that are to run on the other seven CPUs (cores) associated with the slot.

Potentially fewer machines are impacted with this specification, as compared with the request that contains

```
machine_count = 16
request_cpus = 1
```

The interaction of the eight cores within the single slot may be advantageous with respect to communication delay or memory access. But, 8-core slots must be available within the pool.

MPI Applications

MPI applications use a single executable, invoked on one or more machines (slots), executing in parallel. The various implementations of MPI such as Open MPI and MPICH require further framework. HTCondor supports this necessary framework through a user-modified script. This implementation-dependent script becomes the HTCondor executable. The script sets up the framework, and then it invokes the MPI application's executable.

The scripts are located in the `$(RELEASE_DIR)/etc/examples` directory. The script for the Open MPI implementation is `openmpiscript`. The scripts for MPICH implementations are `mp1script` and `mp2script`. An MPICH3 script is not available at this time. These scripts rely on running *ssh* for communication between the nodes of the MPI application. The *ssh* daemon on Unix platforms restricts connections to the approved shells listed in the `/etc/shells` file.

Here is a sample submit description file for an MPICH MPI application:

```
####################################
## Example submit description file
## for MPICH 1 MPI
## works with MPICH 1.2.4, 1.2.5 and 1.2.6
####################################
universe = parallel
executable = mp1script
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
```

The **executable** is the `mp1script` script that will have been modified for this MPI application. This script is invoked on each slot or core. The script, in turn, is expected to invoke the MPI application's executable. To know the MPI application's executable, it is the first in the list of **arguments**. And, since HTCondor must transfer this executable to the machine where it will run, it is listed with the **transfer_input_files** command, and the file transfer mechanism is enabled with the **should_transfer_files** command.

Here is the equivalent sample submit description file, but for an Open MPI application:

```
####################################
## Example submit description file
## for Open MPI
####################################
universe = parallel
executable = openmpiscript
arguments = my_openmpi_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_openmpi_linked_executable
queue
```

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. 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. Use of the *sshd.sh* script requires the definition of two HTCondor 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.

mp1script and *mp2script* require the `PATH` to the MPICH installation to be set. The variable `MPDIR` may be modified in the scripts to indicate its proper value. This directory contains the MPICH *mpirun* executable.

openmpiscript also requires the `PATH` to the Open MPI installation. Either the variable `MPDIR` can be set manually in the script, or the administrator can define `MPDIR` using the configuration variable `OPENMPI_INSTALL_PATH`.

When using Open MPI on a multi-machine HTCondor cluster, the administrator may also want to consider tweaking the `OPENMPI_EXCLUDE_NETWORK_INTERFACES` configuration variable as well as set `MOUNT_UNDER_SCRATCH = /tmp`.

2.9.4 MPI Applications Within HTCondor's Vanilla Universe

The vanilla universe may be preferred over the parallel universe for certain parallel applications such as MPI ones. These applications are ones in which the allocated cores need to be within a single slot. The **request_cpus** command causes a claimed slot to have the required number of CPUs (cores).

There are two ways to ensure that the MPI job can run on any machine that it lands on:

1. Statically build an MPI library and statically compile the MPI code.
2. Use CDE to create a directory tree that contains all of the libraries needed to execute the MPI code.

For Linux machines, our experience recommends using CDE, as building static MPI libraries can be difficult. CDE can be found at http://www.pgbovine.net/cde.html.

Here is a submit description file example assuming that MPI is installed on all machines on which the MPI job may run, or that the code was built using static libraries and a static version of `mpirun` is available.

```
############################################################
##   submit description file for
##   static build of MPI under the vanilla universe
############################################################
universe = vanilla
executable = /path/to/mpirun
request_cpus = 2
arguments = -np 2 my_mpi_linked_executable arg1 arg2 arg3
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_mpi_linked_executable
queue
```

If CDE is to be used, then CDE needs to be run first to create the directory tree. On the host machine which has the original program, the command

```
prompt-> cde mpirun -n 2 my_mpi_linked_executable
```

creates a directory tree that will contain all libraries needed for the program. By creating a tarball of this directory, the user can package up the executable itself, any files needed for the executable, and all necessary libraries. The following example assumes that the user has created a tarball called `cde_my_mpi_linked_executable.tar` which contains the directory tree created by CDE.

```
############################################################
```

```
##   submit description file for
##   MPI under the vanilla universe; CDE used
############################################################
universe = vanilla
executable = cde_script.sh
request_cpus = 2
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = cde_my_mpi_linked_executable.tar
transfer_output_files = cde-package/cde-root/path/to/original/directory
queue
```

The executable is now a specialized shell script tailored to this job. In this example, *cde_script.sh* contains:

```
#!/bin/sh
# Untar the CDE package
tar xpf cde_my_mpi_linked_executable.tar
# cd to the subdirectory where I need to run
cd cde-package/cde-root/path/to/original/directory
# Run my command
./mpirun.cde -n 2 ./my_mpi_linked_executable
# Since HTCondor will transfer the contents of this directory
# back upon job completion.
# We do not want the .cde command and the executable transferred back.
# To prevent the transfer, remove both files.
rm -f mpirun.cde
rm -f my_mpi_linked_executable
```

Any additional input files that will be needed for the executable that are not already in the tarball should be included in the list in **transfer_input_files** command. The corresponding script should then also be updated to move those files into the directory where the executable will be run.

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. HTCondor 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 HTCondor in an order represented by a DAG and processes the results. A *DAG input file* describes the DAG.

DAGMan is itself executed as a scheduler universe job within HTCondor. It submits the HTCondor jobs within nodes in such a way as to enforce the DAG's dependencies. DAGMan also handles recovery and reporting on the HTCondor jobs.

2.10.1 DAGMan Terminology

A node within a DAG may encompass more than a single program submitted to run under HTCondor. Figure 2.1 illustrates the elements of a node.

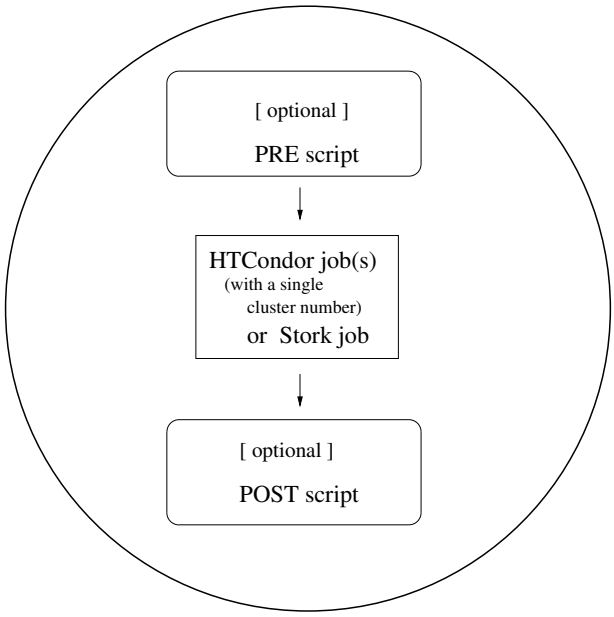

Figure 2.1: One Node within a DAG

More than one HTCondor job may belong to a single node. All HTCondor jobs within a node must be within a single cluster, as given by the job ClassAd attribute `ClusterId`.

DAGMan enforces the dependencies within a DAG using the events recorded in a separate file that is specified by the default configuration. If the exact same DAG were to be submitted more than once, such that these DAGs were running at the same time, expected them to fail in unpredictable and unexpected ways. They would all be using the same single file to enforce dependencies.

As DAGMan schedules and submits jobs within nodes to HTCondor, 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) is based upon the success or failure of 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 may be further constrained to succeed or fail based upon the return values of a PRE script and/or a POST script.

2.10.2 The DAG Input File: Basic Commands

The input file used by DAGMan is called a DAG input file. It specifies the nodes of the DAG as well as the dependencies that order the DAG. All items are optional, except that there must be at least one *JOB* 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 2.2 is presented as a starting point for examples. This DAG contains 4 nodes.

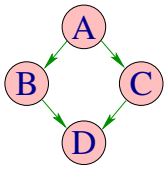

Figure 2.2: Diamond DAG

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 commands appearing in a DAG input file is described below.

JOB

The *JOB* command specifies an HTCondor job. The syntax used for each *JOB* command is

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

A *JOB* entry maps a *JobName* to an HTCondor submit description file. The *JobName* uniquely identifies nodes within the DAG input file and in output messages. Each node name, given by *JobName*, within the DAG must be unique. The *JOB* entry must appear within the DAG input file before other items that reference the node.

The keywords *JOB*, *DIR*, *NOOP*, 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 a 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).

Note that *DIR*, *NOOP*, and *DONE*, if used, must appear in the order shown above.

The optional *DIR* keyword specifies a working directory for this node, from which the HTCondor job will be submitted, and from which a *PRE* and/or *POST* script will be run. If a relative directory is specified, it is relative to the current working directory as the DAG is submitted. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*. A "full" rescue DAG generated by a DAG run with the *-usedagdir* argument will contain DIR specifications, so such a rescue DAG must be run *without* the *-usedagdir* argument. (Note that "full" rescue DAGs are no longer the default.)

The optional *NOOP* keyword identifies that the HTCondor job within the node is not to be submitted to HTCondor. This optimization is useful in cases such as debugging a complex DAG structure, where some of the individual jobs are long-running. For this debugging of structure, some jobs are marked as *NOOP*s, and the DAG is initially run to verify that the control flow through the DAG is correct. The *NOOP* keywords are then removed before submitting the DAG. Any PRE and POST scripts for jobs specified with *NOOP are* executed; to avoid running the PRE and POST scripts, comment them out. The job that is not submitted to HTCondor is given a return value that indicates success, such that the node may also succeed. Return values of any PRE and POST scripts may still cause the node to fail. Even though the job specified with *NOOP* is not submitted, its submit description file must exist; the log file for the job is used, because DAGMan generates dummy submission and termination events for the job.

The optional *DONE* keyword identifies a node as being already completed. This is mainly used by Rescue DAGs generated by DAGMan itself, in the event of a failure to complete the workflow. Nodes with the *DONE* keyword are not executed when the Rescue DAG is run, allowing the workflow to pick up from the previous endpoint. Users should generally not use the *DONE* keyword. The *NOOP* keyword is more flexible in avoiding the execution of a job within a node. Note that, for any node marked *DONE* in a DAG, all of its parents must also be marked *DONE*; otherwise, a fatal error will result. The *DONE* keyword applies to the entire node. A node marked with *DONE* will not have a PRE or POST script run, and the HTCondor job will not be submitted.

DATA

As of version 8.3.5, *condor_dagman* no longer supports DATA nodes.

PARENT . . . CHILD

The *PARENT CHILD* command specifies 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 used for each dependency (PARENT/CHILD) command is

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

The *PARENT* keyword is followed by one or more *ParentJobName*s. The *CHILD* keyword is followed by one or more *ChildJobName*s. 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. The diamond-shaped DAG example may specify the dependencies with

```
PARENT A CHILD B C
```

```
PARENT B C CHILD D
```

An alternative specification for the diamond-shaped DAG may specify some or all of the dependencies on separate lines:

```
PARENT A CHILD B C
PARENT B CHILD D
PARENT C CHILD D
```

As a further 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`

SCRIPT

The optional *SCRIPT* command specifies processing that is done either before a job within a node is submitted or after a job within a node completes its execution. Processing done before a job is submitted is called a *PRE* script. Processing done after a job completes its execution is called a *POST* script. Note that the executable specified does not necessarily have to be a shell script (Unix) or batch file (Windows); but it should be relatively light weight because it will be run directly on the submit machine, not submitted as an HTCondor job.

The syntax used for each *PRE* or *POST* command is

SCRIPT [**DEFER** *status time*] **PRE** *JobName*|**ALL_NODES** *ExecutableName* [*arguments*]

SCRIPT [**DEFER** *status time*] **POST** *JobName*|**ALL_NODES** *ExecutableName* [*arguments*]

The *SCRIPT* command uses the *PRE* or *POST* keyword, which specifies the relative timing of when the script is to be run. The *JobName* identifies the node to which the script is attached. The *ExecutableName* specifies the executable (e.g., shell script or batch file) to be executed, and may not contain spaces. The optional *arguments* are command line arguments to the script, and spaces delimit the arguments. Both *ExecutableName* and optional *arguments* are case sensitive.

Scripts are executed on the submit machine; the submit machine is not necessarily the same machine upon which the node's job is run. Further, a single cluster of HTCondor jobs may be spread across several machines.

The optional *DEFER* feature causes a retry of only the script, if the execution of the script exits with the exit code given by *status*. The retry occurs after at least *time* seconds, rather than being considered failed. While waiting for

the retry, the script does not count against a *maxpre* or *maxpost* limit. The ordering of the *DEFER* feature within the *SCRIPT* specification is fixed. It must come directly after the *SCRIPT* keyword; this is done to avoid backward compatibility issues for any DAG with a *JobName* of DEFER.

A PRE script is commonly used to place files in a staging area for the jobs to use. A POST script is commonly used to clean up or remove files once jobs are 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, uncompresses them, and places the resulting files in the current directory. The HTCondor jobs can then use these files, producing output files. The POST script compresses the output files, writes them out to the tape, and then removes both the staged files and the output files.

If the PRE script fails, then the HTCondor job associated with the node is not submitted, and (as of version 8.5.4) the POST script is not run either (by default). However, if the job is submitted, and there is a POST script, the POST script is always run once the job finishes. (The behavior when the PRE script fails may may be changed to run the POST script by setting configuration variable `DAGMAN_ALWAYS_RUN_POST` to `True` or by passing the **-AlwaysRunPost** argument to *condor_submit_dag*.)

Progress towards completion of the DAG is based upon the success of the nodes within the DAG. The success of a node is based upon the success of the job(s), PRE script, and POST script. A job, PRE script, or POST script with an exit value not equal to 0 is considered failed. **The exit value of whatever component of the node was run last determines the success or failure of the node.** Table 2.1 lists the definition of node success and failure for all variations of script and job success and failure, when `DAGMAN_ALWAYS_RUN_POST` is set to `False`. In this table, a dash (−) represents the case where a script does not exist for the DAG, **S** represents success, and **F** represents failure.

Table 2.2 lists the definition of node success and failure only for the cases where the PRE script fails, when `DAGMAN_ALWAYS_RUN_POST` is set to `True`.

PRE	JOB	POST	**Node**
-	S	-	**S**
-	F	-	**F**
-	S	S	**S**
-	S	F	**F**
-	F	S	**S**
-	F	F	**F**
S	S	-	**S**
S	F	-	**F**
S	S	S	**S**
S	S	F	**F**
S	F	S	**S**
S	F	F	**F**
F	not run	-	**F**
F	not run	not run	**F**

Table 2.1: Node success or failure definition with `DAGMAN_ALWAYS_RUN_POST = False (the default)`

PRE	JOB	POST	**Node**
F	not run	-	**F**
F	not run	S	**S**
F	not run	F	**F**

Table 2.2: Node **S**uccess or **F**ailure definition with `DAGMAN_ALWAYS_RUN_POST = True`

Special script argument macros

The five macros `$JOB`, `$RETRY`, `$MAX_RETRIES`, `$DAG_STATUS` and `$FAILED_COUNT` can be used within the DAG input file as arguments passed to a PRE or POST script. The three macros `$JOBID`, `$RETURN`, and `$PRE_SCRIPT_RETURN` can be used as arguments to POST scripts. The use of these variables is limited to being used as an individual command line *argument* to the script, surrounded by spaces, in order to cause the substitution of the variable's value.

The special macros are as follows:

- `$JOB` evaluates to the (case sensitive) string defined for *JobName*.
- `$RETRY` evaluates to an integer value set to 0 the first time a node is run, and is incremented each time the node is retried. See section 2.10.9 for the description of how to cause nodes to be retried.
- `$MAX_RETRIES` evaluates to an integer value set to the maximum number of retries for the node. See section 2.10.9 for the description of how to cause nodes to be retried. If no retries are set for the node, `$MAX_RETRIES` will be set to 0.
- `$JOBID` (for POST scripts only) evaluates to a representation of the HTCondor job ID of the node job. It is the value of the job ClassAd attribute `ClusterId`, followed by a period, and then followed by the value of the job ClassAd attribute `ProcId`. An example of a job ID might be 1234.0. For nodes with multiple jobs in the same cluster, the `ProcId` value is the one of the last job within the cluster.
- `$RETURN` (for POST scripts only) variable evaluates to the return value of the HTCondor job, if there is a single job within a cluster. With multiple jobs within the same cluster, there are two cases to consider. In the first case, all jobs within the cluster are successful; the value of `$RETURN` will be 0, indicating success. In the second case, one or more jobs from the cluster fail. When *condor_dagman* sees the first terminated event for a job that failed, it assigns that job's return value as the value of `$RETURN`, and it attempts to remove all remaining jobs within the cluster. Therefore, if multiple jobs in the cluster fail with different exit codes, a race condition determines which exit code gets assigned to `$RETURN`.

  A job that dies due to a signal is reported with a `$RETURN` value representing the additive inverse of the signal number. For example, SIGKILL (signal 9) is reported as -9. A job whose batch system submission fails is reported as -1001. A job that is externally removed from the batch system queue (by something other than *condor_dagman*) is reported as -1002.
- `$PRE_SCRIPT_RETURN` (for POST scripts only) variable evaluates to the return value of the PRE script of a node, if there is one. If there is no PRE script, this value will be -1. If the node job was skipped because of failure

of the PRE script, the value of `$RETURN` will be -1004 and the value of `$PRE_SCRIPT_RETURN` will be the exit value of the PRE script; the POST script can use this to see if the PRE script exited with an error condition, and assign success or failure to the node, as appropriate.

- `$DAG_STATUS` is the status of the DAG. Note that this macro's value and definition is unrelated to the attribute named `DagStatus` as defined for use in a node status file. This macro's value is the same as the job ClassAd attribute `DAG_Status` that is defined within the *condor_dagman* job's ClassAd. This macro may have the following values:
  - 0: OK
  - 1: error; an error condition different than those listed here
  - 2: one or more nodes in the DAG have failed
  - 3: the DAG has been aborted by an ABORT-DAG-ON specification
  - 4: removed; the DAG has been removed by *condor_rm*
  - 5: cycle; a cycle was found in the DAG
  - 6: halted; the DAG has been halted (see section 2.10.8)
- `$FAILED_COUNT` is defined by the number of nodes that have failed in the DAG.

Examples that use PRE or POST scripts

Examples use the diamond-shaped DAG. A first example uses a PRE script to expand a compressed file needed as input to each of the HTCondor jobs of nodes B and C. The DAG input file:

```
# 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 its command line arguments to form the file name of the compressed file. The script contains

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

Therefore, the PRE script invokes

```
gunzip B.gz
```

for node B, which uncompresses file `B.gz`, placing the result in file `B`.

A second example uses the `$RETURN` macro. The DAG input file contains the POST script specification:

```
SCRIPT POST A stage-out job_status $RETURN
```

If the HTCondor job of node A exits with the value -1, the POST script is invoked as

```
stage-out job_status -1
```

The slightly different example POST script specification in the DAG input file

```
SCRIPT POST A stage-out job_status=$RETURN
```

invokes the POST script with

```
stage-out job_status=$RETURN
```

This example shows that when there is no space between the = sign and the variable `$RETURN`, there is no substitution of the macro's value.

PRE_SKIP

The behavior of DAGMan with respect to node success or failure can be changed with the addition of a *PRE_SKIP* command. A *PRE_SKIP* line within the DAG input file uses the syntax:

PRE_SKIP *JobName*|**ALL_NODES** *non-zero-exit-code*

The PRE script of a node identified by *JobName* that exits with the value given by *non-zero-exit-code* skips the remainder of the node entirely. Neither the job associated with the node nor the POST script will be executed, and the node will be marked as successful.

2.10.3 Command Order

As of version 8.5.6, commands referencing a *JobName can* come before the JOB command defining that *JobName*.

For example, the command sequence

```
SCRIPT PRE NodeA foo.pl
VARS NodeA state="Wisconsin"
JOB NodeA bar.sub
```

is now legal (it would have been illegal in 8.5.5 and all previous versions).

2.10.4 Node Job Submit File Contents

Each node in a DAG may use a unique submit description file. A key limitation is that each HTCondor submit description file must submit jobs described by a single cluster number; DAGMan cannot deal with a submit description file producing multiple job clusters.

Consider again the diamond-shaped DAG example, where each node job uses 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 a sample HTCondor 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
queue
```

Since each node uses the same HTCondor submit description file, this implies that each node within the DAG runs the same job. The `$(Cluster)` macro produces unique file names for each job's output.

The job ClassAd attribute `DAGParentNodeNames` is also available for use within the submit description file. It defines a comma separated list of each *JobName* which is a parent node of this job's node. This attribute may be used in the **arguments** command for all but scheduler universe jobs. For example, if the job has two parents, with *JobName*s B and C, the submit description file command

```
arguments = $$([DAGParentNodeNames])
```

will pass the string `"B,C"` as the command line argument when invoking the job.

2.10.5 DAG Submission

A DAG is submitted using the tool *condor_submit_dag*. The manual page 938 details the command. The simplest of DAG submissions has the syntax

condor_submit_dag DAGInputFileName

and the current working directory contains the DAG input file.

The diamond-shaped DAG example may be submitted with

```
condor_submit_dag diamond.dag
```

Do not submit the same DAG, with same DAG input file, from within the same directory, such that more than one of this same DAG is running at the same time. It will fail in an unpredictable manner, as each instance of this same DAG will attempt to use the same file to enforce dependencies.

To increase robustness and guarantee recoverability, the *condor_dagman* process is run as an HTCondor job. As such, it needs a submit description file. *condor_submit_dag* generates this needed submit description file, naming it by appending `.condor.sub` to the name of the DAG input file. 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 create the submit description file, but not submit *condor_dagman* to HTCondor. To submit the DAG, once the submit description file is edited, use

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

Submit machines with limited resources are supported by command line options that place limits on the submission and handling of HTCondor jobs and PRE and POST scripts. Presented here are descriptions of the command line options to *condor_submit_dag*. These same limits can be set in configuration. Each limit is applied within a single DAG.

DAG Throttling

Total nodes/clusters: The **-maxjobs** option specifies the maximum number of clusters that *condor_dagman* can submit at one time. Since each node corresponds to a single cluster, this limit restricts the number of nodes that can be submitted (in the HTCondor queue) at a 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 node represents a single HTCondor proc that requires 4 MB of input files, and the proc will run in a directory with a volume of 100 MB of free space. Using the argument **-maxjobs 25** guarantees that a maximum of 25 clusters, using a maximum of 100 MB of space, will be submitted to HTCondor at one time. (See the *condor_submit_dag* man page (11) for more information. Also see the equivalent `DAGMAN_MAX_JOBS_SUBMITTED` configuration option (3.5.23).)

Idle procs: The number of idle procs within a given DAG can be limited with the optional command line argument **-maxidle**. *condor_dagman* will not submit any more node jobs until the number of idle procs in the DAG goes below this specified value, even if there are ready nodes in the DAG. This allows *condor_dagman* to submit jobs in a way that adapts to the load on the HTCondor pool at any given time. If the pool is lightly loaded, *condor_dagman* will end up submitting

more jobs; if the pool is heavily loaded, *condor_dagman* will submit fewer jobs. (See the *condor_submit_dag* man page (11) for more information. Also see the equivalent DAGMAN_MAX_JOBS_IDLE configuration option (3.5.23).)

Note that the **-maxjobs** option applies to counts of *clusters*, whereas the **-maxidle** option applies to counts of *procs*. Unfortunately, this can be a bit confusing. Of course, if none of your submit files create more than one proc, the distinction doesn't matter. For example, though, a node job submit file that queues 5 procs will count as one for **-maxjobs**, but five for **-maxidle** (if all of the procs are idle).

Subsets of nodes: Node submission can also be throttled in a finer-grained manner by grouping nodes into categories. See section 2.10.9 for more details.

PRE/POST scripts: Since PRE and POST scripts run on the submit machine, it may be desirable to limit the number of PRE or POST scripts running at one time. The optional **-maxpre** command line argument limits the number of PRE scripts that may be running at one time, and the optional **-maxpost** command line argument limits the number of POST scripts that may be running at one time. (See the *condor_submit_dag* man page (11) for more information. Also see the equivalent DAGMAN_MAX_PRE_SCRIPTS (3.5.23) and DAGMAN_MAX_POST_SCRIPTS (3.5.23) configuration options.)

2.10.6 File Paths in DAGs

condor_dagman assumes that all relative paths in a DAG input file and the associated HTCondor submit description files are relative to the current working directory when *condor_submit_dag* is run. This works well for submitting a single DAG. It presents problems when multiple independent DAGs are submitted with a single invocation of *condor_submit_dag*. Each of these independent DAGs would logically be in its own directory, such that it could be run or tested independent of other DAGs. Thus, all references to files will be designed to be relative to the DAG's own directory.

Consider an example DAG within a directory named `dag1`. There would be a DAG input file, named `one.dag` for this example. Assume the contents of this DAG input file specify a node job with

```
JOB A  A.submit
```

Further assume that partial contents of submit description file `A.submit` specify

```
executable = programA
input      = A.input
```

Directory contents are

```
dag1 (directory)
      one.dag
      A.submit
      programA
      A.input
```

All file paths are correct relative to the `dag1` directory. Submission of this example DAG sets the current working directory to `dag1` and invokes *condor_submit_dag*:

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

Expand this example such that there are now two independent DAGs, and each is contained within its own directory. For simplicity, assume that the DAG in `dag2` has remarkably similar files and file naming as the DAG in `dag1`. Assume that the directory contents are

```
parent (directory)
     dag1 (directory)
          one.dag
          A.submit
          programA
          A.input
     dag2 (directory)
          two.dag
          B.submit
          programB
          B.input
```

The goal is to use a single invocation of *condor_submit_dag* to run both dag1 and dag2. The invocation

```
cd parent
condor_submit_dag dag1/one.dag dag2/two.dag
```

does not work. Path names are now relative to `parent`, which is *not* the desired behavior.

The solution is the *-usedagdir* command line argument to *condor_submit_dag*. This feature runs each DAG as if *condor_submit_dag* had been run in the directory in which the relevant DAG file exists. A working invocation is

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

Output files will be placed in the correct directory, and the `.dagman.out` file will also be in the correct directory. A Rescue DAG file will be written to the current working directory, which is the directory when *condor_submit_dag* is invoked. The Rescue DAG should be run from that same current working directory. The Rescue DAG includes all the path information necessary to run each node job in the proper directory.

Use of *-usedagdir* does *not* work in conjunction with a JOB node specification within the DAG input file using the *DIR* keyword. Using both will be detected and generate an error.

2.10.7 DAG Monitoring and DAG Removal

After submission, the progress of the DAG can be monitored by looking at the job event log file(s), observing the e-mail that job submission to HTCondor causes, or by using *condor_q -dag*.

There is also a large amount of information logged in an extra file. The name of this extra file is produced by appending `.dagman.out` to the name of the DAG input file; for example, if the DAG input file is `diamond.dag`, this extra file is named `diamond.dag.dagman.out`. If this extra file grows too large, limit its size with the configuration variable `MAX_DAGMAN_LOG`, as defined in section 3.5.2. The `dagman.out` file is an important resource for debugging; save this file if a problem occurs. The `dagman.out` is appended to, rather than overwritten, with each new DAGMan run.

To remove an entire DAG, consisting of the *condor_dagman* job, plus any jobs submitted to HTCondor, remove the *condor_dagman* job by running *condor_rm*. 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   taylor         10/12 11:47   0+00:01:32 R  0   8.7  condor_dagman -f -
 11.0   taylor         10/12 11:48   0+00:00:00 I  0   3.6  B.out
 12.0   taylor         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
```

When a *condor_dagman* job is removed, all node jobs (including sub-DAGs) of that *condor_dagman* will be removed by the *condor_schedd*. As of version 8.5.8, the default is that *condor_dagman* itself also removes the node jobs (to fix a race condition that could result in "orphaned" node jobs). (The *condor_schedd* has to remove the node jobs to deal with the case of removing a *condor_dagman* job that has been held.)

The previous behavior of *condor_dagman* itself *not* removing the node jobs can be restored by setting the `DAGMAN_REMOVE_NODE_JOBS` configuration macro (see 3.5.23) to `False`. This will decrease the load on the *condor_schedd*, at the cost of allowing the possibility of "orphaned" node jobs.

A removed DAG will be considered failed unless the DAG has a FINAL node that succeeds.

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 the HTCondor queue.

2.10.8 Suspending a Running DAG

It may be desired to temporarily suspend a running DAG. For example, the load may be high on the submit machine, and therefore it is desired to prevent DAGMan from submitting any more jobs until the load goes down. There are two ways to suspend (and resume) a running DAG.

- Use *condor_hold*/*condor_release* on the *condor_dagman* job.

After placing the *condor_dagman* job on hold, no new node jobs will be submitted, and no PRE or POST scripts will be run. Any node jobs already in the HTCondor queue will continue undisturbed. Any running PRE or POST scripts will be killed. If the *condor_dagman* job is left on hold, it will remain in the HTCondor queue after all of the currently running node jobs are finished. To resume the DAG, use *condor_release* on the *condor_dagman* job.

Note that while the *condor_dagman* job is on hold, no updates will be made to the `dagman.out` file.

- Use a DAG halt file.

  The second way of suspending a DAG uses the existence of a specially-named file to change the state of the DAG. When in this halted state, no PRE scripts will be run, and no node jobs will be submitted. Running node jobs will continue undisturbed. A halted DAG will still run POST scripts, and it will still update the `dagman.out` file. This differs from behavior of a DAG that is held. Furthermore, a halted DAG will not remain in the queue indefinitely; when all of the running node jobs have finished, DAGMan will create a Rescue DAG and exit.

  To resume a halted DAG, remove the halt file.

  The specially-named file must be placed in the same directory as the DAG input file. The naming is the same as the DAG input file concatenated with the string `.halt`. For example, if the DAG input file is `test1.dag`, then `test1.dag.halt` will be the required name of the halt file.

  As any DAG is first submitted with *condor_submit_dag*, a check is made for a halt file. If one exists, it is removed.

Note that neither *condor_hold* nor a DAG halt is propagated to sub-DAGs. In other words, if you *condor_hold* or create a halt file for a DAG that has sub-DAGs, any sub-DAGs that are already in the queue will continue to submit node jobs.

A *condor_hold* or DAG halt *does*, however, apply to splices, because they are merged into the parent DAG and controlled by a single *condor_dagman* instance.

2.10.9 Advanced Features of DAGMan

Retrying Failed Nodes

DAGMan can retry any failed node in a DAG by specifying the node in the DAG input file with the *RETRY* command. The use of retry is optional. The syntax for retry is

RETRY *JobName*|**ALL_NODES** *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 keyword *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 macro `$RETRY` evaluates to an integer value, set to 0 first time a node is run, and is incremented each time for each time the node is retried. The macro `$MAX_RETRIES` is the value set for *NumberOfRetries*. These macros may be used as arguments passed to a PRE or POST script.

Stopping the Entire DAG

The *ABORT-DAG-ON* command 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*|**ALL_NODES** *AbortExitValue* [**RETURN** *DAGReturnValue*]

If the return value of the node specified by *JobName* matches *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 differs, as it would allow the DAG to continue running, until no more progress can be made due to dependencies.

The behavior differs based on the existence of PRE and/or POST scripts. If a PRE script returns the *AbortExitValue* value, the DAG is immediately aborted. If the HTCondor job within a node returns the *AbortExitValue* value, the DAG is aborted if the node has no POST script. If the POST script returns the *AbortExitValue* value, the DAG is aborted.

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* keyword 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.

A DAG return value other than 0, 1, or 2 will cause the *condor_dagman* job to stay in the queue after it exits and get retried, unless the `on_exit_remove` expression in the `.condor.sub` file is manually modified.

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, of which 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

Macros defined for DAG nodes can be used within the submit description file of the node job. The *VARS* command provides a method for defining a macro. Macros are defined on a per-node basis, using the syntax

VARS *JobName*|**ALL_NODES** *macroname="string"* [*macroname="string"...*]

The macro may be used within the submit description file of the relevant node. A *macroname* may contain alphanumeric characters (a-z, A-Z, and 0-9) and the underscore character. The space character delimits macros, such that there may be more than one macro defined on a single line. Multiple lines defining macros for the same node are permitted.

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

A restriction is that the *macroname* itself cannot begin with the string `queue`, in any combination of upper or lower case letters.

Examples

If the DAG input file contains

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

then the submit description file `A.submit` may use the macro `state`. Consider this submit description file `A.submit`:

```
# file name: A.submit
executable = A.exe
log        = A.log
arguments  = "$(state)"
queue
```

The macro value expands to become a command-line argument in the invocation of the job. The job is invoked with

```
A.exe Wisconsin
```

The use of macros may allow a reduction in the number of distinct submit description files. A separate example shows this intended use of *VARS*. In the case where the submit description file for each node varies only in file naming, macros reduce the number of submit description files to one.

This example references a single submit description file for each of the nodes in the DAG input file, and it uses the *VARS* entry to name files used by each job.

The relevant portion of the DAG input file appears as

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

VARS A filename="A"
VARS B filename="B"
VARS C filename="C"
```

The submit description file appears as

```
# submit description file called:  theonefile.sub
executable   = progX
output       = $(filename)
error        = error.$(filename)
log          = $(filename).log
queue
```

For a DAG such as this one, but with thousands of nodes, the ability to write and maintain a single submit description file together with a single, yet more complex, DAG input file is worthwhile.

Multiple macroname definitions

If a macro name for a specific node in a DAG is defined more than once, as it would be with the partial file contents

```
JOB job1 job1.submit
VARS job1 a="foo"
VARS job1 a="bar"
```

a warning is written to the log, of the format

```
Warning: VAR <macroname> is already defined in job <JobName>
Discovered at file "<DAG input file name>", line <line number>
```

The behavior of DAGMan is such that all definitions for the macro exist, but only the last one defined is used as the variable's value. Using this example, if the `job1.submit` submit description file contains

```
arguments = "$(a)"
```

then the argument will be `bar`.

Special characters within VARS string definitions

The value defined for a macro may contain spaces and tabs. It is also possible to have double quote marks and backslashes within a value. In order to have spaces or tabs within a value specified for a command line argument, use the New Syntax format for the **arguments** submit command, as described in section 11. Escapes for double quote marks depend on whether the New Syntax or Old Syntax format is used for the **arguments** submit command. Note that in both syntaxes, double quote marks require two levels of escaping: one level is for the parsing of the DAG input file, and the other level is for passing the resulting value through *condor_submit*.

As of HTCondor version 8.3.7, single quotes are permitted within the value specification. For the specification of command line **arguments**, single quotes can be used in three ways:

- in Old Syntax, within a macro's value specification
- in New Syntax, within a macro's value specification
- in New Syntax only, to delimit an argument containing white space

There are examples of all three cases below. In New Syntax, to pass a single quote as part of an argument, escape it with another single quote for *condor_submit* parsing as in the example's NodeA `fourth` macro.

As an example that shows uses of all special characters, here are only the relevant parts of a DAG input file. Note that the NodeA value for the macro `second` contains a tab.

```
VARS NodeA first="Alberto Contador"
VARS NodeA second="\"\"Andy Schleck\"\""
VARS NodeA third="Lance\\ Armstrong"
VARS NodeA fourth="Vincenzo ''The Shark'' Nibali"
VARS NodeA misc="!@#$%^&*()_-=+=[]{}?/"

VARS NodeB first="Lance_Armstrong"
VARS NodeB second="\\\"Andreas_Kloden\\\""
VARS NodeB third="Ivan\\_Basso"
VARS NodeB fourth="Bernard_'The_Badger'_Hinault"
VARS NodeB misc="!@#$%^&*()_-=+=[]{}?/"

VARS NodeC args="'Nairo Quintana' 'Chris Froome'"
```

Consider an example in which the submit description file for NodeA uses the New Syntax for the **arguments** command:

```
arguments = "'$(first)' '$(second)' '$(third)' '($fourth)' '$(misc)'"
```

The single quotes around each variable reference are only necessary if the variable value may contain spaces or tabs. The resulting values passed to the NodeA executable are:

```
Alberto Contador
"Andy Schleck"
Lance\ Armstrong
Vincenzo 'The Shark' Nibali
!@#$%^&*()_-=+=[]{}?/
```

Consider an example in which the submit description file for NodeB uses the Old Syntax for the **arguments** command:

```
arguments = $(first) $(second) $(third) $(fourth) $(misc)
```

The resulting values passed to the NodeB executable are:

```
Lance_Armstrong
"Andreas_Kloden"
Ivan\_Basso
Bernard_'The_Badger'_Hinault
!@#$%^&*()_-=+=[]{}?/
```

Consider an example in which the submit description file for NodeC uses the New Syntax for the **arguments** command:

```
arguments = "$(args)"
```

The resulting values passed to the NodeC executable are:

```
Nairo Quintana
Chris Froome
```

Using special macros within a definition

The `$(JOB)` and `$(RETRY)` macros may be used within a definition of the *string* that defines a variable. This usage requires parentheses, such that proper macro substitution may take place when the macro's value is only a portion of the string.

- `$(JOB)` expands to the node *JobName*. If the *VARS* line appears in a DAG file used as a splice file, then `$(JOB)` will be the fully scoped name of the node.

  For example, the DAG input file lines

  ```
  JOB  NodeC NodeC.submit
  VARS NodeC nodename="$(JOB)"
  ```

  set `nodename` to `NodeC`, and the DAG input file lines

```
JOB  NodeD NodeD.submit
VARS NodeD outfilename="$(JOB)-output"
```

set `outfilename` to `NodeD-output`.

- `$(RETRY)` expands to 0 the first time a node is run; the value is incremented each time the node is retried. For example:

```
VARS NodeE noderetry="$(RETRY)"
```

Using VARS to define ClassAd attributes

The *macroname* may also begin with a + character, in which case it names a ClassAd attribute. For example, the VARS specification

```
VARS NodeF +A="\"bob\""
```

results in the job ClassAd attribute

```
A = "bob"
```

Note that ClassAd string values must be quoted, hence there are escaped quotes in the example above. The outer quotes are consumed in the parsing of the DAG input file, so the escaped inner quotes remain in the definition of the attribute value.

Continuing this example, it allows the HTCondor submit description file for NodeF to use the following line:

```
arguments = "$$([A])"
```

The special macros may also be used. For example

```
VARS NodeG +B="$(RETRY)"
```

places the numerical attribute

```
B = 1
```

into the ClassAd when the NodeG job is run for a second time, which is the first retry and the value 1.

Setting Priorities for Nodes

The *PRIORITY* command assigns a priority to a DAG node (and to the HTCondor job(s) associated with the node). The syntax for *PRIORITY* is

PRIORITY *JobName*|**ALL_NODES** *PriorityValue*

The priority value is an integer (which can be negative). A larger numerical priority is better. The default priority is 0.

The node priority affects the order in which nodes that are ready (all of their parent nodes have finished successfully) at the same time will be submitted. The node priority also sets the node job's priority in the queue (that is, its `JobPrio` attribute), which affects the order in which jobs will be run once they are submitted (see 2.7.1 for more information about job priority). The node priority only affects the order of job submission *within a given DAG*; but once jobs are submitted, their `JobPrio` value affects the order in which they will be run relative to all jobs submitted by the same user.

Sub-DAGs can have priorities, just as "regular" nodes can. (The priority of a sub-DAG will affect the priorities of its nodes: see "effective node priorities" below.) Splices cannot be assigned a priority, but individual nodes within a splice *can* be assigned priorities.

Note that node priority does *not* override the DAG dependencies. Also note that node priorities are not *guarantees* of the relative order in which nodes will be run, even among nodes that become ready at the same time – so node priorities should not be used as a substitute for parent/child dependencies. In other words, priorities should be used when it is preferable, but not required, that some jobs run before others. (The order in which jobs are run once they are submitted can be affected by many things other than the job's priority; for example, whether there are machines available in the pool that match the job's requirements.)

PRE scripts can affect the order in which jobs run, so DAGs containing PRE scripts may not submit the nodes in exact priority order, even if doing so would satisfy the DAG dependencies.

Node priority is most relevant if node submission is throttled (via the *-maxjobs* or *-maxidle* command-line arguments or the `DAGMAN_MAX_JOBS_SUBMITTED` or `DAGMAN_MAX_JOBS_IDLE` configuration variables), or if there are not enough resources in the pool to immediately run all submitted node jobs. This is often the case for DAGs with large numbers of "sibling" nodes, or DAGs running on heavily-loaded pools.

Example

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 (and, mostly likely, run) before node B. Without this priority setting for node C, node B would be submitted first because the "JOB" statement for node B comes earlier in the DAG file than the "JOB" statement for node C.

Effective node priorities

The "effective" priority for a node (the priority controlling the order in which nodes are actually submitted, and which is assigned to `JobPrio`) is the sum of the explicit priority (specified in the DAG file) and the priority of the DAG itself. DAG priorities also default to 0, so they are most relevant for sub-DAGs (although a top-level DAG can be submitted with a non-zero priority by specifying a **-priority** value on the *condor_submit_dag* command line). **This algorithm for calculating effective priorities is a simplification introduced in version 8.5.7 (a node's effective priority is no longer dependent on the priorities of its parents).**

Here is an example to clarify:

```
    # File name: priorities.dag
    #
JOB A A.sub
SUBDAG EXTERNAL B SD.dag
PARENT A CHILD B
PRIORITY A 60
PRIORITY B 100

    # File name: SD.dag
    #
JOB SA SA.sub
JOB SB SB.sub
PARENT SA CHILD SB
PRIORITY SA 10
PRIORITY SB 20
```

In this example (assuming that priorities.dag is submitted with the default priority of 0), the effective priority of node A will be 60, and the effective priority of sub-DAG B will be 100. Therefore, the effective priority of node SA will be 110 and the effective priority of node SB will be 120.

The effective priorities listed above are assigned by DAGMan. There is no way to change the priority in the submit description file for a job, as DAGMan will override any **priority** command placed in a submit description file (unless the effective node priority is 0; in this case, any priority specified in the submit file will take effect).

Throttling Nodes by Category

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

CATEGORY *JobName*|**ALL_NODES** *CategoryName*

Category names cannot contain white space.

The *MAXJOBS* command 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 HTCondor 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.

Please see the end of section 2.10.9 on DAG Splicing for a description of the interaction between categories and splices.

Configuration Specific to a DAG

All configuration variables and their definitions that relate to DAGMan may be found in section 3.5.23.

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

1. In an HTCondor configuration file.
2. With an environment variable. Prepend the string `_CONDOR_` to the configuration variable's name.
3. With a line in the DAG input file using the keyword *CONFIG*, such that there is a configuration file specified that is specific to an instance of *condor_dagman*. The configuration file specification may instead be specified on the *condor_submit_dag* command line using the **-config** option.
4. For some configuration variables, *condor_submit_dag* command line argument specifies a configuration variable. For example, the configuration variable `DAGMAN_MAX_JOBS_SUBMITTED` has the corresponding command line argument *-maxjobs*.

For this ordered list, configuration values specified or parsed later in the list override ones specified earlier. For example, a value specified on the *condor_submit_dag* command line overrides corresponding values in any configuration file. And, a value specified in a DAGMan-specific configuration file overrides values specified in a general HTCondor configuration file.

The *CONFIG* command within the DAG input file specifies a configuration file to be used to set configuration variables related to *condor_dagman* when running this DAG. The syntax for *CONFIG* is

CONFIG *ConfigFileName*

As an example, if the DAG input file contains:

```
CONFIG dagman.config
```

then the configuration values in file `dagman.config` will be used for this DAG. If the contents of file `dagman.config` is

```
DAGMAN_MAX_JOBS_IDLE = 10
```

then this configuration is defined for this DAG.

Only a single configuration file can be specified for a given *condor_dagman* run. For example, if one file is specified within a DAG input file, 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 input files and 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 that are not for *condor_dagman* and not utilized by DaemonCore, yet are specified in a *condor_dagman*-specific configuration file are ignored.

Setting ClassAd attributes in the DAG file

The *SET_JOB_ATTR* keyword within the DAG input file specifies an attribute/value pair to be set in the DAGMan job's ClassAd. The syntax for *SET_JOB_ATTR* is

SET_JOB_ATTR *AttributeName=AttributeValue*

As an example, if the DAG input file contains:

```
SET_JOB_ATTR TestNumber = 17
```

the ClassAd of the DAGMan job itself will have an attribute `TestNumber` with the value `17`.

The attribute set by the *SET_JOB_ATTR* command is set only in the ClassAd of the DAGMan job itself – it is not propagated to node jobs of the DAG.

Values with spaces can be set by surrounding the string containing a space with single or double quotes. (Note that the quote marks themselves will be part of the value.)

Only a single attribute/value pair can be specified per *SET_JOB_ATTR* command. If the same attribute is specified multiple times in the DAG (or in multiple DAGs run by the same DAGMan instance) the last-specified value is the one that will be utilized. An attribute set in the DAG file can be overridden by specifying

```
-append '+<attribute> = <value>'
```

on the *condor_submit_dag* command line.

Optimization of Submission Time

condor_dagman works by watching log files for events, such as submission, termination, and going on hold. When a new job is ready to be run, it is submitted to the *condor_schedd*, which needs to acquire a computing resource.

Acquisition requires the *condor_schedd* to contact the central manager and get a claim on a machine, and this claim cycle can take many minutes.

Configuration variable DAGMAN_HOLD_CLAIM_TIME avoids the wait for a negotiation cycle. When set to a non zero value, the *condor_schedd* keeps a claim idle, such that the *condor_startd* delays in shifting from the Claimed to the Preempting state (see Figure 3.1). Thus, if another job appears that is suitable for the claimed resource, then the *condor_schedd* will submit the job directly to the *condor_startd*, avoiding the wait and overhead of a negotiation cycle. This results in a speed up of job completion, especially for linear DAGs in pools that have lengthy negotiation cycle times.

By default, DAGMAN_HOLD_CLAIM_TIME is 20, causing a claim to remain idle for 20 seconds, during which time a new job can be submitted directly to the already-claimed *condor_startd*. A value of 0 means that claims are not held idle for a running DAG. If a DAG node has no children, the value of DAGMAN_HOLD_CLAIM_TIME will be ignored; the KeepClaimIdle attribute will not be defined in the job ClassAd of the node job, unless the job requests it using the submit command **keep_claim_idle**.

Single Submission of Multiple, Independent DAGs

A single use of *condor_submit_dag* may execute multiple, independent DAGs. Each independent DAG has its own, distinct DAG input file. These DAG input files are command-line arguments to *condor_submit_dag*.

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 original independent DAGs with a single DAG. The file name of this Rescue DAG is based on the DAG input file listed first within the command-line arguments. For example, assume that three independent DAGs are submitted with

```
condor_submit_dag A.dag B.dag C.dag
```

The first listed is A.dag. The remainder of the specialized file name adds a suffix onto this first DAG input file name, A.dag. The suffix is _multi.rescue<XXX>, where <XXX> is substituted by the 3-digit number of the Rescue DAG created as defined in section 2.10.10. The first time a Rescue DAG is created for the example, it will have the file name A.dag_multi.rescue001.

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 3.5.23).

INCLUDE

The *INCLUDE* command allows the contents of one DAG file to be parsed as if they were physically included in the referencing DAG file. The syntax for *INCLUDE* is

INCLUDE *FileName*

For example, if we have two DAG files like this:

```
# File name: foo.dag
#
    JOB  A  A.sub
    INCLUDE bar.dag

# File name: bar.dag
#
    JOB  B  B.sub
    JOB  C  C.sub
```

this is equivalent to the single DAG file:

```
    JOB  A  A.sub
    JOB  B  B.sub
    JOB  C  C.sub
```

Note that the included file must be in proper DAG syntax. Also, there are many cases where a valid included DAG file will cause a parse error, such as the including and included files defining nodes with the same name.

*INCLUDE*s can be nested to any depth (be sure not to create a cycle of includes!).

Example: Using INCLUDE to simplify multiple similar workflows

One use of the *INCLUDE* command is to simplify the DAG files when we have a single workflow that we want to run on a number of data sets. In that case, we can do something like this:

```
# File name: workflow.dag
# Defines the structure of the workflow
    JOB Split split.sub
    JOB Process00 process.sub
    ...
    JOB Process99 process.sub
    JOB Combine combine.sub
    PARENT Split CHILD Process00 ... Process99
    PARENT Process00 ... Process99 CHILD Combine
```

```
# File name: split.sub
    executable = my_split
    input = $(dataset).phase1
    output = $(dataset).phase2
    ...

# File name: data57.vars
    VARS Split dataset="data57"
    VARS Process00 dataset="data57"
    ...
    VARS Process99 dataset="data57"
    VARS Combine dataset="data57"

# File name: run_dataset57.dag
    INCLUDE workflow.dag
    INCLUDE data57.vars
```

Then, to run our workflow on dataset 57, we run the following command:

```
condor_submit_dag run_dataset57.dag
```

This avoids having to duplicate the *JOB* and *PARENT/CHILD* commands for every dataset – we can just re-use the `workflow.dag` file, in combination with a dataset-specific vars file.

Composing workflows from multiple DAG files

The organization and dependencies of the jobs within a DAG are the keys to its utility. Some workflows are naturally constructed hierarchically, such that a node within a DAG is also a DAG (instead of a "simple" HTCondor job). HTCondor DAGMan handles this situation easily, and allows DAGs to be nested to any depth.

There are two ways that DAGs can be nested within other DAGs: sub-DAGs (see 2.10.9) and splices (see 2.10.9).

With sub-DAGs, each DAG has its own *condor_dagman* job, which then becomes a node job within the higher-level DAG. With splices, on the other hand, the nodes of the spliced DAG are directly incorporated into the higher-level DAG. Therefore, splices do not result in additional *condor_dagman* instances.

A weakness in scalability exists when submitting external sub-DAGs, because each executing independent DAG requires its own instance of *condor_dagman* to be running. The outer DAG has an instance of *condor_dagman*, and each named SUBDAG has an instance of *condor_dagman* while it is in the HTCondor queue. The scaling issue presents itself when a workflow contains hundreds or thousands of sub-DAGs that are queued at the same time. (In this case, the resources (especially memory) consumed by the multiple *condor_dagman* instances can be a problem.) Further, there may be many Rescue DAGs created if a problem occurs. (Note that the scaling issue depends only on how many sub-DAGs are queued at any given time, not the total number of sub-DAGs in a given workflow; division of a large workflow into *sequential* sub-DAGs can actually enhance scalability.) To alleviate these concerns, the DAGMan language introduces the concept of graph splicing.

Because splices are simpler in some ways than sub-DAGs, they are generally preferred unless certain features are needed that are only available with sub-DAGs. This document: https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=SubDagsVsSplices explains the pros and cons of splices and external sub-DAGs, and should help users decide which alternative is better for their application.

Note that sub-DAGs and splices can be combined in a single workflow, and can be nested to any depth (but be sure to avoid recursion, which will cause problems!).

A DAG Within a DAG Is a SUBDAG

As stated above, the SUBDAG EXTERNAL command causes the specified DAG file to be run by a separate instance of *condor_dagman*, with the *condor_dagman* job becoming a node job within the higher-level DAG.

The syntax for the SUBDAG command is

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

The optional specifications of **DIR**, **NOOP**, and **DONE**, if used, must appear in this order within the entry. **NOOP** and **DONE** for **SUBDAG** nodes have the same effect that they do for **JOB** nodes.

A **SUBDAG** node is essentially the same as any other node, except that the DAG input file for the inner DAG is specified, instead of the HTCondor submit file. The keyword **EXTERNAL** means that the SUBDAG is run within its own instance of *condor_dagman*.

Since more than one DAG is being discussed, here is terminology introduced to clarify which DAG is which. Reuse the example diamond-shaped DAG as given in Figure 2.2. Assume that node B of this diamond-shaped DAG will itself be a DAG. The DAG of node B is called a SUBDAG, inner DAG, or lower-level DAG. The diamond-shaped DAG is called the outer or top-level 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 HTCondor submit description file, used by *condor_dagman*, corresponding to `inner.dag` will be named `inner.dag.condor.sub`. The DAGMan submit description file is always named `<DAG file name>.condor.sub`. Each DAG or SUBDAG results in the submission of *condor_dagman* as an HTCondor job, and *condor_submit_dag* creates this submit description file.

The preferred specification of the 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
```

Within the outer DAG's input file, the **SUBDAG** command specifies a special case of a **JOB** node, where the job is itself a DAG.

One of the benefits of using the SUBDAG feature is that portions of the overall workflow can be constructed and modified during the execution of the DAG (a SUBDAG file doesn't have to exist until just before it is submitted). A drawback can be that each SUBDAG causes its own distinct job submission of *condor_dagman*, leading to a larger number of jobs, together with their potential need of carefully constructed policy configuration to throttle node submission or execution (because each SUBDAG has its own throttles).

Here are details that affect SUBDAGs:

- Nested DAG Submit Description File Generation

  There are three ways to generate the `<DAG file name>.condor.sub` file of a SUBDAG:

  - **Lazily** (the default in HTCondor version 7.5.2 and later versions)
  - **Eagerly** (the default in HTCondor versions 7.4.1 through 7.5.1)
  - **Manually** (the only way prior to version HTCondor version 7.4.1)

  When the `<DAG file name>.condor.sub` file is generated **lazily**, this file is generated immediately before the SUBDAG job is submitted. Generation is accomplished by running

  ```
  condor_submit_dag -no_submit
  ```

  on the DAG input file specified in the **SUBDAG** entry. This is the default behavior. There are advantages to this lazy mode of submit description file creation for the SUBDAG:

  - The DAG input file for a SUBDAG does not have to exist until the SUBDAG is ready to run, so this file can be dynamically created by earlier parts of the outer DAG or by the PRE script of the node containing the SUBDAG.
  - It is now possible to have SUBDAGs within splices. That is not possible with eager submit description file creation, because *condor_submit_dag* does not understand splices.

  The main disadvantage of lazy submit file generation is that a syntax error in the DAG input file of a SUBDAG will not be discovered until the outer DAG tries to run the inner DAG.

  When `<DAG file name>.condor.sub` files are generated **eagerly**, *condor_submit_dag* runs itself recursively (with the *-no_submit* option) on each SUBDAG, so all of the `<DAG file name>.condor.sub` files are generated before the top-level DAG is actually submitted. To generate the `<DAG file name>.condor.sub` files eagerly, pass the *-do_recurse* flag to *condor_submit_dag*; also set the

DAGMAN_GENERATE_SUBDAG_SUBMITS configuration variable to False, so that *condor_dagman* does not re-run *condor_submit_dag* at run time thereby regenerating the submit description files.

To generate the .condor.sub files **manually**, run

```
condor_submit_dag -no_submit
```

on each lower-level DAG file, before running *condor_submit_dag* on the top-level DAG file; also set the DAGMAN_GENERATE_SUBDAG_SUBMITS configuration variable to False, so that *condor_dagman* does not re-run *condor_submit_dag* at run time. The main reason for generating the <DAG file name>.condor.sub files manually is to set options for the lower-level DAG that one would not otherwise be able to set An example of this is the *-insert_sub_file* option. For instance, using the given example do the following to manually generate HTCondor submit description files:

```
condor_submit_dag -no_submit -insert_sub_file fragment.sub inner.dag
condor_submit_dag diamond.dag
```

Note that most *condor_submit_dag* command-line flags have corresponding configuration variables, so we encourage the use of per-DAG configuration files, especially in the case of nested DAGs. This is the easiest way to set different options for different DAGs in an overall workflow.

It is possible to combine more than one method of generating the <DAG file name>.condor.sub files. For example, one might pass the *-do_recurse* flag to *condor_submit_dag*, but leave the DAGMAN_GENERATE_SUBDAG_SUBMITS configuration variable set to the default of True. Doing this would provide the benefit of an immediate error message at submit time, if there is a syntax error in one of the inner DAG input files, but the lower-level <DAG file name>.condor.sub files would still be regenerated before each nested DAG is submitted.

The values of the following command-line flags are passed from the top-level *condor_submit_dag* instance to any lower-level *condor_submit_dag* instances. This occurs whether the lower-level submit description files are generated lazily or eagerly:

- **-verbose**
- **-force**
- **-notification**
- **-allowlogerror**
- **-dagman**
- **-usedagdir**
- **-outfile_dir**
- **-oldrescue**
- **-autorescue**
- **-dorescuefrom**
- **-allowversionmismatch**
- **-no_recurse/do_recurse**
- **-update_submit**

- **-import_env**
- **-suppress_notification**
- **-priority**
- **-dont_use_default_node_log**

The values of the following command-line flags are preserved in any already-existing lower-level DAG submit description files:

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

Other command-line arguments are set to their defaults in any lower-level invocations of *condor_submit_dag*.

The **-force** option will cause existing DAG submit description files to be overwritten without preserving any existing values.

- Submission of the outer DAG

  The outer DAG is submitted as before, with the command

  ```
  condor_submit_dag diamond.dag
  ```

- Interaction with Rescue DAGs

  The use of new-style Rescue DAGs is now the default. With new-style rescue DAGs, the appropriate rescue DAG(s) will be run automatically if there is a failure somewhere in the workflow. For example (given the DAGs in the example at the beginning of the SUBDAG section), if one of the nodes in `inner.dag` fails, this will produce a Rescue DAG for `inner.dag` (named `inner.dag.rescue.001`). Then, since `inner.dag` failed, node B of `diamond.dag` will fail, producing a Rescue DAG for `diamond.dag` (named `diamond.dag.rescue.001`, etc.). If the command

  ```
  condor_submit_dag diamond.dag
  ```

  is re-run, the most recent outer Rescue DAG will be run, and this will re-run the inner DAG, which will in turn run the most recent inner Rescue DAG.

- File Paths

  Remember that, unless the DIR keyword is used in the outer DAG, the inner DAG utilizes the current working directory when the outer DAG is submitted. Therefore, all paths utilized by the inner DAG file must be specified accordingly.

DAG Splicing

As stated above, the SPLICE command causes the nodes of the spliced DAG to be directly incorporated into the higher-level DAG (the DAG containing the SPLICE command).

The syntax for the *SPLICE* command is

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

A splice is a named instance of a subgraph which is specified in a separate DAG file. The splice is treated as an entity for dependency specification in the including DAG. (Conceptually, a splice is treated as a node within the DAG containing the SPLICE command, although there are some limitations, which are discussed below. This means, for example, that splices can have parents and children.) A splice can also be incorporated into an including DAG without any dependencies; it is then considered a disjoint DAG within the including DAG.

The same DAG file can be reused as differently named splices, each one incorporating a copy of the dependency graph (and nodes therein) into 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 `'+'`. (In other words, if a splice named "SpliceX" contains a node named "NodeY", the full node name once the DAGs are parsed is "SpliceX+NodeY". This character is chosen due to a restriction in the allowable characters which may be in a file name across the variety of platforms that HTCondor supports. In any DAG input file, all splices must have unique names, but the same splice name may be reused in different DAG input files.

HTCondor 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* command 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 following series of examples illustrate potential uses of splicing. To simplify the examples, presume that each and every job uses the same, simple HTCondor 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 2.3 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 2.4 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
```

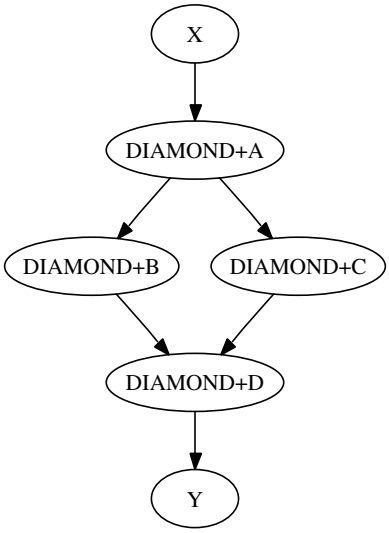

Figure 2.3: The diamond-shaped DAG spliced between two nodes.

```
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)"

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
```

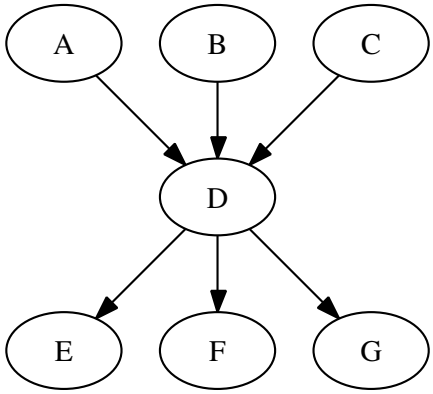

Figure 2.4: The X-shaped DAG.

File `s1.dag` continues the example, presenting the DAG input file that incorporates two separate splices of the X-shaped DAG. Figure 2.5 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 2.6 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.

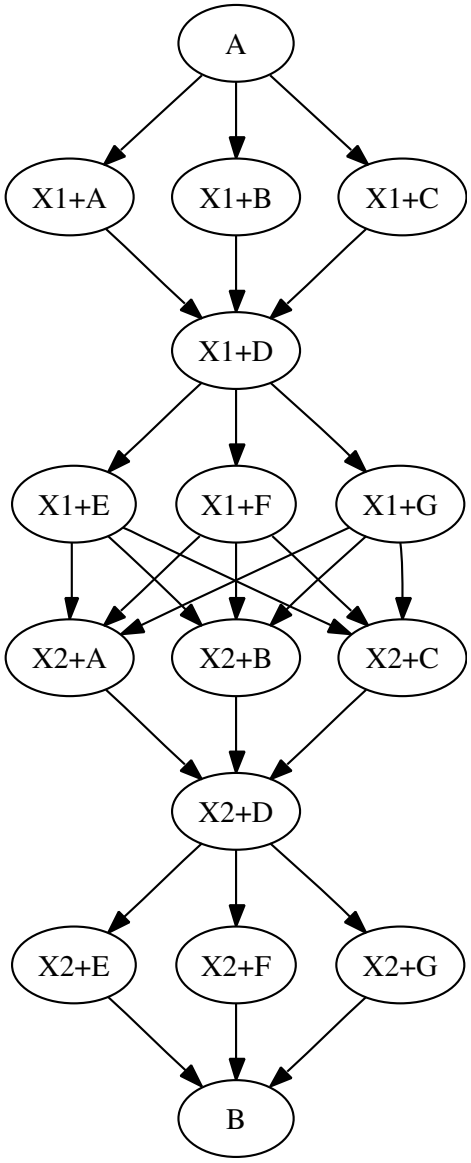

Figure 2.5: The DAG described by `s1.dag`.

```
# BEGIN DAG FILE toplevel.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
```

Splices and rescue DAGs

Because the nodes of a splice are directly incorporated into the DAG containing the SPLICE command, splices do not generate their own rescue DAGs, unlike SUBDAG EXTERNALs.

The DIR option with splices

The *DIR* option specifies a working directory for a splice, from which the splice will be parsed and the jobs within the splice submitted. The directory associated with the splice's *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 node's, 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 "full" rescue DAG generated by a DAG run with the *-usedagdir* argument will contain DIR specifications, so such a rescue DAG must be run *without* the *-usedagdir* argument. (Note that "full" rescue DAGs are no longer the default.)

Limitation: splice DAGs must exist at submit time

Unlike the DAG files referenced in a SUBDAG EXTERNAL command, DAG files referenced in a SPLICE command must exist when the DAG containing the SPLICE command is submitted. (Note that, if a SPLICE is contained within

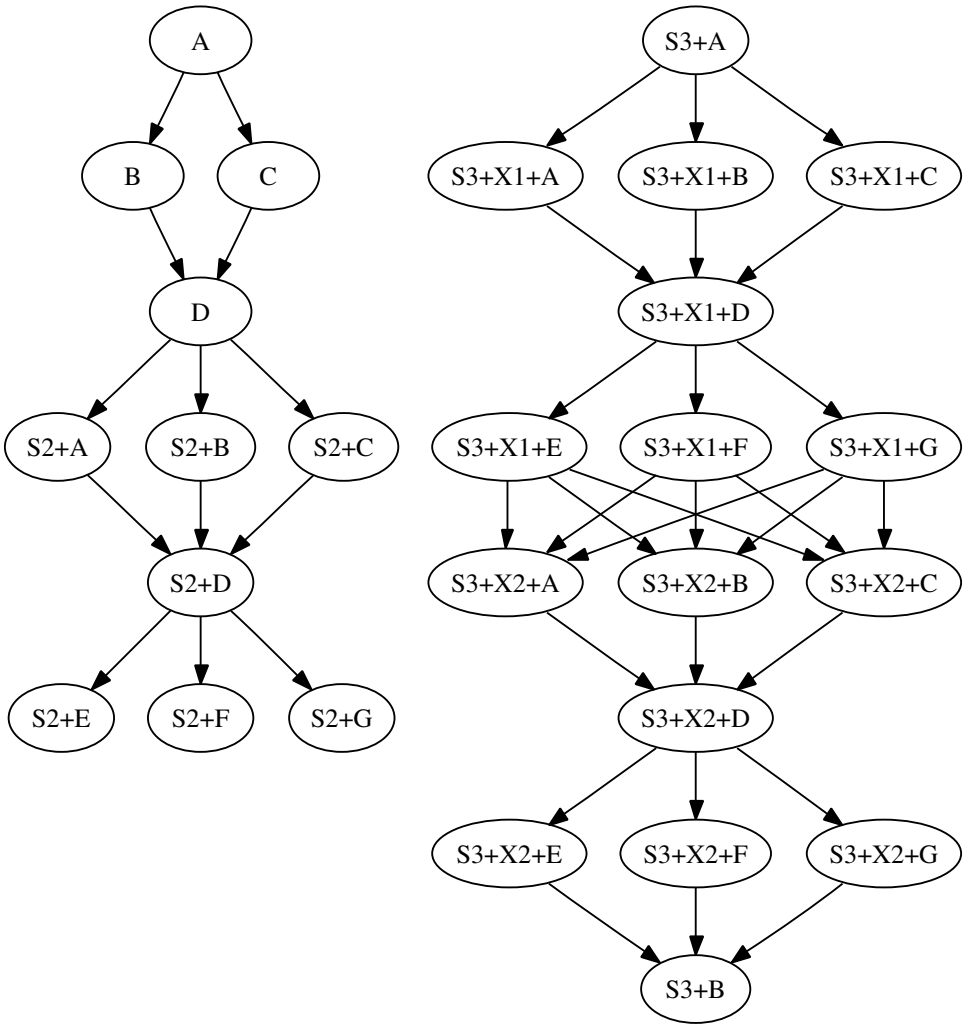

Figure 2.6: The complex splice example DAG.

a sub-DAG, the splice DAG must exist at the time that the sub-DAG is submitted, not when the top-most DAG is submitted, so the splice DAG can be created by a part of the workflow that runs before the relevant sub-DAG.)

Limitation: Splices and PRE or POST Scripts

A PRE or POST script may not be specified for a splice (however, nodes within a spliced DAG can have PRE and POST scripts). (The reason for this is that, when the DAG is parsed, the splices are also parsed and the splice nodes are directly incorporated into the DAG containing the SPLICE command. Therefore, once parsing is complete, there are no actual nodes corresponding to the splice itself to which to "attach" the PRE or POST scripts.)

To achieve the desired effect of having a PRE script associated with a splice, introduce a new NOOP node into the DAG with the splice as a dependency. Attach the PRE script to the NOOP node.

```
# BEGIN DAG FILE example1.dag

# Names a node with no associated node job, a NOOP node
# Note that the file noop.submit does not need to exist
JOB OnlyPreNode noop.submit NOOP

# Attach a PRE script to the NOOP node
SCRIPT PRE OnlyPreNode prescript.sh

# Define the splice
SPLICE TheSplice thenode.dag

# Define the dependency
PARENT OnlyPreNode CHILD TheSplice

# END DAG FILE example1.dag
```

The same technique is used to achieve the effect of having a POST script associated with a splice. Introduce a new NOOP node into the DAG as a child of the splice, and attach the POST script to the NOOP node.

```
# BEGIN DAG FILE example2.dag

# Names a node with no associated node job, a NOOP node
# Note that the file noop.submit does not need to exist.
JOB OnlyPostNode noop.submit NOOP

# Attach a POST script to the NOOP node
SCRIPT POST OnlyPostNode postscript.sh

# Define the splice
SPLICE TheSplice thenode.dag

# Define the dependency
PARENT TheSplice CHILD OnlyPostNode

# END DAG FILE example2.dag
```

Limitation: Splices and the RETRY of a Node, use of VARS, or use of PRIORITY

A RETRY, VARS or PRIORITY command cannot be specified for a SPLICE; however, individual nodes within a spliced DAG can have a RETRY, VARS or PRIORITY specified.

Here is an example showing a DAG that will not be parsed successfully:

```
# top level DAG input file
JOB    A a.sub
SPLICE B b.dag
PARENT A  CHILD B
```

```
# cannot work, as B is not a node in the DAG once
# splice B is incorporated
RETRY B 3
VARS B dataset="10"
PRIORITY B 20
```

The following example *will* work:

```
# top level DAG input file
JOB    A a.sub
SPLICE B b.dag
PARENT A  CHILD B

# file: b.dag
JOB    X x.sub
RETRY X 3
VARS X dataset="10"
PRIORITY X 20
```

When RETRY is desired on an entire subgraph of a workflow, sub-DAGs (see above) must be used instead of splices.

Here is the same example, now defining job B as a SUBDAG, and effecting RETRY on that SUBDAG.

```
# top level DAG input file
JOB    A a.sub
SUBDAG EXTERNAL B b.dag
PARENT A  CHILD B

RETRY B 3
```

Limitation: The Interaction of Categories and MAXJOBS with Splices

Categories normally refer only to nodes within a given splice. All of the assignments of nodes to a category, and the setting of the category throttle, should be done within a single DAG file. However, it is now possible to have categories include nodes from within more than one splice. To do this, the category name is prefixed with the '+' (plus) character. This tells DAGMan that the category is a cross-splice category. Towards deeper understanding, what this really does is prevent renaming of the category when the splice is incorporated into the upper-level DAG. The MAXJOBS specification for the category can appear in either the upper-level DAG file or one of the splice DAG files. It probably makes the most sense to put it in the upper-level DAG file.

Here is an example which applies a single limitation on submitted jobs, identifying the category with `+init`.

```
# relevant portion of file name: upper.dag
```

```
    SPLICE A splice1.dag
    SPLICE B splice2.dag

    MAXJOBS +init 2

# relevant portion of file name: splice1.dag

    JOB C C.sub
    CATEGORY C +init
    JOB D D.sub
    CATEGORY D +init


# relevant portion of file name: splice2.dag

    JOB X X.sub
    CATEGORY X +init
    JOB Y Y.sub
    CATEGORY Y +init
```

For both global and non-global category throttles, settings at a higher level in the DAG override settings at a lower level. In this example:

```
# relevant portion of file name: upper.dag

    SPLICE A lower.dag

    MAXJOBS A+catX 10
    MAXJOBS +catY 2


# relevant portion of file name: lower.dag

    MAXJOBS catX 5
    MAXJOBS +catY 1
```

the resulting throttle settings are 2 for the `+catY` category and 10 for the `A+catX` category in splice. Note that non-global category names are prefixed with their splice name(s), so to refer to a non-global category at a higher level, the splice name must be included.

DAG Splice Connections

In the "default" usage of splices described above, when one splice is the parent of another splice, all "terminal" nodes (nodes with no children) of the parent splice become parents of all "initial" nodes (nodes with no parents) of the child splice. The CONNECT, PIN_IN, and PIN_OUT commands (added in version 8.5.7) allow more flexible dependencies between splices. (The terms PIN_IN and PIN_OUT were chosen because of the hardware analogy.)

The syntax for *CONNECT* is

CONNECT *OutputSpliceName InputSpliceName*

The syntax for *PIN_IN* is

PIN_IN *NodeName PinNumber*

The syntax for *PIN_OUT* is

PIN_OUT *NodeName PinNumber*

All output splice nodes connected to a given pin_out will become parents of all input splice nodes connected to the corresponding pin_in. (The pin_ins and pin_outs exist only to create the correct parent/child dependencies between nodes. Once the DAG is parsed, there are no actual DAG objects corresponding to the pin_ins and pin_outs.)

Any given splice can contain both PIN_IN and PIN_OUT definitions, and can be both an input and output splice in different CONNECT commands. Furthermore, a splice can appear in any number of CONNECT commands (for example, a given splice could be the output splice in two CONNECT commands that have different input splices). It is *not* an error for a splice to have PIN_IN or PIN_OUT definitions that are not associated with a CONNECT command – such PIN_IN and PIN_OUT commands are simply ignored.

Note that the pin_ins and pin_outs must be defined *within* the relevant splices (this can be done with *INCLUDE* commands), not in the DAG that connects the splices.

There are a number of restrictions on splice connections:

- Connections can be made only between two splices; "regular" nodes or sub-DAGs cannot be used in a CONNECT command.
- Pin_ins and pin_outs must be numbered consecutively starting at 1.
- The pin_outs of the output splice in a connect command must match the pin_ins of the input splice in the command.
- All "initial" nodes (nodes with no parents) of an input splice used in a CONNECT command must be connected to a pin_in.

Violating any of these restrictions will result in an error during the parsing of the DAG files.

Note: it is probably desireable for any "terminal" node (a node with no children) in the output splice to be connected to a pin_out – but this is not required.

Here is a simple example:

```
# File: top.dag
    SPLICE A spliceA.dag
    SPLICE B spliceB.dag
    SPLICE C spliceC.dag

    CONNECT A B
    CONNECT B C

# File: spliceA.dag
    JOB A1 A1.sub
    JOB A2 A2.sub

    PIN_OUT A1 1
    PIN_OUT A2 2

# File: spliceB.dag
    JOB B1 B1.sub
    JOB B2 B2.sub
    JOB B3 B3.sub
    JOB B4 B4.sub

    PIN_IN B1 1
    PIN_IN B2 1
    PIN_IN B3 2
    PIN_IN B4 2

    PIN_OUT B1 1
    PIN_OUT B2 2
    PIN_OUT B3 3
    PIN_OUT B4 4

# File: spliceC.dag
    JOB C1 C1.sub

    PIN_IN C1 1
    PIN_IN C1 2
    PIN_IN C1 3
    PIN_IN C1 4
```

In this example, node A1 will be the parent of B1 and B2; node A2 will be the parent of B3 and B4; and nodes B1, B2, B3 and B4 will all be parents of C1.

A diagram of the above example:

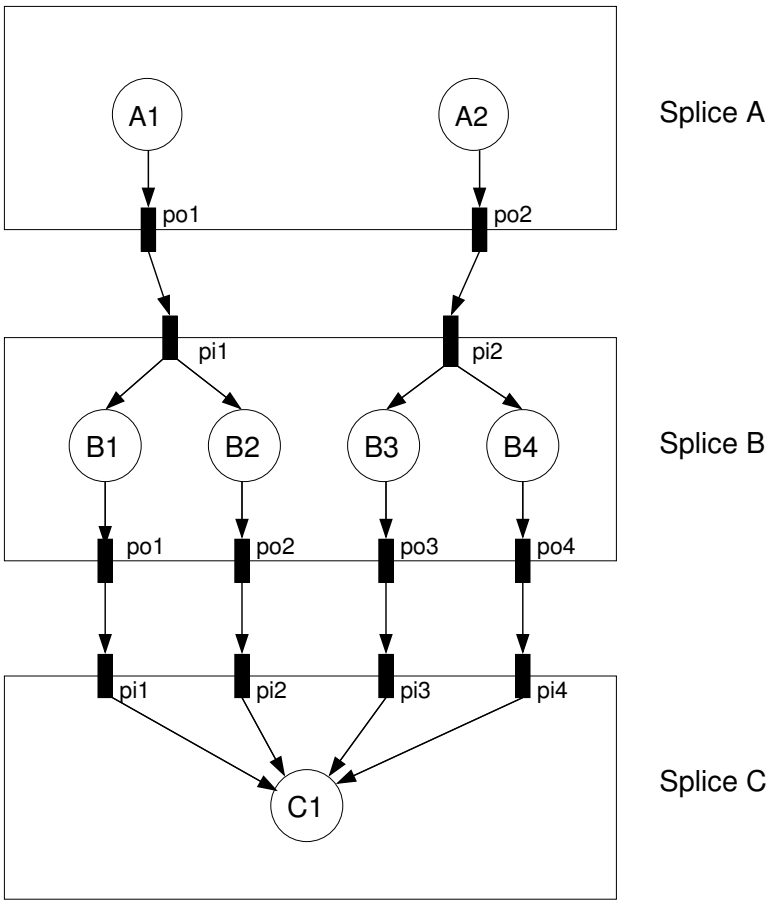

Figure 2.7: Diagram of the splice connect example

FINAL node

A FINAL node is a single and special node that is always run at the end of the DAG, even if previous nodes in the DAG have failed. A FINAL node can be used for tasks such as cleaning up intermediate files and checking the output of previous nodes. The *FINAL* command in the DAG input file specifies a node job to be run at the end of the DAG.

The syntax used for the *FINAL* command is

FINAL *JobName SubmitDescriptionFileName* [**DIR** *directory*] [**NOOP**]

The FINAL node within the DAG is identified by *JobName*, and the HTCondor job is described by the contents of the HTCondor submit description file given by *SubmitDescriptionFileName*.

The keywords *DIR* and *NOOP* are as detailed in section 2.10.2. If both *DIR* and *NOOP* are used, they must appear in the order shown within the syntax specification.

There may only be one FINAL node in a DAG. A parse error will be logged by the *condor_dagman* job in the `dagman.out` file, if more than one FINAL node is specified.

The FINAL node is virtually always run. It is run if the *condor_dagman* job is removed with *condor_rm*. The only case in which a FINAL node is not run is if the configuration variable `DAGMAN_STARTUP_CYCLE_DETECT` is set to `True`, and a cycle is detected at start up time. If `DAGMAN_STARTUP_CYCLE_DETECT` is set to `False` and a cycle is detected during the course of the run, the FINAL node *will* be run.

The success or failure of the FINAL node determines the success or failure of the entire DAG, overriding the status of all previous nodes. This includes any status specified by any ABORT-DAG-ON specification that has taken effect. If some nodes of a DAG fail, but the FINAL node succeeds, the DAG will be considered successful. Therefore, it is important to be careful about setting the exit status of the FINAL node.

The `$DAG_STATUS` and `$FAILED_COUNT` macros can be used both as PRE and POST script arguments, and in node job submit description files. As an example of this, here are the partial contents of the DAG input file,

```
FINAL final_node final_node.sub
SCRIPT PRE final_node final_pre.pl $DAG_STATUS $FAILED_COUNT
```

and here are the partial contents of the submit description file, `final_node.sub`

```
arguments = "$(DAG_STATUS) $(FAILED_COUNT)"
```

If there is a FINAL node specified for a DAG, it will be run at the end of the workflow. If this FINAL node must not do anything in certain cases, use the `$DAG_STATUS` and `$FAILED_COUNT` macros to take appropriate actions. Here is an example of that behavior. It uses a PRE script that aborts if the DAG has been removed with *condor_rm*, which, in turn, causes the FINAL node to be considered failed without actually submitting the HTCondor job specified for the node. Partial contents of the DAG input file:

```
FINAL final_node final_node.sub
SCRIPT PRE final_node final_pre.pl $DAG_STATUS
```

and partial contents of the Perl PRE script, `final_pre.pl`:

```
#! /usr/bin/env perl

if ($ARGV[0] eq 4) {
    exit(1);
}
```

There are restrictions on the use of a FINAL node. The DONE option is *not* allowed for a FINAL node. And, a FINAL node may not be referenced in any of the following specifications:

- PARENT, CHILD

- RETRY
- ABORT-DAG-ON
- PRIORITY
- CATEGORY

As of HTCondor version 8.3.7, DAGMan allows at most two submit attempts of a FINAL node, if the DAG has been removed from the queue with *condor_rm*.

The ALL_NODES option

In the following commands, a specific node name can be replaced by the option *ALL_NODES*:

- **SCRIPT**
- **PRE_SKIP**
- **RETRY**
- **ABORT-DAG-ON**
- **VARS**
- **PRIORITY**
- **CATEGORY**

This will cause the given command to apply to all nodes (except any FINAL node) in that DAG.

The ALL_NODES *never* applies to a FINAL node. (If the *ALL_NODES* option is used in a DAG that has a FINAL node, the `dagman.out` file will contain messages noting that the FINAL node is skipped when parsing the relevant commands.)

The *ALL_NODES* option is case-insensitive.

It is important to note that the *ALL_NODES* option does *not* apply across splices and sub-DAGs. In other words, an *ALL_NODES* option within a splice or sub-DAG will apply only to nodes within that splice or sub-DAG; also, an *ALL_NODES* option in a parent DAG will not apply to any splices or sub-DAGs referenced by the parent DAG.

The *ALL_NODES* option *does* work in combination with the *INCLUDE* command. In other words, a command within an included file that uses the *ALL_NODES* option will apply to all nodes in the including DAG (again, except any FINAL node).

As of version 8.5.8, the *ALL_NODES* option cannot be used when multiple DAG files are specified on the *condor_submit_dag* command line. Hopefully this limitation will be fixed in a future release.

When multiple commands (whether using the *ALL_NODES* option or not) set a given property of a DAG node, the last relevant command overrides earlier commands, as shown in the following examples:

For example, in this DAG:

```
JOB A node.sub
VARS A name="A"
VARS ALL_NODES name="X"
```

the value of *name* for node A will be "X".

In this DAG:

```
JOB A node.sub
VARS A name="A"
VARS ALL_NODES name="X"
VARS A name="foo"
```

the value of *name* for node A will be "foo".

Here is an example DAG using the *ALL_NODES* option:

```
# File: all_ex.dag
    JOB A node.sub
    JOB B node.sub
    JOB C node.sub

    SCRIPT PRE ALL_NODES my_script $JOB

    VARS ALL_NODES name="$(JOB)"

    # This overrides the above VARS command for node B.
    VARS B name="nodeB"

    RETRY all_nodes 3
```

2.10.10 The Rescue DAG

Any time a DAG exits unsuccessfully, DAGMan generates a Rescue DAG. The Rescue DAG records the state of the DAG, with information such as which nodes completed successfully, and the Rescue DAG will be used when the DAG is again submitted. With the Rescue DAG, nodes that have already successfully completed are not re-run.

There are a variety of circumstances under which a Rescue DAG is generated. If a node in the DAG fails, the DAG does not exit immediately; 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 the Rescue DAG and exits. A Rescue DAG is produced on Unix platforms if the *condor_dagman* job itself is removed with *condor_rm*. On Windows, a Rescue DAG is *not* generated in this situation, but re-submitting the original DAG will invoke a lower-level recovery functionality, and it

will produce similar behavior to using a Rescue DAG. A Rescue DAG is produced when a node sets and triggers an *ABORT-DAG-ON* event with a non-zero return value. A zero return value constitutes successful DAG completion, and therefore a Rescue DAG is not generated.

By default, if a Rescue DAG exists, it will be used when the DAG is submitted specifying the original DAG input file. If more than one Rescue DAG exists, the newest one will be used. By using the Rescue DAG, DAGMan will avoid re-running nodes that completed successfully in the previous run. **Note that passing the *-force* option to *condor_submit_dag* or *condor_dagman* will cause *condor_dagman* to not use any existing rescue DAG. This means that previously-completed node jobs will be re-run.**

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 HTCondor job cluster fails, the entire node, including the PRE script, will be re-run. A job cluster may result in the submission of multiple HTCondor jobs. If one of the jobs within the cluster fails, the node fails. Therefore, 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.

Rescue DAG Naming

The file name of the Rescue DAG is obtained by appending the string `.rescue<XXX>` to the original DAG input file name. Values for `<XXX>` start at `001` and continue to `002`, `003`, and beyond. The configuration variable `DAGMAN_MAX_RESCUE_NUM` sets a maximum value for `<XXX>`; see section 3.5.23 for the complete definition of this configuration variable. If you hit the `DAGMAN_MAX_RESCUE_NUM` limit, the last Rescue DAG file is overwritten if the DAG fails again.

If a Rescue DAG exists when the original DAG is re-submitted, the Rescue DAG with the largest magnitude value for `<XXX>` will be used, and its usage is implied.

Example

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 named `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`.

Backtracking to an Older Rescue DAG

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`.

Special Cases

Note that if multiple DAG input files are specified on the *condor_submit_dag* command line, a single Rescue DAG encompassing all of the input DAGs is generated. A DAG file containing splices also produces a single Rescue DAG file. On the other hand, a DAG containing sub-DAGs will produce a separate Rescue DAG for each sub-DAG that is queued (and for the top-level 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 configuration variable `DAGMAN_RESET_RETRIES_UPON_RESCUE`, section 3.5.23, controls whether or not node retries are reset in a Rescue DAG.

Partial versus Full Rescue DAGs

As of HTCondor version 7.7.2, the Rescue DAG file is a partial DAG file, not a complete DAG input file as in the past.

A partial Rescue DAG file contains only information about which nodes are done, and the number of retries remaining for nodes with retries. It does not contain information such as the actual DAG structure and the specification of the submit description file for each node job. Partial Rescue DAGs are automatically parsed in combination with the original DAG input file, which contains information about the DAG structure. This updated implementation means that a change in the original DAG input file, such as specifying a different submit description file for a node job, will take effect when running the partial Rescue DAG. In other words, you can fix mistakes in the original DAG file while still gaining the benefit of using the Rescue DAG.

To use a partial Rescue DAG, you *must* re-run *condor_submit_dag* on the original DAG file, not the Rescue DAG file.

Note that the existence of a DONE specification in a partial Rescue DAG for a node that no longer exists in the original DAG input file is a warning, as opposed to an error, unless the `DAGMAN_USE_STRICT` configuration variable

is set to a value of 1 or higher (which is now the default). Comment out the line with *DONE* in the partial Rescue DAG file to avoid a warning or error.

The previous (prior to version 7.7.2) behavior of producing full DAG input file as the Rescue DAG is obtained by setting the configuration variable `DAGMAN_WRITE_PARTIAL_RESCUE` to the non-default value of `False`. **Note that the option to generate full Rescue DAGs is likely to disappear some time during the 8.3 series.**

To run a full Rescue DAG, either one left over from an older version of DAGMan, or one produced by setting `DAGMAN_WRITE_PARTIAL_RESCUE` to `False`, directly specify the full Rescue DAG file on the command line instead of the original DAG file. For example:

```
condor_submit_dag my.dag.rescue002
```

Attempting to re-submit the original DAG file, if the Rescue DAG file is a complete DAG, will result in a parse failure.

Rescue DAG Generated When There Are Parse Errors

Starting in HTCondor version 7.5.5, passing the **-DumpRescue** option to either *condor_dagman* or *condor_submit_dag* causes *condor_dagman* to output a Rescue DAG file, even if the parsing of a DAG input file fails. In this parse failure case, *condor_dagman* produces a specially named Rescue DAG containing whatever it had successfully parsed up until the point of the parse error. This Rescue DAG may be useful in debugging parse errors in complex DAGs, especially ones using splices. This incomplete Rescue DAG is not meant to be used when resubmitting a failed DAG. Note that this incomplete Rescue DAG generated by the **-DumpRescue** option is a full DAG input file, as produced by versions of HTCondor prior to HTCondor version 7.7.2. It is not a partial Rescue DAG file, regardless of the value of the configuration variable `DAGMAN_WRITE_PARTIAL_RESCUE`.

To avoid confusion between this incomplete Rescue DAG generated in the case of a parse failure and a usable Rescue DAG, a different name is given to the incomplete Rescue DAG. The name appends the string `.parse_failed` to the original DAG input file name. Therefore, if the submission of a DAG with

```
condor_submit_dag  my.dag
```

has a parse failure, the resulting incomplete Rescue DAG will be named `my.dag.parse_failed`.

To further prevent one of these incomplete Rescue DAG files from being used, a line within the file contains the single command *REJECT*. This causes *condor_dagman* to reject the DAG, if used as a DAG input file. This is done because the incomplete Rescue DAG may be a syntactically correct DAG input file. It will be incomplete relative to the original DAG, such that if the incomplete Rescue DAG could be run, it could erroneously be perceived as having successfully executed the desired workflow, when, in fact, it did not.

2.10.11 DAG Recovery

DAG recovery restores the state of a DAG upon resubmission. Recovery is accomplished by reading the `.nodes.log` file that is used to enforce the dependencies of the DAG. The DAG can then continue towards completion.

Recovery is different than a Rescue DAG. Recovery is appropriate when no Rescue DAG has been created. There will be no Rescue DAG if the machine running the *condor_dagman* job crashes, or if the *condor_schedd* daemon crashes, or if the *condor_dagman* job crashes, or if the *condor_dagman* job is placed on hold.

Much of the time, when a not-completed DAG is re-submitted, it will automatically be placed into recovery mode due to the existence and contents of a lock file created as the DAG is first run. In recovery mode, the `.nodes.log` is used to identify nodes that have completed and should not be re-submitted.

DAGMan can be told to work in recovery mode by including the **-DoRecovery** option on the command line, as in the example

```
condor_submit_dag diamond.dag -DoRecovery
```

where `diamond.dag` is the name of the DAG input file.

When debugging a DAG in which something has gone wrong, a first determination is whether a resubmission will use a Rescue DAG or benefit from recovery. The existence of a Rescue DAG means that recovery would be inappropriate. A Rescue DAG is has a file name ending in `.rescue<XXX>`, where `<XXX>` is replaced by a 3-digit number.

Determine if a DAG ever completed (independent of whether it was successful or not) by looking at the last lines of the `.dagman.out` file. If there is a line similar to

```
(condor_DAGMAN) pid 445 EXITING WITH STATUS 0
```

then the DAG completed. This line explains that the *condor_dagman* job finished normally. If there is no line similar to this at the end of the `.dagman.out` file, and output from *condor_q* shows that the *condor_dagman* job for the DAG being debugged is not in the queue, then recovery is indicated.

2.10.12 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 DAG 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 HTCondor. 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 DAG 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 with a period and an integer. For example, the DAG 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 when 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.

2.10.13 Capturing the Status of Nodes in a File

DAGMan can capture the status of the overall DAG and all DAG nodes in a *node status file*, such that the user or a script can monitor this status. This file is periodically rewritten while the DAG runs. To enable this feature, the DAG input file must contain a line with the *NODE_STATUS_FILE* command.

The syntax for a *NODE_STATUS_FILE* command is

NODE_STATUS_FILE *statusFileName* [*minimumUpdateTime*] [**ALWAYS-UPDATE**]

The status file is written on the machine on which the DAG is submitted; its location is given by *statusFileName*, and it may be a full path and file name.

The optional *minimumUpdateTime* specifies the minimum number of seconds that must elapse between updates to the node status file. This setting exists to avoid having DAGMan spend too much time writing the node status file for very large DAGs. If no value is specified, this value defaults to 60 seconds (as of version 8.5.8; previously, it defaulted to 0). The node status file can be updated at most once per `DAGMAN_USER_LOG_SCAN_INTERVAL`, as defined at section 3.5.23, no matter how small the *minimumUpdateTime* value. Also, the node status file will be updated when the DAG finishes, whether successfully or not, even if *minimumUpdateTime* seconds have not elapsed since the last update.

Normally, the node status file is only updated if the status of some nodes has changed since the last time the file was written. However, the optional *ALWAYS-UPDATE* keyword specifies that the node status file should be updated

every time the minimum update time (and DAGMAN_USER_LOG_SCAN_INTERVAL), has passed, even if no nodes have changed status since the last time the file was updated. (The file will change slightly, because timestamps will be updated.) For performance reasons, large DAGs with approximately 10,000 or more nodes are poor candidates for using the *ALWAYS-UPDATE* option.

As an example, if the DAG input file contains the line

```
NODE_STATUS_FILE my.dag.status 30
```

the file `my.dag.status` will be rewritten at intervals of 30 seconds or more.

This node status file is overwritten each time it is updated. Therefore, it only holds information about the *current* status of each node; it does not provide a history of the node status.

NOTE: HTCondor version 8.1.6 changes the format of the node status file.

The node status file is a collection of ClassAds in New ClassAd format. There is one ClassAd for the overall status of the DAG, one ClassAd for the status of each node, and one ClassAd with the time at which the node status file was completed as well as the time of the next update.

Here is an example portion of a node status file:

```
[
  Type = "DagStatus";
  DagFiles = {
    "job_dagman_node_status.dag"
  };
  Timestamp = 1399674138; /* "Fri May  9 17:22:18 2014" */
  DagStatus = 3; /* "STATUS_SUBMITTED ()" */
  NodesTotal = 12;
  NodesDone = 11;
  NodesPre = 0;
  NodesQueued = 1;
  NodesPost = 0;
  NodesReady = 0;
  NodesUnready = 0;
  NodesFailed = 0;
  JobProcsHeld = 0;
  JobProcsIdle = 1;
]
[
  Type = "NodeStatus";
  Node = "A";
  NodeStatus = 5; /* "STATUS_DONE" */
  StatusDetails = "";
  RetryCount = 0;
  JobProcsQueued = 0;
```

```
  JobProcsHeld = 0;
]
...
[
  Type = "NodeStatus";
  Node = "C";
  NodeStatus = 3; /* "STATUS_SUBMITTED" */
  StatusDetails = "idle";
  RetryCount = 0;
  JobProcsQueued = 1;
  JobProcsHeld = 0;
]
[
  Type = "StatusEnd";
  EndTime = 1399674138; /* "Fri May  9 17:22:18 2014" */
  NextUpdate = 1399674141; /* "Fri May  9 17:22:21 2014" */
]
```

Possible `DagStatus` and `NodeStatus` attribute values are:

- 0 (`STATUS_NOT_READY`): At least one parent has not yet finished or the node is a FINAL node.
- 1 (`STATUS_READY`): All parents have finished, but the node is not yet running.
- 2 (`STATUS_PRERUN`): The node's PRE script is running.
- 3 (`STATUS_SUBMITTED`): The node's HTCondor job(s) are in the queue.
- 4 (`STATUS_POSTRUN`): The node's POST script is running.
- 5 (`STATUS_DONE`): The node has completed successfully.
- 6 (`STATUS_ERROR`): The node has failed.

A *NODE_STATUS_FILE* command inside any splice is ignored. If multiple DAG files are specified on the *condor_submit_dag* command line, and more than one specifies a node status file, the first specification takes precedence.

2.10.14 A Machine-Readable Event History, the jobstate.log File

DAGMan can produce a machine-readable history of events. The `jobstate.log` file is designed for use by the Pegasus Workflow Management System, which operates as a layer on top of DAGMan. Pegasus uses the `jobstate.log` file to monitor the state of a workflow. The `jobstate.log` file can used by any automated tool for the monitoring of workflows.

DAGMan produces this file when the command *JOBSTATE_LOG* is in the DAG input file. The syntax for *JOBSTATE_LOG* is

JOBSTATE_LOG *JobstateLogFileName*

No more than one `jobstate.log` file can be created by a single instance of *condor_dagman*. If more than one `jobstate.log` file is specified, the first file name specified will take effect, and a warning will be printed in the `dagman.out` file when subsequent *JOBSTATE_LOG* specifications are parsed. Multiple specifications may exist in the same DAG file, within splices, or within multiple, independent DAGs run with a single *condor_dagman* instance.

The `jobstate.log` file can be considered a filtered version of the `dagman.out` file, in a machine-readable format. It contains the actual node job events that from *condor_dagman*, plus some additional meta-events.

The `jobstate.log` file is different from the node status file, in that the `jobstate.log` file is appended to, rather than being overwritten as the DAG runs. Therefore, it contains a history of the DAG, rather than a snapshot of the current state of the DAG.

There are 5 line types in the `jobstate.log` file. Each line begins with a Unix timestamp in the form of seconds since the Epoch. Fields within each line are separated by a single space character.

DAGMan start This line identifies the *condor_dagman* job. The formatting of the line is

timestamp INTERNAL *** DAGMAN_STARTED *dagmanCondorID* ***

The *dagmanCondorID* field is the *condor_dagman* job's `ClusterId` attribute, a period, and the `ProcId` attribute.

DAGMan exit This line identifies the completion of the *condor_dagman* job. The formatting of the line is

timestamp INTERNAL *** DAGMAN_FINISHED *exitCode* ***

The *exitCode* field is value the *condor_dagman* job returns upon exit.

Recovery started If the *condor_dagman* job goes into recovery mode, this meta-event is printed. During recovery mode, events will only be printed in the file if they were not already printed before recovery mode started. The formatting of the line is

timestamp INTERNAL *** RECOVERY_STARTED ***

Recovery finished or Recovery failure At the end of recovery mode, either a RECOVERY_FINISHED or RECOVERY_FAILURE meta-event will be printed, as appropriate.

The formatting of the line is

timestamp INTERNAL *** RECOVERY_FINISHED ***

or

timestamp INTERNAL *** RECOVERY_FAILURE ***

Normal This line is used for all other event and meta-event types. The formatting of the line is

timestamp JobName eventName condorID jobTag - sequenceNumber

The *JobName* is the name given to the node job as defined in the DAG input file with the command *JOB*. It identifies the node within the DAG.

The *eventName* is one of the many defined event or meta-events given in the lists below.

The *condorID* field is the job's `ClusterId` attribute, a period, and the `ProcId` attribute. There is no *condorID* assigned yet for some meta-events, such as PRE_SCRIPT_STARTED. For these, the dash character ('-') is printed.

The *jobTag* field is defined for the Pegasus workflow manager. Its usage is generalized to be useful to other workflow managers. Pegasus-managed jobs add a line of the following form to their HTCondor submit description file:

```
+pegasus_site = "local"
```

This defines the string `local` as the *jobTag* field.

Generalized usage adds a set of 2 commands to the HTCondor submit description file to define a string as the *jobTag* field:

```
+job_tag_name = "+job_tag_value"
+job_tag_value = "viz"
```

This defines the string `viz` as the *jobTag* field. Without any of these added lines within the HTCondor submit description file, the dash character ('-') is printed for the *jobTag* field.

The *sequenceNumber* is a monotonically-increasing number that starts at one. It is associated with each attempt at running a node. If a node is retried, it gets a new sequence number; a submit failure does not result in a new sequence number. When a Rescue DAG is run, the sequence numbers pick up from where they left off within the previous attempt at running the DAG. Note that this only applies if the Rescue DAG is run automatically or with the *-dorescuefrom* command-line option.

Here is an example of a very simple Pegasus `jobstate.log` file, assuming the example *jobTag* field of `local`:

```
1292620511 INTERNAL *** DAGMAN_STARTED 4972.0 ***
1292620523 NodeA PRE_SCRIPT_STARTED - local - 1
1292620523 NodeA PRE_SCRIPT_SUCCESS - local - 1
1292620525 NodeA SUBMIT 4973.0 local - 1
1292620525 NodeA EXECUTE 4973.0 local - 1
1292620526 NodeA JOB_TERMINATED 4973.0 local - 1
1292620526 NodeA JOB_SUCCESS 0 local - 1
1292620526 NodeA POST_SCRIPT_STARTED 4973.0 local - 1
1292620531 NodeA POST_SCRIPT_TERMINATED 4973.0 local - 1
1292620531 NodeA POST_SCRIPT_SUCCESS 4973.0 local - 1
1292620535 INTERNAL *** DAGMAN_FINISHED 0 ***
```

Events defining the eventName field

- SUBMIT
- EXECUTE
- EXECUTABLE_ERROR
- CHECKPOINTED
- JOB_EVICTED

- JOB_TERMINATED
- IMAGE_SIZE
- SHADOW_EXCEPTION
- GENERIC
- JOB_ABORTED
- JOB_SUSPENDED
- JOB_UNSUSPENDED
- JOB_HELD
- JOB_RELEASED
- NODE_EXECUTE
- NODE_TERMINATED
- POST_SCRIPT_TERMINATED
- GLOBUS_SUBMIT
- GLOBUS_SUBMIT_FAILED
- GLOBUS_RESOURCE_UP
- GLOBUS_RESOURCE_DOWN
- REMOTE_ERROR
- JOB_DISCONNECTED
- JOB_RECONNECTED
- JOB_RECONNECT_FAILED
- GRID_RESOURCE_UP
- GRID_RESOURCE_DOWN
- GRID_SUBMIT
- JOB_AD_INFORMATION
- JOB_STATUS_UNKNOWN
- JOB_STATUS_KNOWN
- JOB_STAGE_IN
- JOB_STAGE_OUT

Meta-Events defining the eventName field

- SUBMIT_FAILURE
- JOB_SUCCESS
- JOB_FAILURE
- PRE_SCRIPT_STARTED
- PRE_SCRIPT_SUCCESS
- PRE_SCRIPT_FAILURE
- POST_SCRIPT_STARTED

- POST_SCRIPT_SUCCESS
- POST_SCRIPT_FAILURE
- DAGMAN_STARTED
- DAGMAN_FINISHED
- RECOVERY_STARTED
- RECOVERY_FINISHED
- RECOVERY_FAILURE

2.10.15 Status Information for the DAG in a ClassAd

The *condor_dagman* job places information about the status of the DAG into its own job ClassAd. The attributes are fully described at section 12. The attributes are

- `DAG_NodesTotal`
- `DAG_NodesDone`
- `DAG_NodesPrerun`
- `DAG_NodesQueued`
- `DAG_NodesPostrun`
- `DAG_NodesReady`
- `DAG_NodesFailed`
- `DAG_NodesUnready`
- `DAG_Status`
- `DAG_InRecovery`

Note that most of this information is also available in the `dagman.out` file as described in section 2.10.7.

2.10.16 Utilizing the Power of DAGMan for Large Numbers of Jobs

Using DAGMan is recommended when submitting large numbers of jobs. The recommendation holds whether the jobs are represented by a DAG due to dependencies, or all the jobs are independent of each other, such as they might be in a parameter sweep. DAGMan offers:

Throttling Throttling limits the number of submitted jobs at any point in time.

Retry of jobs that fail This is a useful tool when an intermittent error may cause a job to fail or may cause a job to fail to run to completion when attempted at one point in time, but not at another point in time. The conditions under which retry occurs are user-defined. In addition, the administrative support that facilitates the rerunning of only those jobs that fail is automatically generated.

Scripts associated with node jobs PRE and POST scripts run on the submit host before and/or after the execution of specified node jobs.

Each of these capabilities is described in detail within this manual section about DAGMan. To make effective use of DAGMan, there is no way around reading the appropriate subsections.

To run DAGMan with large numbers of independent jobs, there are generally two ways of organizing and specifying the files that control the jobs. Both ways presume that programs or scripts will generate needed files, because the file contents are either large and repetitive, or because there are a large number of similar files to be generated representing the large numbers of jobs. The two file types needed are the DAG input file and the submit description file(s) for the HTCondor jobs represented. Each of the two ways is presented separately:

A unique submit description file for each of the many jobs. A single DAG input file lists each of the jobs and specifies a distinct submit description file for each job. The DAG input file is simple to generate, as it chooses an identifier for each job and names the submit description file. For example, the simplest DAG input file for a set of 1000 independent jobs, as might be part of a parameter sweep, appears as

```
# file sweep.dag
JOB job0 job0.submit
JOB job1 job1.submit
JOB job2 job2.submit
.
.
.
JOB job999 job999.submit
```

There are 1000 submit description files, with a unique one for each of the job<N> jobs. Assuming that all files associated with this set of jobs are in the same directory, and that files continue the same naming and numbering scheme, the submit description file for `job6.submit` might appear as

```
# file job6.submit
universe = vanilla
executable = /path/to/executable
log = job6.log
input = job6.in
output = job6.out
arguments = "-file job6.out"
queue
```

Submission of the entire set of jobs uses the command line

```
condor_submit_dag sweep.dag
```

A benefit to having unique submit description files for each of the jobs is that they are available if one of the jobs needs to be submitted individually. A drawback to having unique submit description files for each of the jobs is that there are lots of submit description files.

Single submit description file. A single HTCondor submit description file might be used for all the many jobs of the parameter sweep. To distinguish the jobs and their associated distinct input and output files, the DAG input file assigns a unique identifier with the *VARS* command.

```
# file sweep.dag
JOB job0 common.submit
VARS job0 runnumber="0"
JOB job1 common.submit
VARS job1 runnumber="1"
JOB job2 common.submit
VARS job2 runnumber="2"
.
.
.
JOB job999 common.submit
VARS job999 runnumber="999"
```

The single submit description file for all these jobs utilizes the `runnumber` variable value in its identification of the job's files. This submit description file might appear as

```
# file common.submit
universe = vanilla
executable = /path/to/executable
log = wholeDAG.log
input = job$(runnumber).in
output = job$(runnumber).out
arguments = "-$(runnumber)"
queue
```

The job with `runnumber="8"` expects to find its input file `job8.in` in the single, common directory, and it sends its output to `job8.out`. The single log for all job events of the entire DAG is `wholeDAG.log`. Using one file for the entire DAG meets the limitation that no macro substitution may be specified for the job log file, and it is likely more efficient as well. This node's executable is invoked with

```
/path/to/executable -8
```

These examples work well with respect to file naming and file location when there are less than several thousand jobs submitted as part of a DAG. The large numbers of files per directory becomes an issue when there are greater than several thousand jobs submitted as part of a DAG. In this case, consider a more hierarchical structure for the files instead of a single directory. Introduce a separate directory for each run. For example, if there were 10,000 jobs, there would be 10,000 directories, one for each of these jobs. The directories are presumed to be generated and populated by programs or scripts that, like the previous examples, utilize a run number. Each of these directories named utilizing the run number will be used for the input, output, and log files for one of the many jobs.

As an example, for this set of 10,000 jobs and directories, assume that there is a run number of 600. The directory will be named `dir600`, and it will hold the 3 files called `in`, `out`, and `log`, representing the input, output, and HTCondor job log files associated with run number 600.

The DAG input file sets a variable representing the run number, as in the previous example:

```
# file biggersweep.dag
JOB job0 bigger.submit
VARS job0 runnumber="0"
JOB job1 bigger.submit
VARS job1 runnumber="1"
JOB job2 bigger.submit
VARS job2 runnumber="2"
.
.
.
JOB job9999 bigger.submit
VARS job9999 runnumber="9999"
```

A single HTCondor submit description file may be written. It resides in the same directory as the DAG input file.

```
# file bigger.submit
universe = vanilla
executable = /path/to/executable
log = log
input = in
output = out
arguments = "-$(runnumber)"
initialdir = dir$(runnumber)
queue
```

One item to care about with this set up is the underlying file system for the pool. The transfer of files (or not) when using **initialdir** differs based upon the job **universe** and whether or not there is a shared file system. See section 11 for the details on the submit command **initialdir**.

Submission of this set of jobs is no different than the previous examples. With the current working directory the same as the one containing the submit description file, the DAG input file, and the subdirectories,

```
condor_submit_dag biggersweep.dag
```

2.10.17 Workflow Metrics

condor_dagman may report workflow metrics to one or more HTTP servers. This capability is currently only used for workflows run under *Pegasus*. The reporting is disabled by setting the `CONDOR_DEVELOPERS` configuration variable

to NONE, or by setting the PEGASUS_METRICS environment variable to any value other than True (case-insensitive) or 1. The dagman.out file will indicate whether or not metrics were reported.

For every DAG, a metrics file is created independent of the reporting of those metrics. This metrics file is named <dag_file_name>.metrics, where <dag_file_name> is the name of the DAG input file. In a workflow with nested DAGs, each nested DAG will create its own metrics file.

Here is an example metrics output file:

```
{
    "client":"condor_dagman",
    "version":"8.1.0",
    "planner":"/lfs1/devel/Pegasus/pegasus/bin/pegasus-plan",
    "planner_version":"4.3.0cvs",
    "type":"metrics",
    "wf_uuid":"htcondor-test-job_dagman_metrics-A-subdag",
    "root_wf_uuid":"htcondor-test-job_dagman_metrics-A",
    "start_time":1375313459.603,
    "end_time":1375313491.498,
    "duration":31.895,
    "exitcode":1,
    "dagman_id":"26",
    "parent_dagman_id":"11",
    "rescue_dag_number":0,
    "jobs":4,
    "jobs_failed":1,
    "jobs_succeeded":3,
    "dag_jobs":0,
    "dag_jobs_failed":0,
    "dag_jobs_succeeded":0,
    "total_jobs":4,
    "total_jobs_run":4,
    "total_job_time":0.000,
    "dag_status":2
}
```

Here is an explanation of each of the items in the file:

- client: the name of the client workflow software; in the example, it is "condor_dagman"
- version: the version of the client workflow software
- planner: the workflow planner, as read from the braindump.txt file
- planner_version: the planner software version, as read from the braindump.txt file
- type: the type of data, "metrics"

- `wf_uuid`: the workflow ID, generated by *pegasus-plan*, as read from the `braindump.txt` file
- `root_wf_uuid`: the root workflow ID, which is relevant for nested workflows. It is generated by *pegasus-plan*, as read from the `braindump.txt` file.
- `start_time`: the start time of the client, in epoch seconds, with millisecond precision
- `end_time`: the end time of the client, in epoch seconds, with millisecond precision
- `duration`: the duration of the client, in seconds, with millisecond precision
- `exitcode`: the *condor_dagman* exit code
- `dagman_id`: the value of the `ClusterId` attribute of the *condor_dagman* instance
- `parent_dagman_id`: the value of the `ClusterId` attribute of the parent *condor_dagman* instance of this DAG; empty if this DAG is *not* a SUBDAG
- `rescue_dag_number`: the number of the Rescue DAG being run, or 0 if not running a Rescue DAG
- `jobs`: the number of nodes in the DAG input file, not including SUBDAG nodes
- `jobs_failed`: the number of failed nodes in the workflow, not including SUBDAG nodes
- `jobs_succeeded`: the number of successful nodes in the workflow, not including SUBDAG nodes; this includes jobs that succeeded after retries
- `dag_jobs`: the number of SUBDAG nodes in the DAG input file
- `dag_jobs_failed`: the number of SUBDAG nodes that failed
- `dag_jobs_succeeded`: the number of SUBDAG nodes that succeeded
- `total_jobs`: the total number of jobs in the DAG input file
- `total_jobs_run`: the total number of nodes executed in a DAG. It should be equal to `jobs_succeeded + jobs_failed + dag_jobs_succeeded + dag_jobs_failed`
- `total_job_time`: the sum of the time between the first execute event and the terminated event for all jobs that are not SUBDAGs
- `dag_status`: the final status of the DAG, with values
  - `0`: OK
  - `1`: error; an error condition different than those listed here
  - `2`: one or more nodes in the DAG have failed
  - `3`: the DAG has been aborted by an ABORT-DAG-ON specification
  - `4`: removed; the DAG has been removed by *condor_rm*
  - `5`: a cycle was found in the DAG
  - `6`: the DAG has been halted; see section 2.10.8 for an explanation of halting a DAG

Note that any `dag_status` other than 0 corresponds to a non-zero exit code.

The `braindump.txt` file is generated by *pegasus-plan*; the name of the `braindump.txt` file is specified with the `PEGASUS_BRAINDUMP_FILE` environment variable. If not specified, the file name defaults to `braindump.txt`, and it is placed in the current directory.

Note that the `total_job_time` value is always zero, because the calculation of that value has not yet been implemented.

If a DAG succeeds, but the metrics reporting fails, the DAG is still considered successful.

The metrics are reported only at the end of a DAG run. This includes reporting the metrics if the *condor_dagman* job is removed, or if the DAG drains from the queue because of being halted by a halt file.

The metrics are reported by the *condor_dagman_metrics_reporter* executable as described in the manual page at 780.

2.10.18 DAGMan and Accounting Groups

As of version 8.5.6, *condor_dagman* propagates **accounting_group** and **accounting_group_user** values specified for *condor_dagman* itself to all jobs within the DAG (including sub-DAGs).

The **accounting_group** and **accounting_group_user** values can be specified using the **-append** flag to *condor_submit_dag*, for example:

```
condor_submit_dag -append accounting_group=group_physics -append accounting_group_user=a
```

See section 3.6.7 for a discussion of group accounting and section 3.6.8 for a discussion of accounting groups with hierarchical group quotas.

2.11 Virtual Machine Applications

The **vm** universe facilitates an HTCondor job that matches and then lands a disk image on an execute machine within an HTCondor pool. This disk image is intended to be a virtual machine. In this manner, the virtual machine is the job to be executed.

This section describes this type of HTCondor job. See section 3.5.25 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*. Other commands specific to the type of virtual machine software identify the disk image.

VMware, Xen, and KVM virtual machine software are supported. As these differ from each other, the submit description file specifies one of

```
vm_type = vmware
```

or

```
vm_type = xen
```

or

```
vm_type = kvm
```

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

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, HTCondor 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.

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 HTCondor 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
```

The required disk image must be identified for a virtual machine. This **vm_disk** command specifies a list of comma-separated files. Each disk file is specified by colon-separated fields. The first field is the path and file name of the disk file. The second field specifies the device. The third field specifies permissions, and the optional fourth specifies the format. Here is an example that identifies a single file:

```
vm_disk = swap.img:sda2:w:raw
```

If HTCondor will be transferring the disk file, then the file name given in **vm_disk** should not contain any path information. Otherwise, the full path to the file should be given.

Setting values in the submit description file for some commands have consequences for the virtual machine description file. These commands are

- **vm_memory**
- **vm_macaddr**
- **vm_networking**
- **vm_networking_type**
- **vm_disk**

For VMware virtual machines, setting values for these commands causes HTCondor to modify the `.vmx` file, overwriting existing values. For KVM and Xen virtual machines, HTCondor uses these values when it produces the description file.

For Xen and KVM jobs, if any files need to be transferred from the submit machine to the machine where the **vm** universe job will execute, HTCondor must be explicitly told to do so with the standard file transfer attributes:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /myxen/diskfile.img,/myxen/swap.img
```

Any and all needed files that will not accessible directly from the machines where the job may execute must be listed.

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) to where VMware-specific files and applications reside. One example of a VMware-specific application is the VMDK files, which form a virtual hard drive (disk image) for the virtual machine. VMX files containing the primary configuration for the virtual machine would also be in this directory.

HTCondor 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`, HTCondor does transfer the contents of the directory. With a value of `False`, HTCondor 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, HTCondor 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, HTCondor 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`.

Because *VMware Player* does not support snapshots, machines using *VMware Player* may only run **vm** jobs that set **vmware_snapshot_disk** to `False`. These jobs will also set **vmware_should_transfer_files** to `True`. A job using *VMware Player* will go on hold if it attempts to use a snapshot. The pool administrator should have configured the pool such that machines will not start jobs they can not run.

Note that if snapshot disks are requested and file transfer is not being used, the **vmware_dir** setting given in the submit description file should not contain any symbolic link path components, as described on the https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToAdminRecipes page under the answer to why VMware jobs with symbolic links fail.

Here is a sample submit description file for a VMware virtual machine:

```
universe                       = vm
executable                     = vmware_sample_job
log                            = simple.vm.log.txt
vm_type                        = vmware
vm_memory                      = 64
vmware_dir                     = C:\condor-test
vmware_should_transfer_files = True
queue
```

This sample uses the **vmware_dir** command to identify the location of the disk image to be executed as an HTCondor job. The contents of this directory are transferred to the machine assigned to execute the HTCondor job.

Xen-Specific Submit Commands

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 = included` implies that the kernel is to be found in disk image given by the definition of the single file specified in **vm_disk**.
2. `xen_kernel = path-to-kernel` gives the 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 **transfer_input_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`.

2.11.2 Checkpoints

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. The submit command to create checkpoints is

```
vm_checkpoint = true
```

Without this command, no checkpoints are created (by default). With the command, a checkpoint is created any time the **vm** universe jobs is evicted from the machine upon which it is executing. This occurs as a result of the machine configuration indicating that it will no longer execute this job.

vm universe jobs can *not* use a checkpoint server.

Periodic creation of checkpoints is not supported at this time.

Enabling both networking and checkpointing for a **vm** universe job can cause networking problems when the job restarts, particularly if the job migrates to a different machine. *condor_submit* will normally reject such jobs. To enable both, then add the command

```
when_to_transfer_output = ON_EXIT_OR_EVICT
```

Take care with respect to the use of network connections within the virtual machine and their interaction with checkpoints. Open network connections at the time of the checkpoint will likely be lost when the checkpoint is subsequently used to resume execution of the virtual machine. This occurs whether or not the execution resumes on the same machine or a different one within the HTCondor pool.

2.11.3 Disk Images

VMware on Windows and Linux

Following the platform-specific guest OS installation instructions found at http://partnerweb.vmware.com/GOSIG/home.html, creates a VMware disk image.

Xen and KVM

While the following web page contains instructions specific to Fedora on how to create a virtual guest image, it should provide a good starting point for other platforms as well.

http://fedoraproject.org/wiki/Virtualization_Quick_Start

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 HTCondor 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. A script, executed with the proper authorization level, is the likely source of the shut down commands.

Under a Windows 2000, Windows XP, or Vista virtual machine, an administrator issues the command

```
shutdown -s -t 01
```

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 HTCondor, before a **vm** universe job is submitted.

2.11.5 Failures to Launch

It is not uncommon for a **vm** universe job to fail to launch because of a problem with the execute machine. In these cases, HTCondor will reschedule the job and note, in its user event log (if requested), the reason for the failure and that the job will be rescheduled. The reason is unlikely to be directly useful to you as an HTCondor user, but may help your HTCondor administrator understand the problem.

If the VM fails to launch for other reasons, the job will be placed on hold and the reason placed in the job ClassAd's `HoldReason` attribute. The following table may help in understanding such reasons.

`VMGAHP_ERR_JOBCLASSAD_NO_VM_MEMORY_PARAM`

```
    The attribute JobVMMemory was not set in the job ad sent to the
VM GAHP.  HTCondor will usually prevent you from submitting a VM universe job
without JobVMMemory set.  Examine your job and verify that JobVMMemory is set.
If it is, please contact your administrator.
```

`VMGAHP_ERR_JOBCLASSAD_NO_VMWARE_VMX_PARAM`

```
    The attribute VMPARAM_VMware_Dir was not set in the job ad sent to the
VM GAHP.  HTCondor will usually set this attribute when you submit a valid
VMWare job (it is derived from vmware_dir).  If you used condor_submit to
submit this job, contact your administrator.  Otherwise, examine your job
and verify that VMPARAM_VMware_Dir is set.  If it is, contact your
administrator.
```

`VMGAHP_ERR_JOBCLASSAD_KVM_NO_DISK_PARAM`

```
    The attribute VMPARAM_vm_Disk was not set in the job ad sent to the
VM GAHP.  HTCondor will usually set this attribute when you submit a valid
KVM job (it is derived from vm_disk).  Examine your job and verify that
VMPARAM_vm_Disk is set.  If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_KVM_INVALID_DISK_PARAM

    The attribute vm_disk was invalid.  Please consult the manual,
or the condor_submit man page, for information about the syntax of
vm_disk.  A syntactically correct value may be invalid if the
on-disk permissions of a file specified in it do not match the requested
permissions.  Presently, files not transferred to the root of the working
directory must be specified with full paths.

VMGAHP_ERR_JOBCLASSAD_KVM_MISMATCHED_CHECKPOINT

    KVM jobs can not presently checkpoint if any of their disk files are not
on a shared filesystem.  Files on a shared filesystem must be specified in
vm_disk with full paths.

VMGAHP_ERR_JOBCLASSAD_XEN_NO_KERNEL_PARAM

    The attribute VMPARAM_Xen_Kernel was not set in the job ad sent to the
VM GAHP.  HTCondor will usually set this attribute when you submit a valid
Xen job (it is derived from xen_kernel).  Examine your job and verify that
VMPARAM_Xen_Kernel is set.  If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_MISMATCHED_HARDWARE_VT

    Don't use 'vmx' as the name of your kernel image.  Pick something else and
change xen_kernel to match.

VMGAHP_ERR_JOBCLASSAD_XEN_KERNEL_NOT_FOUND

    HTCondor could not read from the file specified by xen_kernel.
Check the path and the file's permissions.  If it's on a shared filesystem,
you may need to alter your job's requirements expression to ensure the
filesystem's availability.

VMGAHP_ERR_JOBCLASSAD_XEN_INITRD_NOT_FOUND

    HTCondor could not read from the file specified by xen_initrd.
Check the path and the file's permissions.  If it's on a shared filesystem,
you may need to alter your job's requirements expression to ensure the
filesystem's availability.
```

```
VMGAHP_ERR_JOBCLASSAD_XEN_NO_ROOT_DEVICE_PARAM

    The attribute VMPARAM_Xen_Root was not set in the job ad sent to the
VM GAHP.  HTCondor will usually set this attribute when you submit a valid
Xen job (it is derived from xen_root).  Examine your job and verify that
VMPARAM_Xen_Root is set.  If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_XEN_NO_DISK_PARAM

    The attribute VMPARAM_vm_Disk was not set in the job ad sent to the
VM GAHP.  HTCondor will usually set this attribute when you submit a valid
Xen job (it is derived from vm_disk).  Examine your job and verify that
VMPARAM_vm_Disk is set.  If it is, please contact your administrator.

VMGAHP_ERR_JOBCLASSAD_XEN_INVALID_DISK_PARAM

    The attribute vm_disk was invalid.  Please consult the manual,
or the condor_submit man page, for information about the syntax of
vm_disk.  A syntactically correct value may be invalid if the
on-disk permissions of a file specified in it do not match the requested
permissions.  Presently, files not transferred to the root of the working
directory must be specified with full paths.

VMGAHP_ERR_JOBCLASSAD_XEN_MISMATCHED_CHECKPOINT

    Xen jobs can not presently checkpoint if any of their disk files are not
on a shared filesystem.  Files on a shared filesystem must be specified in
vm_disk with full paths.
```

2.12 Docker Universe Applications

A docker universe job instantiates a Docker container from a Docker image, and HTCondor manages the running of that container as an HTCondor job, on an execute machine. This running container can then be managed as any HTCondor job. For example, it can be scheduled, removed, put on hold, or be part of a workflow managed by DAGMan.

The docker universe job will only be matched with an execute host that advertises its capability to run docker universe jobs. When an execute machine with docker support starts, the machine checks to see if the *docker* command is available and has the correct settings for HTCondor. Docker support is advertised if available and if it has the correct settings.

The image from which the container is instantiated is defined by specifying a Docker image with the submit command **docker_image**. This image must be pre-staged on a docker hub that the execute machine can access.

After submission, the job is treated much the same way as a vanilla universe job. Details of file transfer are the same as applied to the vanilla universe. One of the benefits of Docker containers is the file system isolation they provide. Each container has a distinct file system, from the root on down, and this file system is completely independent of the file system on the host machine. The container does not share a file system with either the execute host or the submit host, with the exception of the scratch directory, which is volume mounted to the host, and is the initial working directory of the job. Optionally, the administrator may configure other directories from the host machine to be volume mounted, and thus visible inside the container. See the docker section of the administrator's manual for details.

Therefore, the submit description file should contain the submit command

```
should_transfer_files = YES
```

With this command, all input and output files will be transferred as required to and from the scratch directory mounted as a Docker volume.

If no **executable** is specified in the submit description file, it is presumed that the Docker container has a default command to run.

When the job completes, is held, evicted, or is otherwise removed from the machine, the container will be removed.

Here is a complete submit description file for a sample docker universe job:

```
universe                = docker
docker_image            = debian
executable              = /bin/cat
arguments               = /etc/hosts
should_transfer_files   = YES
when_to_transfer_output = ON_EXIT
output                  = out.$(Process)
error                   = err.$(Process)
log                     = log.$(Process)
request_memory          = 100M
queue 1
```

A debian container is the HTCondor job, and it runs the */bin/cat* program on the `/etc/hosts` file before exiting.

2.13 Time Scheduling for Job Execution

Jobs may be scheduled to begin execution at a specified time in the future with HTCondor'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.

2.13.1 Job Deferral

Job deferral allows the specification of the exact date and time at which a job is to begin executing. HTCondor 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 define the job to allow some flexibility in the execution of jobs that miss their execution time.

Deferred Execution Time

A job's deferral time is the exact time that HTCondor 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 HTCondor will begin to execute the job.

After a job is matched and all of its files have been transferred to an execution machine, HTCondor checks to see if the job's ClassAd contains a deferral time. If it does, HTCondor calculates the number of seconds between the execution machine's current system time and 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 in the Running state. 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 HTCondor'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, HTCondor begins execution for the job, but immediately suspends it.

The deferral time is specified in the job's submit description file with the command **deferral_time**.

Deferral Window

If a job arrives at its execution machine after the deferral time has passed, 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, HTCondor 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.

The deferral window is specified in the job's submit description file with the command **deferral_window**.

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 depends on the options provided within that flavor of Unix. In some, it appears as

```
%  date --date "MM/DD/YYYY HH:MM:SS" +%s
```

and in others, it appears as

```
%  date -d "YYYY-MM-DD 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 after submission before beginning execution:

```
deferral_time = (QDate + 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
```

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 2.3: The list of submit commands and their value ranges.

Limitations

There are some limitations to HTCondor'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.

2.13.2 CronTab Scheduling

HTCondor's CronTab scheduling functionality allows jobs to be scheduled to execute 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. As such, HTCondor developers are fond of referring to CronTab scheduling as *Crondor*. The scheduling of jobs using HTCondor's CronTab feature calculates and utilizes the `DeferralTime` ClassAd attribute.

Also, unlike the Unix *cron* daemon, HTCondor 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 of specifications within the submit description file. HTCondor uses these to calculate a `DeferralTime` for the job.

Table 2.3 lists the submit commands and acceptable values for these commands. At least one of these must be defined in order for HTCondor 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
```

where this **cron_minute** example 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
```

where this **cron_minute** example specifies every five minutes within the specified range to represent (10,15,20,25,30), and **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 HTCondor to make necessary preparations before the deferral time occurs.

Consider the submit description file example that includes

```
cron_minute = 0
cron_hour = *
cron_prep_time = 300
```

The job is scheduled to begin execution at the top of every hour. Note that the setting of **cron_hour** in this example is not required, as the default value will be `*`, specifying any and every hour of the day. 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. HTCondor 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_minute = 0
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 HTCondor, use of at least one of the submit description file commands beginning with **cron_** causes HTCondor 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 HTCondor operates on the job queue at times that are independent of job events, such as when job execution completes. Therefore, HTCondor may operate on the job queue just after a job's deferral time states that it is to begin execution. HTCondor attempts to start a job when the following pseudo-code boolean expression evaluates to `True`:

```
( time() + SCHEDD_INTERVAL ) >= ( DeferralTime - CronPrepTime )
```

If the `time()` plus the number of seconds until the next time HTCondor 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 HTCondor 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 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 HTCondor 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.
- HTCondor 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 HTCondor job; if HTCondor 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.

2.14 Special Environment Considerations

2.14.1 AFS

The HTCondor daemons do not run authenticated to AFS; they do not possess AFS tokens. Therefore, no child process of HTCondor 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 HTCondor 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 HTCondor can access them using remote system calls (if this is a standard universe job), or copy them to an NFS volume.
2. If the files must be kept 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 this is a standard universe job, then the host ACL needs to give read/write permission to any process on the submit machine. If this is a vanilla universe job, then 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 Center for High Throughput Computing hopes to improve upon how HTCondor deals with AFS authentication in a subsequent release.

Please see section 3.14.1 for further discussion of this problem.

2.14.2 NFS

If the current working directory when a job is submitted is accessed via an NFS automounter, HTCondor may have problems if the automounter later decides to unmount the volume before the 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 the job, use the submit command **initialdir** 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
```

HTCondor attempts to flush the NFS cache on a submit machine in order to refresh a job's initial working directory. This allows files written by the job into an NFS mounted initial working directory to be immediately visible on the submit machine. Since the flush operation can require multiple round trips to the NFS server, it is expensive. Therefore, a job may disable the flushing by setting

```
+IwdFlushNFSCache = False
```

in the job's submit description file. See page 995 for a definition of the job ClassAd attribute.

2.14.3 HTCondor Daemons That Do Not Run as root

HTCondor is normally installed such that the HTCondor daemons have root permission. This allows HTCondor to run the *condor_shadow* daemon and the job with the submitting user's UID and file access rights. When HTCondor is started as root, HTCondor jobs can access whatever files the user that submits the jobs can.

However, it is possible that the HTCondor installation does not have root access, or has decided not to run the daemons as root. That is unfortunate, since HTCondor is designed to be run as root. To see if HTCondor is running as root on a specific machine, use the command

```
condor_status -master -l <machine-name>
```

where `<machine-name>` is the name of the specified machine. This command displays the full condor_master ClassAd; if the attribute `RealUid` equals zero, then the HTCondor daemons are indeed running with root access. If the `RealUid` attribute is not zero, then the HTCondor daemons do not have root access.

NOTE: The Unix program *ps* is *not* an effective method of determining if HTCondor 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, as it will be to perform a privileged operation.

If daemons are not running with root access, make any and all files and/or directories that the 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 from which the HTCondor job is submitted.

2.14.4 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 40 minutes exists for a job's ClassAd attribute `JobLeaseDuration`, or this attribute may be set in the submit description file, using **job_lease_duration**, 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 the user is 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.

2.15 Potential Problems

2.15.1 Renaming of argv[0]

When HTCondor 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 an HTCondor job.

Unfortunately, some programs read argv[0] expecting their own program name and get confused if they find something unexpected like condor_exec.

Chapter 3

Administrators' Manual

3.1 Introduction

This is the HTCondor Administrator's Manual. Its purpose is to aid in the installation and administration of an HTCondor pool. For help on using HTCondor, see the HTCondor User's Manual.

An HTCondor 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 HTCondor is to match waiting requests with available resources. Every part of HTCondor 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 one who sets the policy for the use of the machine. This person has absolute power over the use of the machine, and HTCondor goes out of its way to minimize the impact on this owner caused by HTCondor. It is up to the resource owner to define a policy for when HTCondor requests will serviced and when they will be denied.

Each resource request has an owner as well: the user who submitted the job. These people want HTCondor 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 HTCondor administrator is to configure the HTCondor pool to find the happy medium that keeps both resource owners and users of resources satisfied. The purpose of this manual is to relate the mechanisms that HTCondor provides to enable the administrator to find this happy medium.

3.1.1 The Different Roles a Machine Can Play

Every machine in an HTCondor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by a single machine in the 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 the pool. This 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 HTCondor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the HTCondor 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 the pool, since these pool machines all send updates over the network to the central manager.

Execute Any machine in the pool, including the central manager, can be configured as to whether or not it should execute HTCondor jobs. Obviously, some of the machines will have to serve this function, or the pool will not be useful. Being an execute machine does not require lots of resources. About the only resource that might matter is disk space. In general the more resources a machine has in terms of swap space, memory, number of CPUs, the larger variety of resource requests it can serve.

Submit Any machine in the pool, including the central manager, can be configured as to whether or not it should allow HTCondor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First, every submitted job that is currently running on a remote machine runs a process on the submit machine. As a result, lots of running jobs will need a fair amount of swap space and/or real memory. In addition, the checkpoint files from standard universe jobs are stored on the local disk of the submit machine. If these jobs have a large memory image and there are a lot of them, the submit machine will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated by using a checkpoint server, however the binaries of the jobs are still stored on the submit machine.

Checkpoint Server Machines in the pool can be configured to act as checkpoint servers. This is optional, and is not part of the standard HTCondor binary distribution. A checkpoint server is a machine that stores checkpoint files for sets of jobs. A machine with this role should have lots of disk space and a good network connection to the rest of the pool, as the traffic can be quite heavy.

3.1.2 The HTCondor Daemons

The following list describes all the daemons and programs that could be started under HTCondor and what they do:

condor_master This daemon is responsible for keeping all the rest of the HTCondor daemons running on each machine in the pool. It spawns the other daemons, and it periodically checks to see if there are new binaries installed for any of them. If there are, the *condor_master* daemon will restart the affected daemons. In addition, if any daemon crashes, the *condor_master* will send e-mail to the HTCondor administrator of the pool and restart the daemon. The *condor_master* also supports various administrative commands that enable the administrator to start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in the pool, regardless of the functions that each machine is performing.

condor_startd This daemon represents a given resource to the HTCondor pool, as a machine capable of running jobs. It advertises certain attributes about machine that are used to match it with pending resource requests. The *condor_startd* will run on any machine in the pool that is to be able to execute jobs. It is responsible for enforcing

the policy that the resource owner configures, which determines under what conditions jobs will be started, suspended, resumed, vacated, or killed. When the *condor_startd* is ready to execute an HTCondor job, it spawns the *condor_starter*.

condor_starter This daemon is the entity that actually spawns the HTCondor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the *condor_starter* notices this, sends back any status information to the submitting machine, and exits.

condor_schedd This daemon represents resource requests to the HTCondor pool. Any machine that is to be a submit machine needs to have a *condor_schedd* running. When users submit jobs, the jobs go to the *condor_schedd*, where they are stored in the *job queue*. The *condor_schedd* manages the job queue. Various tools to view and manipulate the job queue, such as *condor_submit*, *condor_q*, and *condor_rm*, all must connect to the *condor_schedd* to do their work. If the *condor_schedd* is not running on a given machine, none of these commands will work.

The *condor_schedd* advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a job has been matched with a given resource, the *condor_schedd* spawns a *condor_shadow* daemon to serve that particular request.

condor_shadow This daemon runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for HTCondor'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 performs the system call on the submit machine, and the result is sent back over the network to the job on the execute machine. In addition, the *condor_shadow* is responsible for making decisions about the request, such as where checkpoint files should be stored, and how certain files should be accessed.

condor_collector This daemon is responsible for collecting all the information about the status of an HTCondor pool. All other daemons periodically send ClassAd updates to the *condor_collector*. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool. The *condor_status* command can be used to query the *condor_collector* for specific information about various parts of HTCondor. In addition, the HTCondor daemons themselves query the *condor_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 HTCondor system. Periodically, the *condor_negotiator* begins a *negotiation cycle*, where it queries the *condor_collector* for the current state of all the resources in the pool. It contacts each *condor_schedd* that has waiting resource requests in priority order, and tries to match available resources with those requests. The *condor_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 *condor_negotiator* can preempt that resource and match it with the user with better priority.

NOTE: A higher numerical value of the user priority in HTCondor translate into worse priority for that user. The best priority is 0.5, the lowest numerical value, and this priority gets worse as this number grows.

condor_kbdd This daemon is used on both Linux and Windows platforms. On those platforms, the *condor_startd* frequently cannot determine console (keyboard or mouse) activity directly from the system, and requires a separate process to do so. On Linux, the *condor_kbdd* connects to the X Server and periodically checks to

see if there has been any activity. On Windows, the *condor_kbdd* runs as the logged-in user and registers with the system to receive keyboard and mouse events. When it detects console activity, the *condor_kbdd* sends a command to the *condor_startd*. That way, the *condor_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 The checkpoint server services requests to store and retrieve checkpoint files. If the pool is configured to use a checkpoint server, but that machine or the server itself is down, HTCondor will revert to sending the checkpoint files for a given job back to the submit machine.

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_transferer This short lived daemon is invoked by the *condor_replication* daemon to accomplish the task of transferring a state file before exiting.

condor_procd This daemon controls and monitors process families within HTCondor. Its use is optional in general, but it must be used if group-ID based tracking (see Section 3.14.11) 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.

condor_rooster This daemon wakes hibernating machines based upon configuration details.

condor_defrag This daemon manages the draining of machines with fragmented partitionable slots, so that they become available for jobs requiring a whole machine or larger fraction of a machine.

condor_shared_port This daemon listens for incoming TCP packets on behalf of HTCondor daemons, thereby reducing the number of required ports that must be opened when HTCondor is accessible through a firewall.

When compiled from source code, the following daemons may be compiled in to provide optional functionality.

condor_hdfs This daemon manages the configuration of a Hadoop file system as well as the invocation of a properly configured Hadoop file system.

3.2 Installation, Start Up, Shut Down, and Reconfiguration

This section contains the instructions for installing HTCondor. The installation will have a default configuration that can be customized. Sections of the manual below explain customization.

Please read this *entire* section before starting installation.

Please read the copyright and disclaimer information in section . Installation and use of HTCondor is acknowledgment that you have read and agree to the terms.

Before installing HTCondor, please consider joining the htcondor-world mailing list. Traffic on this list is kept to an absolute minimum; it is only used to announce new releases of HTCondor. To subscribe, go to https://lists.cs.wisc.edu/mailman/listinfo/htcondor-world, and fill out the online form.

You might also want to consider joining the htcondor-users mailing list. This list is meant to be a forum for HTCondor users to learn from each other and discuss using HTCondor. It is an excellent place to ask the HTCondor community about using and configuring HTCondor. To subscribe, go to https://lists.cs.wisc.edu/mailman/listinfo/htcondor-users, and fill out the online form.

Note that forward and reverse DNS lookup must be enabled for HTCondor to work properly.

3.2.1 Obtaining the HTCondor Software

The first step to installing HTCondor is to download it from the HTCondor web site, http://htcondor.org/. The downloads are available from the downloads page, at http://htcondor.org/downloads/.

3.2.2 Installation on Unix

The HTCondor binary distribution is packaged in the following files and directories:

`LICENSE-2.0.txt` the licensing agreement. By installing HTCondor, you agree to the contents of this file

`README` general information

`bin` directory which contains the distribution HTCondor user programs.

`bosco_install` the Perl script used to install Bosco.

`condor_configure` the Perl script used to install and configure HTCondor.

`condor_install` the Perl script used to install HTCondor.

`etc` directory which contains the distribution HTCondor configuration data.

`examples` directory containing C, Fortran and C++ example programs to run with HTCondor.

`include` directory containing HTCondor header files.

lib directory which contains the distribution HTCondor libraries.

libexec directory which contains the distribution HTCondor auxiliary programs for use internally by HTCondor.

man directory which contains the distribution HTCondor manual pages.

sbin directory containing HTCondor daemon binaries and admin tools.

src directory containing source for some interfaces.

Preparation

Before installation, you need to make a few important decisions about the basic layout of your pool. These decisions answer the following questions:

1. What machine will be the central manager?
2. What machines should be allowed to submit jobs?
3. Will HTCondor run as root or not?
4. Who will be administering HTCondor 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 HTCondor go?
7. Where should the parts of the HTCondor 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 HTCondor?

1. What machine will be the central manager? One machine in your pool must be the central manager. Install HTCondor on this machine first. This is the centralized information repository for the HTCondor 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 HTCondor tools will stop working. Because of the importance of this machine for the proper

functioning of HTCondor, 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, and a pool with about 1000 machines will require approximately 100 Mbytes of memory for the central manager's tasks.

A faster CPU will speed up matchmaking.

Generally jobs should not be either submitted or run on the central manager machine.

2. **Which machines should be allowed to submit jobs?** HTCondor 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 HTCondor pool is behind a firewall, and all machines inside the firewall are trusted, the `ALLOW_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. HTCondor tries to be secure by default: it is shipped with an invalid value that allows no machine to connect and submit jobs.

3. **Will HTCondor run as root or not?** We strongly recommend that the HTCondor daemons be installed and run as the Unix user root. Without this, HTCondor can do very little to enforce security and policy decisions. You can install HTCondor as any user; however there are serious security and performance consequences do doing a non-root installation. Please see section 3.8.13 in the manual for the details and ramifications of installing and running HTCondor as a Unix user other than root.

4. **Who will administer HTCondor?** Either root will be administering HTCondor directly, or someone else will be acting as the HTCondor administrator. If root has delegated the responsibility to another person, keep in mind that as long as HTCondor 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.

   The HTCondor administrator will be regularly updating HTCondor by following these instructions or by using the system-specific installation methods below. The administrator will also customize policies of the HTCondor submit and execute nodes. This person will also receive information from HTCondor if something goes wrong with the pool, as described in the documentation of the `CONDOR_ADMIN` configuration variable.

5. **Will you have a Unix user named condor, and will its home directory be shared?** To simplify installation of HTCondor, you should create a Unix user named condor on all machines in the pool. The HTCondor 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 HTCondor. 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.

   Note that the user condor must not be an account into which a person can log in. If a person can log in as user condor, it permits a major security breach, in that the user condor could submit jobs that run as any other user, providing complete access to the user's data by the jobs. A standard way of not allowing log in to an account on Unix platforms is to enter an invalid shell in the password file.

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 HTCondor files. See section 3.8.13 on UIDs in HTCondor in the Administrator's Manual for details.

6. Where should the machine-specific directories for HTCondor go? HTCondor needs a few directories that are unique on every machine in your pool. These are execute, spool, log, (and possibly lock). Generally, all of them 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 HTCondor is to be run as. Do not stage other files in any of these directories; any files not created by HTCondor in these directories are subject to removal.

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 the directories 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 HTCondor 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. (HTCondor 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 3.10 on Installing a Checkpoint Server on 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. The amount of space will depend on how many executables, and what size they are, that need to be stored in the spool directory.

log Each HTCondor 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 HTCondor 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 HTCondor directories. Then, the only consequence of filling up the directories will be HTCondor's malfunction, not your whole machine.

7. Where should the parts of the HTCondor system be installed?

- Release directory
  - User Binaries
  - System Binaries
  - `lib` Directory
  - `etc` Directory
- Documentation
- Configuration Files

Configuration Files There can be more than one configuration file. They allow different levels of control over how HTCondor is configured on each machine in the 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. Local configuration files override settings in the global file permitting different daemons to run, different policies for when to start and stop HTCondor jobs, and so on. There may be configuration files specific to each platform in the pool. See section 3.14.3 on about Configuring HTCondor for Multiple Platforms for details.

The location of configuration files is described in section 3.3.2.

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 that HTCondor 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 HTCondor 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 HTCondor daemons and agents, or programs that only the HTCondor administrator would need to run. Therefore, add these programs only to the `PATH` of the HTCondor administrator.
- Private HTCondor Binaries:
  All of the files in the `libexec` directory are HTCondor programs that should never be run by hand, but are only used internally by HTCondor.
- `lib` Directory:
  The files in the `lib` directory are the HTCondor libraries that must be linked in with user jobs for all of HTCondor'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 HTCondor 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 HTCondor. `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 HTCondor checks automatically to find its global configuration file.

Documentation The documentation provided with HTCondor 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 HTCondor documentation on the web at: http://htcondor.org/manual.

8. Am I using AFS? If you are using AFS at your site, be sure to read the section 3.14.1 in the manual. HTCondor does not currently have a way to authenticate itself to AFS. A solution is not ready for Version 8.6.10. This implies that you are probably not going to want to have the `LOCAL_DIR` for HTCondor on AFS. However, you can (and probably should) have the HTCondor `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 HTCondor from a directory on AFS. Again, read manual section 3.14.1 for all the details.

9. Do I have enough disk space for HTCondor? The compressed downloads of HTCondor currently range from a low of about 13 Mbytes for 64-bit Ubuntu 12/Linux to about 115 Mbytes for Windows. The compressed source code takes approximately 17 Mbytes.

In addition, you will need a lot of disk space in the local directory of any machines that are submitting jobs to HTCondor. See question 6 above for details on this.

Unix Installation from an RPM

RPMs are available for HTCondor Version 8.6.10. We provide a Yum repository, as well as installation and configuration in one easy step. This RPM installation is currently available for Red Hat-compatible systems only. As of HTCondor version 7.5.1, the HTCondor RPM installs into File Hierachy Standard locations.

Yum repositories and instructions are at http://htcondor.org/yum/ . The repositories are named to distinguish stable releases from development releases and by Red Hat version number. The 4 repositories are:

- `htcondor-stable-rhel6.repo`
- `htcondor-stable-rhel7.repo`
- `htcondor-development-rhel6.repo`
- `htcondor-development-rhel7.repo`

Here is an ordered set of steps that get HTCondor running using the RPM.

1. The HTCondor package will automatically add a `condor` user/group, if it does not exist already. Sites wishing to control the attributes of this user/group should add the `condor` user/group manually before installation.

2. Download and install the meta-data that describes the appropriate YUM repository. This example is for the stable series, on RHEL 7.

```
cd /etc/yum.repos.d
wget http://htcondor.org/yum/repo.d/htcondor-stable-rhel7.repo
```

   Note that this step need be done only once; do not get the same repository more than once.

3. Import signing key The RPMs are signed in the Redhat 6 and RedHat 7 repositories.

```
wget http://htcondor.org/yum/RPM-GPG-KEY-HTCondor
rpm --import RPM-GPG-KEY-HTCondor
```

4. Install HTCondor.

```
yum install condor-all
```

5. As needed, edit the HTCondor configuration files to customize. The configuration files are in the directory `/etc/condor/` . Do *not* use *condor_configure* or *condor_install* for configuration. The installation will be able to find configuration files without additional administrative intervention, as the configuration files are placed in `/etc`, and HTCondor searches this directory.

6. Start HTCondor daemons:

```
/sbin/service condor start
```

Unix Installation from a Debian Package

Debian packages are available in HTCondor Version 8.6.10. We provide an APT repository, as well as installation and configuration in one easy step. These Debian packages of HTCondor are currently available for Debian 7 (wheezy) and Debian 8 (jessie). As of HTCondor version 7.5.1, the HTCondor Debian package installs into File Hierachy Standard locations.

The HTCondor APT repositories are specified at http://htcondor.org/debian/ . See this web page for repository information.

Here is an ordered set of steps that get HTCondor running.

1. The HTCondor package will automatically add a `condor` user/group, if it does not exist already. Sites wishing to control the attributes of this user/group should add the `condor` user/group manually before installation.

2. If not already present, set up access to the appropriate APT repository; they are distinguished as stable or development release, and by operating system. Ensure that the correct one of the following release and operating system-specific lines is in the file `/etc/apt/sources.list` .

```
deb http://htcondor.org/debian/stable/ wheezy contrib
deb http://htcondor.org/debian/development/ wheezy contrib
deb http://htcondor.org/debian/stable/ jessie contrib
deb http://htcondor.org/debian/development/ jessie contrib
```

Note that this step need be done only once; do not add the same repository more than once.

3. Install and start HTCondor services:

```
apt-get update
apt-get install condor
```

4. As needed, edit the HTCondor configuration files to customize. The configuration files are in the directory `/etc/condor/` . Do *not* use *condor_configure* or *condor_install* for configuration. The installation will be able to find configuration files without additional administrative intervention, as the configuration files are placed in `/etc`, and HTCondor searches this directory.

   Then, if any configuration changes are made, restart HTCondor with

```
/etc/init.d/condor restart
```

Unix Installation from a Tarball

Note that installation from a tarball is no longer the preferred method for installing HTCondor on Unix systems. Installation via RPM or Debian package is recommended if available for your Unix version.

An overview of the tarball-based installation process is as follows:

1. Untar the HTCondor software.
2. Run *condor_install* or *condor_configure* to install the software.

Details are given below.

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* (and *bosco_install*), as well as `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src` subdirectories.

The Perl script *condor_configure* installs HTCondor. 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.

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 the named directories. As the names imply, *condor_install* is used to install HTCondor, whereas *condor_configure* is used to modify the configuration of an existing HTCondor install.

condor_configure and *condor_install* are completely command-line driven and are 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 HTCondor release directories. The default command-line argument 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 can take on within the HTCondor 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=submit,execute
```

Install HTCondor on the central manager machine first. If HTCondor 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 HTCondor 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 an 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 HTCondor 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 765 for details.

Starting HTCondor Under Unix After Installation

Now that HTCondor has been installed on the machine(s), there are a few things to check before starting up HTCondor.

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 HTCondor. See the section 3.8.1 to learn how to do this.
2. For Linux platforms, run the *condor_kbdd* to monitor keyboard and mouse activity on all machines within the pool that will run a *condor_startd*; these are machines that execute jobs. To do this, the subsystem `KBDD` will need to be added to the `DAEMON_LIST` configuration variable definition.

   For Unix platforms other than Linux, HTCondor 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 is 'mouse', since most 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 HTCondor'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) HTCondor needs to be able to find the `utmp` file. According to the Linux File System Standard, this file should be `/var/run/utmp`. If HTCondor 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 HTCondor daemons, execute the command `<release_dir>/sbin/condor_master`. This is the HTCondor master, whose only job in life is to make sure the other HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor libraries to take advantage of all of HTCondor'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 3.14.4 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 3.14.1 in the manual.
4. We strongly recommend that you start up HTCondor (run the *condor_master* daemon) as user root. If you must start HTCondor as some user other than root, see section 3.8.13.

3.2.3 Installation on Windows

This section contains the instructions for installing the Windows version of HTCondor. The install program will set up a slightly customized configuration file that can be further customized after the installation has completed.

Be sure that the HTCondor tools are of the same version as the daemons installed. The HTCondor executable for distribution is packaged in a single file named similarly to:

```
condor-8.4.11-390598-Windows-x86.msi
```

This file is approximately 107 Mbytes in size, and it can be removed once HTCondor is fully installed.

For any installation, HTCondor services are installed and run as the Local System account. Running the HTCondor services as any other account (such as a domain user) is not supported and could be problematic.

Installation Requirements

- HTCondor for Windows is supported for Windows Vista or a more recent version.
- 300 megabytes of free disk space is recommended. Significantly more disk space could be necessary to be able to run jobs with large data files.
- HTCondor for Windows will operate on either an NTFS or FAT32 file system. However, for security purposes, NTFS is preferred.
- HTCondor for Windows uses the Visual C++ 2012 C runtime library.

Preparing to Install HTCondor under Windows

Before installing the Windows version of HTCondor, there are two major decisions to make about the basic layout of the pool.

1. What machine will be the central manager?
2. Is there enough disk space for HTCondor?

If the answers to these questions are already known, skip to the Windows Installation Procedure section below, section 3.2.3. If 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 HTCondor 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 HTCondor tools will stop working. Because of the importance of this machine for the proper functioning of HTCondor, we recommend installing 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 network layout when choosing the central manager.

  Install HTCondor on the central manager before installing on the other machines within the pool.

  Generally jobs should not be either submitted or run on the central manager machine.

- Is there enough disk space for HTCondor?

  The HTCondor release directory takes up a fair amount of space. The size requirement for the release directory is approximately 250 Mbytes. HTCondor itself, however, needs space to store all of the jobs and their input files. If there will be large numbers of jobs, consider installing HTCondor on a volume with a large amount of free space.

Installation Procedure Using the MSI Program

Installation of HTCondor must be done by a user with administrator privileges. After installation, the HTCondor services will be run under the local system account. When HTCondor is running a user job, however, it will run that user job with normal user permissions.

Download HTCondor, and start the installation process by running the installer. The HTCondor installation is completed by answering questions and choosing options within the following steps.

If HTCondor is already installed. If HTCondor has been previously installed, a dialog box will appear before the installation of HTCondor proceeds. The question asks if you wish to preserve your current HTCondor 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 HTCondor 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.

Also fill in name and company information, or use the defaults as given.

STEP 2: HTCondor Pool Configuration. The HTCondor configuration needs to be set based upon if this is a new pool or to join an existing one. Choose the appropriate radio button.

For a new pool, enter a chosen name for the pool. To join an existing pool, enter the host name of the central manager of the pool.

STEP 3: This Machine's Roles. Each machine within an HTCondor pool can either submit jobs or execute submitted jobs, or both submit and execute jobs. A check box determines if this machine will be a submit point for the pool.

A set of radio buttons determines the ability and configuration of the ability to execute jobs. There are four choices:

Do not run jobs on this machine. This machine will not execute HTCondor jobs.

Always run jobs and never suspend them.

Run jobs when the keyboard has been idle for 15 minutes.

Run jobs when the keyboard has been idle for 15 minutes, and the CPU is idle.

For testing purposes, it is often helpful to use the always run HTCondor jobs option.

For a machine that is to execute jobs and the choice is one of the last two in the list, HTCondor needs to further know what to do with the currently running jobs. There are two choices:

Keep the job in memory and continue when the machine meets the condition chosen for when to run jobs.

Restart the job on a different machine.

This choice involves a trade off. Restarting the job on a different machine is less intrusive on the workstation owner than leaving the job in memory for a later time. A suspended job left in memory will require swap space, which could 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 4: The Account Domain. Enter the machine's accounting (or UID) domain. On this version of HTCondor for Windows, this setting is only used for user priorities (see section 3.6) and to form a default e-mail address for the user.

STEP 5: E-mail Settings. Various parts of HTCondor will send e-mail to an HTCondor administrator if something goes wrong and requires human attention. Specify the e-mail address and the SMTP relay host of this administrator. Please pay close attention to this e-mail, since it will indicate problems in the HTCondor pool.

STEP 6: Java Settings. In order to run jobs in the **java** universe, HTCondor must have the path to the jvm executable on the machine. The installer will search for and list the jvm path, if it finds one. If not, enter the path. To disable use of the **java** universe, leave the field blank.

STEP 7: Host Permission Settings. Machines within the HTCondor pool will need various types of access permission. The three categories of permission are read, write, and administrator. Enter the machines or domain to be given access permissions, or use the defaults provided. Wild cards and macros are permitted.

Read Read access allows a machine to obtain information about HTCondor 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 HTCondor team to obtain information about the HTCondor 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 HTCondor daemons, for example, to start an HTCondor job. Note that for a machine to join the HTCondor 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 do things such as change other user's priorities, modify the job queue, turn HTCondor services on and off, and restart HTCondor. 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 configuration file, please see the section titled Setting Up IP/Host-Based Security in HTCondor in section section 3.8.9.

STEP 8: VM Universe Setting. A radio button determines whether this machine will be configured to run **vm** universe jobs utilizing VMware. In addition to having the VMware Server installed, HTCondor also needs *Perl* installed. The resources available for **vm** universe jobs can be tuned with these settings, or the defaults listed can be used.

Version Use the default value, as only one version is currently supported.

Maximum Memory The maximum memory that each virtual machine is permitted to use on the target machine.

Maximum Number of VMs The number of virtual machines that can be run in parallel on the target machine.

Networking Support The VMware instances can be configured to use network support. There are four options in the pull-down menu.

- None: No networking support.

- NAT: Network address translation.
- Bridged: Bridged mode.
- NAT and Bridged: Allow both methods.

Path to Perl Executable The path to the *Perl* executable.

STEP 9: HDFS Settings. A radio button enables support for the Hadoop Distributed File System (HDFS). When enabled, a further radio button specifies either name node or data node mode.

Running HDFS requires Java to be installed, and HTCondor must know where the installation is. Running HDFS in data node mode also requires the installation of Cygwin, and the path to the Cygwin directory must be added to the global `PATH` environment variable.

HDFS has several configuration options that must be filled in to be used.

Primary Name Node The full host name of the primary name node.

Name Node Port The port that the name node is listening on.

Name Node Web Port The port the name node's web interface is bound to. It should be different from the name node's main port.

STEP 10: Choose Setup Type The next step is where the destination of the HTCondor files will be decided. We recommend that HTCondor be installed in the location shown as the default in the install choice: `C:\Condor`. This is due to several hard coded paths in scripts and configuration files. Clicking on the Custom choice permits changing the installation directory.

Installation on the local disk is chosen for several reasons. The HTCondor services run as local system, and within Microsoft Windows, local system has no network privileges. Therefore, for HTCondor to operate, HTCondor 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 HTCondor 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 HTCondor on a hard drive that is not local, if a dependency is added to the service control manager such that HTCondor starts after the required file services are available.

Unattended Installation Procedure Using the Included Setup Program

This section details how to run the HTCondor for Windows installer in an unattended batch mode. This mode is one that occurs completely from the command prompt, without the GUI interface.

The HTCondor 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=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="$(IP_ADDRESS)"
set ARGS=%ARGS% INSTALLDIR="C:\Condor"
set ARGS=%ARGS% POOLHOSTNAME="$(IP_ADDRESS)"
set ARGS=%ARGS% ACCOUNTINGDOMAIN="none"
set ARGS=%ARGS% JVMLOCATION="C:\Windows\system32\java.exe"
set ARGS=%ARGS% USEVMUNIVERSE="N"
set ARGS=%ARGS% VMMEMORY="128"
set ARGS=%ARGS% VMMAXNUMBER="$(NUM_CPUS)"
set ARGS=%ARGS% VMNETWORKING="N"
REM set ARGS=%ARGS% LOCALCONFIG="http://my.example.com/condor_config.$(FULL_HOSTNAME)"

msiexec /qb /l* condor-install-log.txt /i condor-8.0.0-133173-Windows-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 HTCondor 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 HTCondor should do when it has to stop the execution of a user job. When set to Y, HTCondor will vacate the job and start it somewhere else if possible. When set to N, HTCondor 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 HTCondor 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 HTCondor daemons. This value should be set in accordance with the `HOSTALLOW_READ` setting in the configuration file, as described in section 3.8.9.

HOSTALLOWWRITE is a list of host names that are allowed to issue WRITE commands to HTCondor daemons. This value should be set in accordance with the `HOSTALLOW_WRITE` setting in the configuration file, as described in section 3.8.9.

HOSTALLOWADMINISTRATOR is a list of host names that are allowed to issue ADMINISTRATOR commands to HTCondor daemons. This value should be set in accordance with the `HOSTALLOW_ADMINISTRATOR` setting in the configuration file, as described in section 3.8.9.

INSTALLDIR defines the path to the directory where HTCondor will be installed.

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.

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.

LOCALCONFIG defines the location of the local configuration file. The value can be the path to a file on the local machine, or it can be a URL beginning with `http`. If the value is a URL, then the *condor_urlfetch* tool is invoked to fetch configuration whenever the configuration is read.

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 /lxv* condor-install-log.txt /i condor-8.0.0-173133-Windows-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.mspx.

Manual Installation HTCondor on Windows

If you are to install HTCondor on many different machines, you may wish to use some other mechanism to install HTCondor 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 HTCondor manually without using the provided GUI-based setup program:

The Service The service that HTCondor 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 HTCondor 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 HTCondor uses a few registry entries in its operation. The key that HTCondor uses is HKEY_LOCAL_MACHINE/Software/Condor. The values that HTCondor puts in this registry key serve two purposes.

1. The values of CONDOR_CONFIG and RELEASE_DIR are used for HTCondor to start its service.
   CONDOR_CONFIG should point to the `condor_config` file. In this version of HTCondor, it **must** reside on the local disk.
   RELEASE_DIR should point to the directory where HTCondor 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 HTCondor to operate are identical to the Unix version of HTCondor, except that executable files end in `.exe`. For example the on Unix one of the files is `condor_master` and on HTCondor 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 HTCondor will expect to be present given your configuration. This normally is simply a matter of creating the `log`, `spool`, and `execute` directories. Do not stage other files in any of these directories; any files not created by HTCondor in these directories are subject to removal.

Starting HTCondor Under Windows After Installation

After the installation of HTCondor is completed, the HTCondor service must be started. If you used the GUI-based setup program to install HTCondor, the HTCondor service should already be started. If you installed manually, HTCondor must be started by hand, or you can simply reboot. NOTE: The HTCondor service will start automatically whenever you reboot your machine.

To start HTCondor 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 HTCondor 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 HTCondor node should start jobs
- *condor_schedd.exe*, if you indicated that this HTCondor node should submit jobs to the HTCondor pool.

Also, you should now be able to open up a new cmd (DOS prompt) window, and the HTCondor bin directory should be in your path, so you can issue the normal HTCondor commands, such as *condor_q* and *condor_status*.

HTCondor is Running Under Windows ... Now What?

Once HTCondor services are running, try submitting test jobs. Example 2 within section 2.5.1 presents a vanilla universe job.

3.2.4 Upgrading – Installing a New Version on an Existing Pool

An upgrade changes the running version of HTCondor from the current installation to a newer version. The safe method to install and start running a newer version of HTCondor in essence is: shut down the current installation of HTCondor, install the newer version, and then restart HTCondor 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, as well as incorporate any configuration variables that are new or changed in the new version. Set the `CONDOR_CONFIG` environment variable to point to the new copy of the configuration, so the new version of HTCondor will use the new configuration when restarted.

As of HTCondor version 8.2.0, the default configuration file has been substantially reduced in size by defining compile-time default values for most configuration variables. Therefore, when upgrading from a version of HTCondor earlier than 8.2.0 to a more recent version, the option of reducing the size of the configuration file is an option. The goal is to identify and use only the configuration variable values that differ from the compile-time default values. This is facilitated by using *condor_config_val* with the **-writeconfig:upgrade** *a*rgument, to create a file that behaves the same as the current configuration, but is much smaller, because values matching the default values (as well as some obsolete variables) have been removed. Items in the file created by running *condor_config_val* with the **-writeconfig:upgrade** *a*rgument will be in the order that they were read from the original configuration files. This file is a convenient guide to stripping the cruft from old configuration files.

When upgrading from a version of HTCondor earlier than 6.8 to more recent version, note that the configuration settings must be modified for security reasons. Specifically, the `HOSTALLOW_WRITE` configuration variable must be explicitly changed, or no jobs can be submitted, and error messages will be issued by HTCondor tools.

Another way to upgrade leaves HTCondor running. HTCondor will automatically restart itself if the *condor_master* binary is updated, and this method takes advantage of this. Download the newer version, placing it such that it does not overwrite the currently running version. With the download will be a new set of configuration files; update this new set with any specializations implemented in the currently running version of HTCondor. Then, modify the currently running installation by changing its configuration such that the path to binaries points instead to the new binaries. One way to do that (under Unix) is to use a symbolic link that points to the current HTCondor installation directory (for example, `/opt/condor`). Change the symbolic link to point to the new directory. If HTCondor is configured to locate its binaries via the symbolic link, then after the symbolic link changes, the *condor_master* daemon notices the new binaries and restarts itself. How frequently it checks is controlled by the configuration variable `MASTER_CHECK_NEW_EXEC_INTERVAL`, which defaults 5 minutes.

When the *condor_master* notices new binaries, it begins a graceful restart. On an execute machine, a graceful restart means that running jobs are preempted. Standard universe jobs will attempt to take a checkpoint. This could be a bottleneck if all machines in a large pool attempt to do this at the same time. If they do not complete within the cutoff time specified by the `KILL` policy expression (defaults to 10 minutes), then the jobs are killed without producing a checkpoint. It may be appropriate to increase this cutoff time, and a better approach may be to upgrade the pool in stages rather than all at once.

For universes other than the standard universe, jobs are preempted. If jobs have been guaranteed a certain amount of uninterrupted run time with `MaxJobRetirementTime`, then the job is not killed until the specified amount of retirement time has been exceeded (which is 0 by default). The first step of killing the job is a soft kill signal, which can be intercepted by the job so that it can exit gracefully, perhaps saving its state. If the job has not gone away once the `KILL` expression fires (10 minutes by default), then the job is forcibly hard-killed. Since the graceful shutdown of jobs

may rely on shared resources such as disks where state is saved, the same reasoning applies as for the standard universe: it may be appropriate to increase the cutoff time for large pools, and a better approach may be to upgrade the pool in stages to avoid jobs running out of time.

Another time limit to be aware of is the configuration variable `SHUTDOWN_GRACEFUL_TIMEOUT`. This defaults to 30 minutes. If the graceful restart is not completed within this time, a fast restart ensues. This causes jobs to be hard-killed.

3.2.5 Shutting Down and Restarting an HTCondor Pool

All of the commands described in this section are subject to the security policy chosen for the HTCondor pool. As such, the commands must be either run from a machine that has the proper authorization, or run by a user that is authorized to issue the commands. Section 3.8 details the implementation of security in HTCondor.

Shutting Down HTCondor There are a variety of ways to shut down all or parts of an HTCondor pool. All utilize the *condor_off* tool.

To stop a single execute machine from running jobs, the *condor_off* command specifies the machine by host name.

```
condor_off -startd <hostname>
```

A running **standard** universe job will be allowed to take a checkpoint before the job is killed. A running job under another universe will be killed. If it is instead desired that the machine stops running jobs only after the currently executing job completes, the command is

```
condor_off -startd -peaceful <hostname>
```

Note that this waits indefinitely for the running job to finish, before the *condor_startd* daemon exits.

Th shut down all execution machines within the pool,

```
condor_off -all -startd
```

To wait indefinitely for each machine in the pool to finish its current HTCondor job, shutting down all of the execute machines as they no longer have a running job,

```
condor_off -all -startd -peaceful
```

To shut down HTCondor on a machine from which jobs are submitted,

```
condor_off -schedd <hostname>
```

If it is instead desired that the submit machine shuts down only after all jobs that are currently in the queue are finished, first disable new submissions to the queue by setting the configuration variable

```
MAX_JOBS_SUBMITTED = 0
```

See instructions below in section 3.2.6 for how to reconfigure a pool. After the reconfiguration, the command to wait for all jobs to complete and shut down the submission of jobs is

```
condor_off -schedd -peaceful <hostname>
```

Substitute the option **-all** for the host name, if all submit machines in the pool are to be shut down.

Restarting HTCondor, If HTCondor Daemons Are Not Running If HTCondor is not running, perhaps because one of the *condor_off* commands was used, then starting HTCondor daemons back up depends on which part of HTCondor is currently not running.

If no HTCondor daemons are running, then starting HTCondor is a matter of executing the *condor_master* daemon. The *condor_master* daemon will then invoke all other specified daemons on that machine. The *condor_master* daemon executes on every machine that is to run HTCondor.

If a specific daemon needs to be started up, and the *condor_master* daemon is already running, then issue the command on the specific machine with

```
condor_on -subsystem <subsystemname>
```

where `<subsystemname>` is replaced by the daemon's subsystem name. Or, this command might be issued from another machine in the pool (which has administrative authority) with

```
condor_on <hostname> -subsystem <subsystemname>
```

where `<subsystemname>` is replaced by the daemon's subsystem name, and `<hostname>` is replaced by the host name of the machine where this *condor_on* command is to be directed.

Restarting HTCondor, If HTCondor Daemons Are Running If HTCondor daemons are currently running, but need to be killed and newly invoked, the *condor_restart* tool does this. This would be the case for a new value of a configuration variable for which using *condor_reconfig* is inadequate.

To restart all daemons on all machines in the pool,

```
condor_restart -all
```

To restart all daemons on a single machine in the pool,

```
condor_restart <hostname>
```

where `<hostname>` is replaced by the host name of the machine to be restarted.

3.2.6 Reconfiguring an HTCondor Pool

To change a global configuration variable and have all the machines start to use the new setting, change the value within the file, and send a *condor_reconfig* command to each host. Do this with a *single* command,

```
condor_reconfig -all
```

If the global configuration file is not shared among all the machines, as it will be if using a shared file system, the change must be made to each copy of the global configuration file before issuing the *condor_reconfig* command.

Issuing a *condor_reconfig* command is inadequate for some configuration variables. For those, a restart of HTCondor is required. Those configuration variables that require a restart are listed in section 3.3.11. The manual page for *condor_restart* is at 858.

3.3 Introduction to Configuration

This section of the manual contains general information about HTCondor configuration, relating to all parts of the HTCondor system. If you're setting up an HTCondor pool, you should read this section before you read the other configuration-related sections:

- Section 3.4 contains information about configuration templates, which are now the preferred way to set many configuration macros.
- Section 3.5 contains information about the hundreds of individual configuration macros. In general, it is best to try to achieve your desired configuration using configuration templates before resorting to setting individual configuration macros, but it is sometimes necessary to set individual configuration macros.
- The settings that control the policy under which HTCondor will start, suspend, resume, vacate or kill jobs are described in section 3.7 on Policy Configuration for the *condor_startd*.

3.3.1 HTCondor Configuration Files

The HTCondor configuration files are used to customize how HTCondor operates at a given site. The basic configuration as shipped with HTCondor can be used as a starting point, but most likely you will want to modify that configuration to some extent.

Each HTCondor 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). Some configuration variables are evaluated by HTCondor as ClassAd expressions; some are not. Consult the documentation for each specific case. Unless otherwise noted, configuration values that are expected to be numeric or boolean constants can be any valid ClassAd expression of operators on constants. Example:

```
MINUTE           = 60
HOUR             = (60 * $(MINUTE))
SHUTDOWN_GRACEFUL_TIMEOUT = ($(HOUR)*24)
```

3.3.2 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 earlier 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 HTCondor user from subverting the system through its configuration. The order in which items are parsed is:

1. a single initial configuration file, which has historically been known as the global configuration file (see below);
2. other configuration files that are referenced and parsed due to specification within the single initial configuration file (these files have historically been known as local configuration files);
3. if HTCondor daemons are *not* running as `root` on Unix platforms, the file `$(HOME)/.condor/user_config` if it exists, or the file defined by configuration variable `USER_CONFIG_FILE`;

   if HTCondor daemons are *not* running as Local System on Windows platforms, the file `%USERPROFILE\.condor\user_config` if it exists, or the file defined by configuration variable `USER_CONFIG_FILE`;
4. specific environment variables whose names are prefixed with `_CONDOR_` (note that these environment variables directly define macro name/value pairs, not the names of configuration files).

Some HTCondor tools utilize environment variables to set their configuration; these tools search for specifically-named environment variables. The variable names 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.

The location of the single initial configuration file differs on Windows from Unix platforms. For Unix platforms, the location of the single initial configuration file starts at the top of the following list. The first file that exists is used, and then remaining possible file locations from this list become irrelevant.

1. the file specified by the `CONDOR_CONFIG` environment variable. If there is a problem reading that file, HTCondor will print an error message and exit right away.
2. `/etc/condor/condor_config`
3. `/usr/local/etc/condor_config`
4. `~condor/condor_config`

For Windows platforms, the location of the single initial configuration file is determined by the contents of the environment variable `CONDOR_CONFIG`. If this environment variable is not defined, then the location is the registry value of `HKEY_LOCAL_MACHINE/Software/Condor/CONDOR_CONFIG`.

The single, initial configuration file may contain the specification of one or more other configuration files, referred to here as local configuration files. Since more than one file may contain a definition of the same variable, and since the last definition of a variable sets the value, the parse order of these local configuration files is fully specified here. In order:

1. The value of configuration variable LOCAL_CONFIG_DIR lists one or more directories which contain configuration files. The list is parsed from left to right. The leftmost (first) in the list is parsed first. Within each directory, a lexicographical ordering by file name determines the ordering of file consideration.
2. The value of configuration variable LOCAL_CONFIG_FILE lists one or more configuration files. These listed files are parsed from left to right. The leftmost (first) in the list is parsed first.
3. If one of these steps changes the value (right hand side) of LOCAL_CONFIG_DIR, then LOCAL_CONFIG_DIR is processed for a second time, using the changed list of directories.

The parsing and use of configuration files may be bypassed by setting environment variable CONDOR_CONFIG with the string ONLY_ENV. With this setting, there is no attempt to locate or read configuration files. This may be useful for testing where the environment contains all needed information.

3.3.3 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 insensitive identifier. There may 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[:<default if macro_name not defined>])
```

The colon and default are optional in a macro invocation. 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 HTCondor refuses to parse.

A macro invocation where the macro name is not defined results in a substitution of the empty string. Consider the example

```
MAX_ALLOC_CPUS = $(NUMCPUS)-1
```

If `NUMCPUS` is not defined, then this macro substitution becomes

```
MAX_ALLOC_CPUS = -1
```

The default value may help to avoid this situation. The default value may be a literal

```
MAX_ALLOC_CPUS = $(NUMCPUS:4)-1
```

such that if `NUMCPUS` is not defined, the result of macro substitution becomes

```
MAX_ALLOC_CPUS = 4-1
```

The default may be another macro invocation:

```
MAX_ALLOC_CPUS = $(NUMCPUS:$(DETECTED_CPUS))-1
```

These default specifications are restricted such that a macro invocation with a default can not be nested inside of another default. An alternative way of stating this restriction is that there can only be one colon character per line. The effect of nested defaults can be achieved by placing the macro definitions on separate lines of the configuration.

All entries in a configuration file must have an operator, which will be an equals sign (=). Identifiers are alphanumerics 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.

```
[HTCondor 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 3.3.12 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 HTCondor may use can be restricted to a range using the `HIGHPORT` and `LOWPORT` configuration variables.

```
MASTER.LOWPORT   = 20000
MASTER.HIGHPORT  = 20100
```

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 HTCondor 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.

As of HTCondor version 8.1.1, evaluation works in the expected manner when combining the definition of a macro with use of a prefix that gives the subsystem name and a period. Consider the example

```
FILESPEC = A
MASTER.FILESPEC = B
```

combined with a later definition that incorporates `FILESPEC` in a macro:

```
USEFILE = mydir/$(FILESPEC)
```

When the *condor_master* evaluates variable USEFILE, it evaluates to mydir/B. Previous to HTCondor version 8.1.1, it evaluated to mydir/A. When any other subsystem evaluates variable USEFILE, it evaluates to mydir/A.

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** *xyzzy* on the command line.

Continuing the example, configure the *condor_schedd* daemon named xyzzy. 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 HTCondor 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.

3.3.4 Comments and Line Continuations

An HTCondor 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_ADMINISTRATOR = $(ADMIN_MACHINES)
```

Where a line continuation character directly precedes a comment, the entire comment line is ignored, and the following line is used in the continuation. Line continuation characters within comments are ignored.

Both this example

```
A = $(B) \
# $(C)
$(D)
```

and this example

```
A = $(B) \
# $(C) \
$(D)
```

result in the same value for A:

```
A = $(B) $(D)
```

3.3.5 Multi-Line Values

As of version 8.5.6, the value for a macro can comprise multiple lines of text. The syntax for this is as follows:

```
<macro_name> @=<tag>
<macro_definition lines>
@<tag>
```

For example:

```
JOB_ROUTER_DEFAULTS @=jrd
  [
    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 && (time() - QDate) > 3600*6);
    delete_WantJobRouter = true;
    set_requirements = true;
  ]
  @jrd
```

Note that in this example, the square brackets are part of the JOB_ROUTER_DEFAULTS value.

3.3.6 Executing a Program to Produce Configuration Macros

Instead of reading from a file, HTCondor can 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 HTCondor 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.

(Note that the `include command` syntax (see below) is now the preferred way to execute a program to generate configuration macros.)

3.3.7 Including Configuration from Elsewhere

Externally defined configuration can be incorporated using the following syntax:

```
include [ifexist] : <file>
include : <cmdline>|
include [ifexist] command [into <cache-file>] : <cmdline>
```

(Note that the `ifexist` and `into` options were added in version 8.5.7. Also note that the `command` option must be specified in order to use the `into` option – just using the bar after `<cmdline>` will not work.)

In the file form of the `include` command, the `<file>` specification must describe a single file, the contents of which will be parsed and incorporated into the configuration. Unless the `ifexist` option is specified, the non-existence of the file is a fatal error.

In the command line form of the `include` command (specified with either the `command` option or by appending a bar (|) character after the `<cmdline>` specification), the `<cmdline>` specification must describe a command line (program and arguments); the command line will be executed, and the output will be parsed and incorporated into the configuration.

If the `into` option is not used, the command line will be executed every time the configuration file is referenced. This may well be undesirable, and can be avoided by using the `into` option. The `into` keyword must be followed by the full pathname of a file into which to write the output of the command line. If that file exists, it will be read and the command line will not be executed. If that file does not exist, the output of the command line will be written into it and then the cache file will be read and incorporated into the configuration. If the command line produces no output, a zero length file will be created. If the command line returns a non-zero exit code, configuration will abort and the cache file will not be created unless the `ifexist` keyword is also specified.

The `include` key word is case insensitive. There are *no* requirements for white space characters surrounding the colon character.

Consider the example

```
FILE = config.$(FULL_HOSTNAME)
include : $(LOCAL_DIR)/$(FILE)
```

Values are acquired for configuration variables `FILE`, and `LOCAL_DIR` by immediate evaluation, causing variable `FULL_HOSTNAME` to also be immediately evaluated. The resulting value forms a full path and file name. This file is read and parsed. The resulting configuration is incorporated into the current configuration. This resulting configuration

may contain further nested `include` specifications, which are also parsed, evaluated, and incorporated. Levels of nested `include`s are limited, such that infinite nesting is discovered and thwarted, while still permitting nesting.

Consider the further example

```
SCRIPT_FILE = script.$(IP_ADDRESS)
include : $(RELEASE_DIR)/$(SCRIPT_FILE) |
```

In this example, the bar character at the end of the line causes a script to be invoked, and the output of the script is incorporated into the current configuration. The same immediate parsing and evaluation occurs in this case as when a file's contents are included.

For pools that are transitioning to using this new syntax in configuration, while still having some tools and daemons with HTCondor versions earlier than 8.1.6, special syntax in the configuration will cause those daemons to fail upon startup, rather than continuing, but incorrectly parsing the new syntax. Newer daemons will ignore the extra syntax. Placing the `@` character before the `include` key word causes the older daemons to fail when they attempt to parse this syntax.

Here is the same example, but with the syntax that causes older daemons to fail when reading it.

```
FILE = config.$(FULL_HOSTNAME)
@include : $(LOCAL_DIR)/$(FILE)
```

A daemon older than version 8.1.6 will fail to start. Running an older *condor_config_val* identifies the `@include` line as being bad. A daemon of HTCondor version 8.1.6 or more recent sees:

```
FILE = config.$(FULL_HOSTNAME)
include : $(LOCAL_DIR)/$(FILE)
```

and starts up successfully.

Here is an example using the new `ifexist` and `into` options:

```
# stuff.pl writes "STUFF=1" to stdout
include ifexist command into $(LOCAL_DIR)/stuff.config : perl $(LOCAL_DIR)/stuff.pl
```

3.3.8 Reporting Errors and Warnings

As of version 8.5.7, warning and error messages can be included in HTCondor configuration files.

The syntax for warning and error messages is as follows:

```
warning : <warning message>
error : <error message>
```

The warning and error messages will be printed when the configuration file is used (when almost any HTCondor command is run, for example). Error messages (unlike warnings) will prevent the successful use of the configuration file. This will, for example, prevent a daemon from starting, and prevent *condor_config_val* from returning a value.

Here's an example of using an error message in a configuration file (combined with some of the new include features documented above):

```
# stuff.pl writes "STUFF=1" to stdout
include command into $(LOCAL_DIR)/stuff.config : perl $(LOCAL_DIR)/stuff.pl
if ! defined stuff
  error : stuff is needed!
endif
```

3.3.9 Conditionals in Configuration

Conditional `if`/`else` semantics are available in a limited form. The syntax:

```
if <simple condition>
   <statement>
   . . .
   <statement>
else
   <statement>
   . . .
   <statement>
endif
```

An `else` key word and statements are not required, such that simple `if` semantics are implemented. The `<simple condition>` does not permit compound conditions. It optionally contains the exclamation point character (`!`) to represent the not operation, followed by

- the `defined` keyword followed by the name of a variable. If the variable is defined, the statement(s) are incorporated into the expanded input. If the variable is *not* defined, the statement(s) are not incorporated into the expanded input. As an example,

  ```
  if defined MY_UNDEFINED_VARIABLE
     X = 12
  else
     X = -1
  endif
  ```

  results in `X = -1`, when `MY_UNDEFINED_VARIABLE` is *not* yet defined.

- the `version` keyword, representing the version number of of the daemon or tool currently reading this conditional. This keyword is followed by an HTCondor version number. That version number can be of the form

`x.y.z` or `x.y`. The version of the daemon or tool is compared to the specified version number. The comparison operators are

- == for equality. Current version 8.2.3 is equal to 8.2.
- >= to see if the current version number is greater than or equal to. Current version 8.2.3 is greater than 8.2.2, and current version 8.2.3 is greater than or equal to 8.2.
- <= to see if the current version number is less than or equal to. Current version 8.2.0 is less than 8.2.2, and current version 8.2.3 is less than or equal to 8.2.

As an example,

```
if version >= 8.1.6
   DO_X = True
else
   DO_Y = True
endif
```

results in defining `DO_X` as `True` if the current version of the daemon or tool reading this if statement is 8.1.6 or a more recent version.

- `True` or `yes` or the value 1. The statement(s) are incorporated.
- `False` or `no` or the value 0 The statement(s) are *not* incorporated.
- `$(<variable>)` may be used where the immediately evaluated value is a simple boolean value. A value that evaluates to the empty string is considered `False`, otherwise a value that does not evaluate to a simple boolean value is a syntax error.

The syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
elif <simple condition>
   <statement>
   . . .
   <statement>
endif
```

is the same as syntax

```
if <simple condition>
   <statement>
   . . .
   <statement>
```

```
else
   if <simple condition>
      <statement>
      . . .
      <statement>
   endif
endif
```

3.3.10 Function Macros in Configuration

A set of predefined functions increase flexibility. Both submit description files and configuration files are read using the same parser, so these functions may be used in both submit description files and configuration files.

Case is significant in the function's name, so use the same letter case as given in these definitions.

$CHOICE(index, listname) or $CHOICE(index, item1, item2, ...) An item within the list is returned. The list is represented by a parameter name, or the list items are the parameters. The `index` parameter determines which item. The first item in the list is at index 0. If the index is out of bounds for the list contents, an error occurs.

$ENV(environment-variable-name[:default-value]) Evaluates to the value of environment variable `environment-variable-name`. If there is no environment variable with that name, Evaluates to UNDEFINED unless the optional `:default-value` is used; in which case it evaluates to default-value. For example,

```
A = $ENV(HOME)
```

binds `A` to the value of the `HOME` environment variable.

$F[fpduwnxbqa](filename) One or more of the lower case letters may be combined to form the function name and thus, its functionality. Each letter operates on the `filename` in its own way.

- `f` convert relative path to full path by prefixing the current working directory to it. This option works only in *condor_submit* files.
- `p` refers to the entire directory portion of `filename`, with a trailing slash or backslash character. Whether a slash or backslash is used depends on the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified.
- `d` refers to the last portion of the directory within the path, if specified. It will have a trailing slash or backslash, as appropriate to the platform of the machine. The slash will be recognized on Linux platforms; either a slash or backslash will be recognized on Windows platforms, and the parser will use the same character specified unless u or w is used. if b is used the trailing slash or backslash will be omitted.
- `u` convert path separators to Unix style slash characters
- `w` convert path separators to Windows style backslash characters

- `n` refers to the file name at the end of any path, but without any file name extension. As an example, the return value from `$Fn(/tmp/simulate.exe)` will be `simulate` (without the `.exe` extension).
- `x` refers to a file name extension, with the associated period (`.`). As an example, the return value from `$Fn(/tmp/simulate.exe)` will be `.exe`.
- `b` when combined with the d option, causes the trailing slash or backslash to be omitted. When combined with the x option, causes the leading period (`.`) to be omitted.
- `q` causes the return value to be enclosed within quotes. Double quote marks are used unless a is also specified.
- `a` When combined with the q option, causes the return value to be enclosed within single quotes.

`$DIRNAME(filename)` is the same as `$Fp(filename)`

`$BASENAME(filename)` is the same as `$Fnx(filename)`

`$INT(item-to-convert)` or `$INT(item-to-convert, format-specifier)` Expands, evaluates, and returns a string version of `item-to-convert`. The `format-specifier` has the same syntax as a C language or Perl format specifier. If no `format-specifier` is specified, `"%d"` is used as the format specifier.

`$RANDOM_CHOICE(choice1, choice2, choice3, ...)` A random choice of one of the parameters in the list of parameters is made. For example, if one of the integers 0-8 (inclusive) should be randomly chosen:

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

`$RANDOM_INTEGER(min, max [, step])` A random integer within the range `min` and `max`, inclusive, is selected. 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):

```
$RANDOM_INTEGER(0, 8, 2)
```

`$REAL(item-to-convert)` or `$REAL(item-to-convert, format-specifier)` Expands, evaluates, and returns a string version of `item-to-convert` for a floating point type. The `format-specifier` is a C language or Perl format specifier. If no `format-specifier` is specified, `"%16G"` is used as a format specifier.

`$SUBSTR(name, start-index)` or `$SUBSTR(name, start-index, length)` Expands `name` and returns a substring of it. The first character of the string is at index 0. The first character of the substring is at index `start-index`. If the optional `length` is not specified, then the substring includes characters up to the end of the string. A negative value of `start-index` works back from the end of the string. A negative value of `length` eliminates use of characters from the end of the string. Here are some examples that all assume

```
Name = abcdef
```

- `$SUBSTR(Name, 2)` is `cdef`.
- `$SUBSTR(Name, 0, -2)` is `abcd`.
- `$SUBSTR(Name, 1, 3)` is `bcd`.
- `$SUBSTR(Name, -1)` is `f`.

- `$SUBSTR(Name, 4, -3)` is the empty string, as there are no characters in the substring for this request.

Environment references are not currently used in standard HTCondor configurations. However, they can sometimes be useful in custom configurations.

3.3.11 Macros That Will Require a Restart When Changed

When any of the following listed configuration variables are changed, HTCondor must be restarted. Reconfiguration using *condor_reconfig* will not be enough.

- `BIND_ALL_INTERFACES`
- `FetchWorkDelay`
- `MAX_NUM_CPUS`
- `MAX_TRACKING_GID`
- `MEMORY`
- `MIN_TRACKING_GID`
- `NETWORK_HOSTNAME`
- `NETWORK_INTERFACE`
- `NUM_CPUS`
- `PREEMPTION_REQUIREMENTS_STABLE`
- `PRIVSEP_ENABLED`
- `PROCD_ADDRESS`
- `SLOT_TYPE_<N>`
- `OFFLINE_MACHINE_RESOURCE_<name>`

3.3.12 Pre-Defined Macros

HTCondor provides pre-defined macros that help configure HTCondor. 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. This implies that a change to the underlying value of any of these variables will require a full restart of HTCondor in order to use the changed value.

$(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, *without* a domain name.

$(IP_ADDRESS) The ASCII string version of the local machine's "most public" IP address. This address may be IPv4 or IPv6, but the macro will always be set.

HTCondor selects the "most public" address heuristically. Your configuration should not depend on HTCondor picking any particular IP address for this macro; this macro's value may not even be one of the IP addresses HTCondor is configured to advertise.

labelparam:IPv4Address

$(IPV4_ADDRESS) The ASCII string version of the local machine's "most public" IPv4 address; unset if the local machine has no IPv4 address.

See `IP_ADDRESS` about "most public".

$(IPV6_ADDRESS) The ASCII string version of the local machine's "most public" IPv6 address; unset if the local machine has no IPv6 address.

See `IP_ADDRESS` about "most public".

$(IP_ADDRESS_IS_V6) A boolean which is true if and only if `IP_ADDRESS` is an IPv6 address. Useful for conditonal configuration.

$(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 HTCondor system. The possible subsystem names are:

- `C_GAHP`
- `C_GAHP_WORKER_THREAD`
- `CKPT_SERVER`
- `COLLECTOR`
- `DBMSD`
- `DEFRAG`
- `EC2_GAHP`
- `GANGLIAD`
- `GCE_GAHP`
- `GRIDMANAGER`
- `HAD`
- `HDFS`
- `JOB_ROUTER`
- `KBDD`
- `LEASEMANAGER`
- `MASTER`

- NEGOTIATOR
- QUILL
- REPLICATION
- ROOSTER
- SCHEDD
- SHADOW
- SHARED_PORT
- STARTD
- STARTER
- SUBMIT
- TOOL
- TRANSFERER

$(DETECTED_CPUS) The integer number of hyper-threaded CPUs, as given by $(DETECTED_CORES), when COUNT_HYPERTHREAD_CPUS is True. The integer number of physical (non hyper-threaded) CPUs, as given by $(DETECTED_PHYSICAL_CPUS), when COUNT_HYPERTHREAD_CPUS is False. When COUNT_HYPERTHREAD_CPUS is True.

$(DETECTED_PHYSICAL_CPUS) The integer number of physical (non hyper-threaded) CPUs. This will be equal the number of unique CPU IDs.

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 HTCondor. 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 896 for details.

$(OPSYS) Defines the string used to identify the operating system of the local machine to HTCondor. If it is not defined in the configuration file, HTCondor will automatically insert the operating system of this machine as determined by *uname*.

$(OPSYS_VER) Defines the integer used to identify the operating system version number.

$(OPSYS_AND_VER) Defines the string used prior to HTCondor version 7.7.2 as $(OPSYS).

$(UNAME_ARCH) The architecture as reported by *uname*(2)'s machine field. Always the same as ARCH on Windows.

$(UNAME_OPSYS) The operating system as reported by *uname*(2)'s sysname field. Always the same as OPSYS on Windows.

$(DETECTED_MEMORY) The amount of detected physical memory (RAM) in MiB.

$(DETECTED_CORES) The number of CPU cores that the operating system schedules. On machines that support hyper-threading, this will be the number of hyper-threads.

$(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.

$(FILESYSTEM_DOMAIN) Defaults to the fully qualified host name of the machine it is evaluated on. See section 3.5.5, Shared File System Configuration File Entries for the full description of its use and under what conditions it could be desirable to change it.

$(UID_DOMAIN) Defaults to the fully qualified host name of the machine it is evaluated on. See section 3.5.5 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.

3.4 Configuration Templates

Achieving certain behaviors in an HTCondor pool often requires setting the values of a number of configuration macros in concert with each other. We have added configuration templates as a way to do this more easily, at a higher level, without having to explicitly set each individual configuration macro.

Configuration templates are pre-defined; users cannot define their own templates.

Note that the value of an individual configuration macro that is set by a configuration template can be overridden by setting that configuration macro later in the configuration.

Detailed information about configuration templates (such as the macros they set) can be obtained using the *condor_config_val* `use` option (see 11). (This document does not contain such information because the *condor_config_val* command is a better way to obtain it.)

3.4.1 Configuration Templates: Using Predefined Sets of Configuration

Predefined sets of configuration can be identified and incorporated into the configuration using the syntax

```
use <category name> : <template name>
```

The `use` key word is case insensitive. There are *no* requirements for white space characters surrounding the colon character. More than one `<template name>` identifier may be placed within a single `use` line. Separate the names by a space character. There is no mechanism by which the administrator may define their own custom `<category name>` or `<template name>`.

Each predefined `<category name>` has a fixed, case insensitive name for the sets of configuration that are predefined. Placement of a `use` line in the configuration brings in the predefined configuration it identifies.

As of version 8.5.6, some of the configuration templates take arguments (as described below).

3.4.2 Available Configuration Templates

There are four `<category name>` values. Within a category, a predefined, case insensitive name identifies the set of configuration it incorporates.

ROLE category Describes configuration for the various roles that a machine might play within an HTCondor pool. The configuration will identify which daemons are running on a machine.

- `Personal`
  Settings needed for when a single machine is the entire pool.
- `Submit`
  Settings needed to allow this machine to submit jobs to the pool. May be combined with `Execute` and `CentralManager` roles.
- `Execute`
  Settings needed to allow this machine to execute jobs. May be combined with `Submit` and `CentralManager` roles.
- `CentralManager`
  Settings needed to allow this machine to act as the central manager for the pool. May be combined with `Submit` and `Execute` roles.

FEATURE category Describes configuration for implemented features.

- `Remote_Runtime_Config`
  Enables the use of *condor_config_val* **-rset** to the machine with this configuration. Note that there are security implications for use of this configuration, as it potentially permits the arbitrary modification of configuration. Variable `SETTABLE_ATTRS_CONFIG` must also be defined.
- `Remote_Config`
  Enables the use of *condor_config_val* **-set** to the machine with this configuration. Note that there are security implications for use of this configuration, as it potentially permits the arbitrary modification of configuration. Variable `SETTABLE_ATTRS_CONFIG` must also be defined.
- `VMware`
  Enables use of the vm universe with VMware virtual machines. Note that this feature depends on Perl.
- `GPUs`
  Sets configuration based on detection with the *condor_gpu_discovery* tool, and defines a custom resource using the name `GPUs`. Supports both OpenCL and CUDA if detected.
- `PartitionableSlot( slot_type_num [, allocation] )`
  Sets up a partitionable slot of the specified slot type number and allocation (defaults for slot_type_num and allocation are 1 and 100% respectively). See 3.7.1 for information on partitionalble slot policies.

- `AssignAccountingGroup( map_filename )` Sets up a *condor_schedd* job transform that assigns an accounting group to each job as it is submitted. The accounting is determined by mapping the Owner attribute of the job using the given map file.
- `ScheddUserMapFile( map_name, map_filename )` Defines a *condor_schedd* usermap named map_name using the given map file.
- `SetJobAttrFromUserMap( dst_attr, src_attr, map_name [, map_filename] )` Sets up a *condor_schedd* job transform that sets the dst_attr attribute of each job as it is submitted. The value of dst_attr is determined by mapping the src_attr of the job using the usermap named map_name. If the optional map_filename argument is specifed, then this metaknob also defines a *condor_schedd* usermap named map_Name using the given map file.
- `StartdCronOneShot( job_name, exe [, hook_args] )`
  Create a one-shot *condor_startd* job hook. (See 4.4.3 for more information about job hooks.)
- `StartdCronPeriodic( job_name, period, exe [, hook_args] )`
  Create a periodic-shot *condor_startd* job hook. (See 4.4.3 for more information about job hooks.)
- `StartdCronContinuous( job_name, exe [, hook_args] )`
  Create a (nearly) continuous *condor_startd* job hook. (See 4.4.3 for more information about job hooks.)
- `ScheddCronOneShot( job_name, exe [, hook_args] )`
  Create a one-shot *condor_schedd* job hook. (See 4.4.3 for more information about job hooks.)
- `ScheddCronPeriodic( job_name, period, exe [, hook_args] )`
  Create a periodic-shot *condor_schedd* job hook. (See 4.4.3 for more information about job hooks.)
- `ScheddCronContinuous( job_name, exe [, hook_args] )`
  Create a (nearly) continuous *condor_schedd* job hook. (See 4.4.3 for more information about job hooks.)
- `OneShotCronHook( STARTD_CRON | SCHEDD_CRON, job_name, hook_exe [,hook_args] )`
  Create a one-shot job hook. (See 4.4.3 for more information about job hooks.)
- `PeriodicCronHook( STARTD_CRON | SCHEDD_CRON , job_name, period, hook_exe [,hook_args] )`
  Create a periodic job hook. (See 4.4.3 for more information about job hooks.)
- `ContinuousCronHook( STARTD_CRON | SCHEDD_CRON , job_name, hook_exe [,hook_args] )`
  Create a (nearly) continuous job hook. (See 4.4.3 for more information about job hooks.)

- `UWCS_Desktop_Policy_Values`
  Configuration values used in the `UWCS_DESKTOP` policy. (Note that these values were previously in the parameter table; configuration that uses these values will have to use the `UWCS_Desktop_Policy_Values` template. For example, `POLICY : UWCS_Desktop` uses the `FEATURE : UWCS_Desktop_Policy_Values` template.)

`POLICY category` Describes configuration for the circumstances under which machines choose to run jobs.

- `Always_Run_Jobs`

  Always start jobs and run them to completion, without consideration of *condor_negotiator* generated preemption or suspension. This is the default policy, and it is intended to be used with dedicated resources. If this policy is used together with the `Limit_Job_Runtimes` policy, order the specification by placing this `Always_Run_Jobs` policy first.

- `UWCS_Desktop`

  This was the default policy before HTCondor version 8.1.6. It is intended to be used with desktop machines not exclusively running HTCondor jobs. It injects `UWCS` into the name of some configuration variables.

- `Desktop`

  An updated and reimplementation of the `UWCS_Desktop` policy, but *without* the `UWCS` naming of some configuration variables.

- `Limit_Job_Runtimes( limit_in_seconds )`

  Limits running jobs to a maximum of the specified time using preemption. (The default limit is 24 hours.) If this policy is used together with the `Always_Run_Jobs` policy, order the specification by placing this `Limit_Job_Runtimes` policy second.

- `Preempt_If_Cpus_Exceeded`

  If the startd observes the number of CPU cores used by the job exceed the number of cores in the slot by more than 0.8 on average over the past minute, preempt the job immediately ignoring any job retirement time.

- `Hold_If_Cpus_Exceeded`

  If the startd observes the number of CPU cores used by the job exceed the number of cores in the slot by more than 0.8 on average over the past minute, immediately place the job on hold ignoring any job retirement time. The job will go on hold with a reasonable hold reason in job attribute `HoldReason` and a value of 101 in job attribute `HoldReasonCode`. The hold reason and code can be customized by specifying `HOLD_REASON_CPU_EXCEEDED` and `HOLD_SUBCODE_CPU_EXCEEDED` respectively.

  Standard universe jobs can't be held by startd policy expressions, so this metaknob automatically ignores them.

- `Preempt_If_Memory_Exceeded`

  If the startd observes the memory usage of the job exceed the memory provisioned in the slot, preempt the job immediately ignoring any job retirement time.

- `Hold_If_Memory_Exceeded`

  If the startd observes the memory usage of the job exceed the memory provisioned in the slot, immediately place the job on hold ignoring any job retirement time. The job will go on hold with a reasonable hold reason in job attribute `HoldReason` and a value of 102 in job attribute `HoldReasonCode`. The hold reason and code can be customized by specifying `HOLD_REASON_MEMORY_EXCEEDED` and `HOLD_SUBCODE_MEMORY_EXCEEDED` respectively.

  Standard universe jobs can't be held by startd policy expressions, so this metaknob automatically ignores them.

- `Preempt_If( policy_variable )`

  Preempt jobs according to the specified policy. `policy_variable` must be the name of a configuration macro containing an expression that evaluates to `True` if the job should be preempted.

  See an example here: 3.4.4.

- `Want_Hold_If( policy_variable, subcode, reason_text )`
  Add the given policy to the `WANT_HOLD` expression; if the `WANT_HOLD` expression is defined, `policy_variable` is prepended to the existing expression; otherwise `WANT_HOLD` is simply set to the value of the textttpolicy_variable macro.
  Standard universe jobs can't be held by startd policy expressions, so this metaknob automatically ignores them.
  See an example here: 3.4.4.
- `Startd_Publish_CpusUsage`
  Publish the number of CPU cores being used by the job into to slot ad as attribute `CpusUsage`. This value will be the average number of cores used by the job over the past minute, sampling every 5 seconds.

`SECURITY category` Describes configuration for an implemented security model.

- `Host_Based`
  The default security model (based on IPs and DNS names). Do *not* combine with `User_Based` security.
- `User_Based`
  Grants permissions to an administrator and uses `With_Authentication`. Do *not* combine with `Host_Based` security.
- `With_Authentication`
  Requires both authentication and integrity checks.
- `Strong`
  Requires authentication, encryption, and integrity checks.

3.4.3 Configuration Template Transition Syntax

For pools that are transitioning to using this new syntax in configuration, while still having some tools and daemons with HTCondor versions earlier than 8.1.6, special syntax in the configuration will cause those daemons to fail upon start up, rather than use the new, but misinterpreted, syntax. Newer daemons will ignore the extra syntax. Placing the `@` character before the `use` key word causes the older daemons to fail when they attempt to parse this syntax.

As an example, consider the *condor_startd* as it starts up. A *condor_startd* previous to HTCondor version 8.1.6 fails to start when it sees:

```
@use feature : GPUs
```

Running an older *condor_config_val* also identifies the `@use` line as being bad. A *condor_startd* of HTCondor version 8.1.6 or more recent sees

```
use feature : GPUs
```

3.4.4 Configuration Template Examples

- Preempt a job if its memory usage exceeds the requested memory:

```
MEMORY_EXCEEDED = (isDefined(MemoryUsage) && MemoryUsage > RequestMemory)
use POLICY : PREEMPT_IF(MEMORY_EXCEEDED)
```

- Put a job on hold if its memory usage exceeds the requested memory:

```
MEMORY_EXCEEDED = (isDefined(MemoryUsage) && MemoryUsage > RequestMemory)
use POLICY : WANT_HOLD_IF(MEMORY_EXCEEDED, 102, memory usage exceeded request_memor
```

- Update dynamic GPU information every 15 minutes:

```
use FEATURE : StartdCronPeriodic(DYNGPU, 15*60, $(LOCAL_DIR)\dynamic_gpu_info.pl, $
```

  where `dynamic_gpu_info.pl` is a simple perl script that strips off the DetectedGPUs line from textttcondor_gpu_discovery:

```
#!/usr/bin/env perl
my @attrs = `@ARGV`;
for (@attrs) {
next if ($_ =~ /^Detected/i);
print $_;
}
```

3.5 Configuration Macros

The section contains a list of the individual configuraton macros for HTCondor. Before attempting to set up HTCondor configuration, you should probably read the introduction to configuration section (3.3) and possibly the configuration template section (3.4).

The settings that control the policy under which HTCondor will start, suspend, resume, vacate or kill jobs are described in section 3.7 on Policy Configuration for the *condor_startd*, not in this section.

3.5.1 HTCondor-wide Configuration File Entries

This section describes settings which affect all parts of the HTCondor system. Other system-wide settings can be found in section 3.5.4 on “Network-Related Configuration File Entries”, and section 3.5.5 on “Shared File System Configuration File Entries”.

CONDOR_HOST This macro 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 3.8.9, on Setting up IP/host-based security in HTCondor 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 3.9.1 on page 433.

Multiple *condor_collector* daemons may be running simultaneously, if COLLECTOR_HOST is defined with a comma separated list of hosts. Multiple *condor_collector* daemons may run for the implementation of high availability; see section 3.13 for details. With more than one running, updates are sent to all. With more than one running, queries are sent to one of the *condor_collector* daemons, chosen at random.

COLLECTOR_PORT The default port used when contacting the *condor_collector* and the default port the *condor_collector* listens on if no port is specified. This variable is referenced if no port is given and there is no other means to find the *condor_collector* port. The default value is 9618.

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 A list of HTCondorView servers, separated by commas and/or spaces. Each HTCondorView server is denoted by the host name of the machine it is running on, optionally appended by a colon and the port number. 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 HTCondorView server is used. See section 3.14.6 on page 472 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. If the host specifies SCHEDD_NAME or MASTER_NAME, that name must be included in the form name@hostname. 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 with the optional name included. For most pools, this macro is not defined, nor does it need to be defined..

RELEASE_DIR The full path to the HTCondor 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 HTCondor directory where user-level programs are installed. The default value is $(RELEASE_DIR)/bin.

LIB This directory points to the HTCondor directory where libraries used to link jobs for HTCondor'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. The default value is `$(RELEASE_DIR)/lib`.

LIBEXEC This directory points to the HTCondor directory where support commands that HTCondor needs will be placed. Do not add this directory to a user or system-wide path.

INCLUDE This directory points to the HTCondor directory where header files reside. The default value is `$(RELEASE_DIR)/include`. 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 HTCondor directory where HTCondor's system binaries (such as the binaries for the HTCondor daemons) and administrative tools are installed. Whatever directory `$(SBIN)` points to ought to be in the `PATH` of users acting as HTCondor administrators. The default value is `$(BIN)` in Windows and `$(RELEASE_DIR)/sbin` on all other platforms.

LOCAL_DIR The location of the local HTCondor directory on each machine in your pool. The default value is `$(RELEASE_DIR)` on Windows and `$(RELEASE_DIR)/hosts/$(HOSTNAME)` on all other platforms.

Another possibility is to use the condor user's home directory, which may be specified with `$(TILDE)`. For example:

```
LOCAL_DIR = $(tilde)
```

LOG Used to specify the directory where each HTCondor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the `$(LOG)` macro by default. The log directory also acts as the current working directory of the HTCondor daemons as the run, so if one of them should produce a core file for any reason, it would be placed in the directory defined by this macro. The default value is `$(LOCAL_DIR)/log`.

Do not stage other files in this directory; any files not created by HTCondor in this directory are subject to removal.

RUN A path and directory name to be used by the HTCondor init script to specify the directory where the *condor_master* should write its process ID (PID) file. The default if not defined is `$(LOG)`.

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 HTCondor jobs and not submit them, it would not need a spool directory (or this macro defined). The default value is `$(LOCAL_DIR)/spool`. The *condor_schedd* will not function if `SPOOL` is not defined.

Do not stage other files in this directory; any files not created by HTCondor in this directory are subject to removal.

EXECUTE This directory acts as a place to create the scratch directory of any HTCondor 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. The default value is `$(LOCAL_DIR)/execute`. The *condor_startd* will not function if `EXECUTE` is undefined. To customize the execute directory independently for each batch slot, use `SLOT<N>_EXECUTE`.

Do not stage other files in this directory; any files not created by HTCondor in this directory are subject to removal.

TMP_DIR A directory path to a directory where temporary files are placed by various portions of the HTCondor system. The daemons and tools that use this directory are the *condor_gridmanager*, *condor_config_val* when using the **-rset** option, systems that use lock files when configuration variable `CREATE_LOCKS_ON_LOCAL_DISK` is `True`, the Web Service API, and the *condor_credd* daemon. There is no default value.

If both `TMP_DIR` and `TEMP_DIR` are defined, the value set for `TMP_DIR` is used and `TEMP_DIR` is ignored.

TEMP_DIR A directory path to a directory where temporary files are placed by various portions of the HTCondor system. The daemons and tools that use this directory are the *condor_gridmanager*, *condor_config_val* when using the **-rset** option, systems that use lock files when configuration variable `CREATE_LOCKS_ON_LOCAL_DISK` is `True`, the Web Service API, and the *condor_credd* daemon. There is no default value.

If both `TMP_DIR` and `TEMP_DIR` are defined, the value set for `TMP_DIR` is used and `TEMP_DIR` is ignored.

SLOT<N>_EXECUTE Specifies an execute directory for use by a specific batch slot. `<N>` represents the number of the batch slot, such as 1, 2, 3, etc. This execute directory serves the same purpose as `EXECUTE`, but it allows the configuration of 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 3.3.6 on page 191 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 3.14.3 about Configuring HTCondor for Multiple Platforms on page 467.

If all files in a directory are local configuration files to be processed, then consider using LOCAL_CONFIG_DIR, defined at section 3.5.1.

REQUIRE_LOCAL_CONFIG_FILE A boolean value that defaults to True. When True, HTCondor 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 A directory may be used as a container for local configuration files. The files found in the directory are sorted into lexicographical order by file name, 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. See also LOCAL_CONFIG_DIR_EXCLUDE_REGEXP.

USER_CONFIG_FILE The file name of a configuration file to be parsed after other local configuration files and before environment variables set configuration. Relevant only if HTCondor daemons are *not* run as root on Unix platforms or Local System on Windows platforms. The default is $(HOME)/.condor/user_config on Unix platforms. The default is %USERPROFILE%\.condor\user_config on Windows platforms. If a fully qualified path is given, that is used. If a fully qualified path is *not* given, then the Unix path $(HOME)/.condor/ prefixes the file name given on Unix platforms, or the Windows path %USERPROFILE%\.condor\ prefixes the file name given on Windows platforms.

The ability of a user to use this user-specified configuration file can be disabled by setting this variable to the empty string:

```
USER_CONFIG_FILE =
```

LOCAL_CONFIG_DIR_EXCLUDE_REGEXP A regular expression that specifies file names to be ignored when looking for configuration files within the directories specified via LOCAL_CONFIG_DIR. The default expression ignores files with names beginning with a '.' or a '#', as well as files with names ending in '~'. This avoids accidents that can be caused by treating temporary files created by text editors as configuration files.

CONDOR_IDS The User ID (UID) and Group ID (GID) pair that the HTCondor 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 HTCondor 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 HTCondor daemons are spawned as root, then HTCondor will search for a `condor` user on the system, and run as that user's UID and GID. See section 3.8.13 on UIDs in HTCondor for more details.

CONDOR_ADMIN The email address that HTCondor will send mail to if something goes wrong in the 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. The default value is root@`$(FULL_HOSTNAME)`.

<SUBSYS>_ADMIN_EMAIL The email address that HTCondor will send mail to if something goes wrong with the named `<SUBSYS>`. Identical to `CONDOR_ADMIN`, but done on a per subsystem basis. There is no default value.

CONDOR_SUPPORT_EMAIL The email address to be included at the bottom of all email HTCondor sends out under the label "Email address of the local HTCondor administrator:". This is the address where HTCondor users at your site should send their questions about HTCondor and get technical support. If this setting is not defined, HTCondor will use the address specified in `CONDOR_ADMIN` (described above).

EMAIL_SIGNATURE Every e-mail sent by HTCondor includes a short signature line appended to the body. By default, this signature includes the URL to the global HTCondor project website. When set, this variable defines an alternative signature line to be used instead of the default. Note that the value can only be one line in length. This variable could be used to direct users to look at local web site with information specific to the installation of HTCondor.

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 HTCondor should work. Only if you installed things in a non-standard location on your system would you need to change this setting. The default value is `$(BIN)/condor_mail.exe` on Windows and `/usr/bin/mail` on all other platforms. The *condor_schedd* will not function unless `MAIL` is defined. For security reasons, non-Windows platforms should not use this setting and should use `SENDMAIL` instead.

SENDMAIL The full path to the *sendmail* executable. If defined, which it is by default on non-Windows platforms, *sendmail* is used instead of the mail program defined by `MAIL`.

MAIL_FROM The e-mail address that notification e-mails appear to come from. Contents is that of the `From` header. There is no default value; if undefined, the `From` header may be nonsensical.

SMTP_SERVER For Windows platforms only, the host name of the server through which to route notification e-mail. There is no default value; if undefined and the debug level is at `FULLDEBUG`, an error message will be generated.

RESERVED_SWAP The amount of swap space in MiB to reserve for this machine. HTCondor will not start up more *condor_shadow* processes if the amount of free swap space on this machine falls below this level. The default value is 0, which disables this check. It is anticipated that this configuration variable will no longer be used in the near future. If `RESERVED_SWAP` is *not* set to 0, the value of `SHADOW_SIZE_ESTIMATE` is used.

RESERVED_DISK Determines how much disk space you want to reserve for your own machine. When HTCondor 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 HTCondor 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 HTCondor 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 HTCondor config files. See section 3.8.13 on UIDs in HTCondor for details.

If no value for `LOCK` is provided, the value of `LOG` is used.

HISTORY Defines the location of the HTCondor history file, which stores information about all HTCondor 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.

HISTORY_HELPER_MAX_CONCURRENCY Specifies the maximum number of concurrent remote *condor_history* queries allowed at a time; defaults to 2. When this maximum is exceeded, further queries will be queued in a non-blocking manner. Setting this option to 0 disables remote history access. A remote history access is defined as an invocation of *condor_history* that specifies a **-name** option to query a *condor_schedd* running on a remote machine.

HISTORY_HELPER_MAX_HISTORY Specifies the maximum number of ClassAds to parse on behalf of remote history clients. The default is 10,000. This allows the system administrator to indirectly manage the maximum amount of CPU time spent on each client. Setting this option to 0 disables remote history access.

MAX_JOB_QUEUE_LOG_ROTATIONS The *condor_schedd* daemon 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.

CLASSAD_LOG_STRICT_PARSING A boolean value that defaults to `True`. When `True`, ClassAd log files will be read using a strict syntax checking for ClassAd expressions. ClassAd log files include the job queue log and the accountant log. When `False`, ClassAd log files are read without strict expression syntax checking, which allows some legacy ClassAd log data to be read in a backward compatible manner. This configuration variable may no longer be supported in future releases, eventually requiring all ClassAd log files to pass strict ClassAd syntax checking.

DEFAULT_DOMAIN_NAME The value to be appended to a machine's host name, representing a domain name, which HTCondor 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 HTCondor 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 HTCondor download is bogus, and must be changed. If this variable is removed from the global configuration file, or if the definition is empty, then HTCondor attempts to discover the value.

NO_DNS A boolean value that defaults to `False`. When `True`, HTCondor constructs host names using the host's IP address together with the value defined for `DEFAULT_DOMAIN_NAME`.

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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor daemons (normally the *condor_master*) were started. This allows HTCondor 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 HTCondor. 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 HTCondor 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 HTCondor. You set the parameter to whatever behavior you want HTCondor 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 HTCondor 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 HTCondor 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_COLLECTOR_MAX_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 HTCondor can cause NIS servers to become overwhelmed by queries for uid and group information in large pools. In order to avoid this problem, HTCondor caches UID and group information internally. This integer value allows pool administrators to specify (in seconds) how long HTCondor should wait until refreshes a cache entry. The default is set to 72000 seconds, or 20 hours, 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 HTCondor 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*. This configuration variable has no effect on Windows.

SYSAPI_GET_LOADAVG If set to False, then HTCondor 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 HTCondor 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 HTCondor 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.

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 3.9.5 on page 442 for more information).

ALLOW_SCRIPTS_TO_RUN_AS_EXECUTABLES A boolean value that, when `True`, permits scripts on Windows platforms to be used in place of the **executable** in a job submit description file, in place of a *condor_dagman* pre or post script, or in producing the configuration, for example. Allows a script to be used in any circumstance previously limited to a Windows executable or a batch file. The default value is `True`. See section 7.2.7 on page 646 for further description.

OPEN_VERB_FOR_<EXT>_FILES A string that defines a Windows *verb* for use in a root hive registry look up. `<EXT>` defines the file name extension, which represents a scripting language, also needed for the look up. See section 7.2.7 on page 646 for a more complete description.

ENABLE_CLASSAD_CACHING A boolean value that controls the caching of ClassAds. Caching saves memory when an HTCondor process contains many ClassAds with the same expressions. The default value is `True` for all daemons other than the *condor_shadow*, *condor_starter*, and *condor_master*. A value of `True` enables caching.

STRICT_CLASSAD_EVALUATION A boolean value that controls how ClassAd expressions are evaluated. If set to `True`, then New ClassAd evaluation semantics are used. This means that attribute references without a `MY.` or `TARGET.` prefix are only looked up in the local ClassAd. If set to the default value of `False`, Old ClassAd evaluation semantics are used. See section 4.1.1 on page 505 for details.

CLASSAD_USER_LIBS A comma separated list of paths to shared libraries that contain additional ClassAd functions to be used during ClassAd evaluation.

CLASSAD_USER_PYTHON_MODULES A comma separated list of python modules to load, which are to be used during ClassAd evaluation. If module `foo` is in this list, then function `bar` can be invoked in ClassAds via the expression `python_invoke("foo", "bar", ...)`. Any further arguments are converted from ClassAd expressions to python; the function return value is converted back to ClassAds. The python modules are loaded at configuration time, so any module-level statements are executed. Module writers can invoke `classad.register` at the module-level in order to use python functions directly.

Functions executed by ClassAds should be non-blocking and have no side-effects; otherwise, unpredictable HTCondor behavior may occur.

CLASSAD_USER_PYTHON_LIB Specifies the path to the python libraries, which is needed when `CLASSAD_USER_PYTHON_MODULES` is set. Defaults to `$(LIBEXEC)/libclassad_python_user.so`, and would rarely be changed from the default value.

CONDOR_FSYNC A boolean value that controls whether HTCondor calls `fsync()` when writing the user job and transaction logs. Setting this value to `False` will disable calls to `fsync()`, which can help performance for *condor_schedd* log writes at the cost of some durability of the log contents, should there be a power or hardware failure. The default value is `True`.

STATISTICS_TO_PUBLISH A comma and/or space separated list that identifies which statistics collections are to place attributes in ClassAds. Additional information specifies a level of verbosity and other identification of which attributes to include and which to omit from ClassAds. The special value `NONE` disables all publishing, so no statistics will be published; no option is included. For other list items that define this variable, the syntax defines the two aspects by separating them with a colon. The first aspect defines a collection, which may specify which daemon is to publish the statistics, and the second aspect qualifies and refines the details of which attributes to publish for the collection, including a verbosity level. If the first aspect is `ALL`, the option is applied to all collections. If the first aspect is `DEFAULT`, the option is applied to all collections, with the intent that further list items will specify publishing that is to be different than the default. This first aspect may be `SCHEDD` or `SCHEDULER` to publish Statistics attributes in the ClassAd of the *condor_schedd*. It may be `TRANSFER` to publish file transfer statistics. It may be `STARTER` to publish Statistics attributes in the ClassAd of the *condor_starter*. Or, it may be `DC` or `DAEMONCORE` to publish DaemonCore statistics. One or more options are specified after the colon.

Option	*Description*
0	turns off the publishing of any statistics attributes
1	the default level, where some statistics attributes are published and others are omitted
2	the verbose level, where all statistics attributes are published
3	the super verbose level, which is currently unused, but intended to be all statistics attributes published at the verbose level plus extra information
R	include attributes from the most recent time interval; the default
!R	omit attributes from the most recent time interval
D	include attributes for debugging
!D	omit attributes for debugging; the default
Z	include attributes even if the attribute's value is 0
!Z	omit attributes when the attribute's value is 0
L	include attributes that represent the lifetime value; the default
!L	omit attributes that represent the lifetime value

If this variable is not defined, then the default for each collection is used. If this variable is defined, and the definition does not specify each possible collection, then no statistics are published for those collections not defined. If an option specifies conflicting possibilities, such as `R!R`, then the last one takes precedence and is applied.

As an example, to cause a verbose setting of the publication of Statistics attributes only for the *condor_schedd*, and do not publish any other Statistics attributes:

```
STATISTICS_TO_PUBLISH = SCHEDD:2
```

As a second example, to cause all collections other than those for `DAEMONCORE` to publish at a verbosity setting of `1`, and omit lifetime values, where the `DAEMONCORE` includes all statistics at the verbose level:

```
STATISTICS_TO_PUBLISH = DEFAULT:1!L, DC:2RDZL
```

STATISTICS_TO_PUBLISH_LIST A comma and/or space separated list of statistics attribute names that should be published in updates to the *condor_collector* daemon, even though the verbosity specified in `STATISTICS_TO_PUBLISH` would not normally send them. This setting has the effect of redefining the verbosity level of the statistics attributes that it mentions, so that they will always match the current statistics publication level as specified in `STATISTICS_TO_PUBLISH`.

STATISTICS_WINDOW_SECONDS An integer value that controls the time window size, in seconds, for collecting windowed daemon statistics. These statistics are, by convention, those attributes with names that are of the form `Recent<attrname>`. Any data contributing to a windowed statistic that is older than this number of seconds is dropped from the statistic. For example, if `STATISTICS_WINDOW_SECONDS = 300`, then any jobs submitted more than 300 seconds ago are not counted in the windowed statistic `RecentJobsSubmitted`. Defaults to 1200 seconds, which is 20 minutes.

The window is broken into smaller time pieces called quantum. The window advances one quantum at a time.

STATISTICS_WINDOW_SECONDS_<collection> The same as `STATISTICS_WINDOW_SECONDS`, but used to override the global setting for a particular statistic collection. Collection names currently implemented are `DC` or `DAEMONCORE` and `SCHEDD` or `SCHEDULER`.

STATISTICS_WINDOW_QUANTUM For experts only, an integer value that controls the time quantization that form a time window, in seconds, for the data structures that maintain windowed statistics. Defaults to 240 seconds, which is 6 minutes. This default is purposely set to be slightly smaller than the update rate to the *condor_collector*. Setting a smaller value than the default increases the memory requirement for the statistics. Graphing of statistics at the level of the quantum expects to see counts that appear like a saw tooth.

STATISTICS_WINDOW_QUANTUM_<collection> The same as STATISTICS_WINDOW_QUANTUM, but used to override the global setting for a particular statistic collection. Collection names currently implemented are DC or DAEMONCORE and SCHEDD or SCHEDULER.

TCP_KEEPALIVE_INTERVAL The number of seconds specifying a keep alive interval to use for any HTCondor TCP connection. The default keep alive interval is 360 (6 minutes); this value is chosen to minimize the likelihood that keep alive packets are sent, while still detecting dead TCP connections before job leases expire. A smaller value will consume more operating system and network resources, while a larger value may cause jobs to fail unnecessarily due to network disconnects. Most users will not need to tune this configuration variable. A value of 0 will use the operating system default, and a value of -1 will disable HTCondor's use of a TCP keep alive.

ENABLE_IPV4 A boolean with the additional special value of auto. If true, HTCondor will use IPv4 if available, and fail otherwise. If false, HTCondor will not use IPv4. If auto, HTCondor will use IPv4 if it can find an interface with an IPv4 address; this is the default.

ENABLE_IPV6 A boolean with the additional special value of auto. If true, HTCondor will use IPv6 if available, and fail otherwise. If false, HTCondor will not use IPv6. If auto, HTCondor will use IPv6 if it can find an interface with an IPv6 address; this is the default.

PREFER_IPV4 A boolean which will cause HTCondor to prefer IPv4 when it is able to choose. HTCondor will otherwise prefer IPv6. The default is True.

ADVERTISE_IPV4_FIRST A string (treated as a boolean). If ADVERTISE_IPV4_FIRST evaluates to True, HTCondor will advertise its IPv4 addresses before its IPv6 addresses; otherwise the IPv6 addresses will come first. Defaults to $(PREFER_IPV4).

IGNORE_TARGET_PROTOCOL_PREFERENCE A string (treated as a boolean). If IGNORE_TARGET_PROTOCOL_PREFERENCE evaluates to True, the target's listed protocol preferences will be ignored; othwerwise they will not. Defaults to $(PREFER_IPV4).

IGNORE_DNS_PROTOCOL_PREFERENCE A string (treated as a boolean). IGNORE_DNS_PROTOCOL_PREFERENCE evaluates to True, the protocol order returned by the DNS will be ignored; otherwise it will not. Defaults to $(PREFER_IPV4).

PREFER_OUTBOUND_IPV4 A string (treated as a boolean). PREFER_OUTBOUND_IPV4 evaluates to True, HTCondor will prefer IPv4; otherwise it will not. Defaults to $(PREFER_IPV4).

<SUBSYS>_CLASSAD_USER_MAP_NAMES A string defining a list of names for username-to-accounting group mappings for the specified daemon. Names must be separated by spaces or commas.

CLASSAD_USER_MAPFILE_<name> A string giving the name of a file to parse to initialize the map for the given username. Note that this macro is only used if <SUBSYS>_CLASSAD_USER_MAP_NAMES is defined for the relevant daemon.

CLASSAD_USER_MAPDATA_<name> A string containing data to be used to initialize the map for the given username. Note that this macro is only used if <SUBSYS>_CLASSAD_USER_MAP_NAMES is defined for the relevant daemon, and CLASSAD_USER_MAPFILE_<name> is *not* defined for the given name.

The format for the map file and map data is the same as the format for the security unified map file (see 3.8.4 for details).

The first field must be * (or a subset name - see below), the second field is a regex that we will match against the input, and the third field will be the output if the regex matches, the 3 and 4 argument form of the ClassAd userMap() function (see 4.1.2) expect that the third field will be a comma separated list of values. for example:

```
# file: groups.mapdata
* John  chemistry,physics,glassblowing
* Juan  physics,chemistry
* Bob   security
* Alice security,math
```

Optional submaps: If the first field of the mapfile contains something other than *, then a submap is defined. To select a submap for lookup, the first argument for userMap() should be "mapname.submap". For example:

```
# mapdata 'groups' with submaps
*   Bob   security
*   Alice security,math
alt Alice math,hacking
```

IGNORE_LEAF_OOM A boolean value that, when True, tells HTCondor *not* to kill and hold a job that is within its memory allocation, even if other processes within the same cgroup have exceeded theirs. The default value is True. (Note that this represents a change in behavior compared to versions of HTCondor older than 8.6.0; this configuration macro first appeared in version 8.4.11. To restore the previous behavior, set this value to False.)

3.5.2 Daemon Logging Configuration File Entries

These entries control how and where the HTCondor daemons write to log files. Many of the entries in this section represents multiple macros. There is one for each subsystem (listed in section 3.3.12). The macro name for each substitutes <SUBSYS> with the name of the subsystem corresponding to the daemon.

<SUBSYS>_LOG Defines the path and file name of the log file for a given subsystem. For example, $(STARTD_LOG) gives the location of the log file for the *condor_startd* daemon. The default value for most daemons is the daemon's name in camel case, concatenated with Log. For example, the default log defined for the *condor_master* daemon is $(LOG)/MasterLog. The default value for other subsystems is $(LOG)/<SUBSYS>LOG. If the log file cannot be written to, then the daemon will attempt to log this into a new file of the name $(LOG)/dprintf_failure.<SUBSYS> before the daemon exits.

MAX_<SUBSYS>_LOG Controls the maximum size in bytes or amount of time that a log will be allowed to grow. For any log not specified, the default is $(MAX_DEFAULT_LOG), which currently defaults to 10 MiB in size. Values are specified with the same syntax as MAX_DEFAULT_LOG.

Note that a log file for the *condor_procd* does not use this configuration variable definition. Its implementation is separate. See section 3.5.16 for the definition of MAX_PROCD_LOG.

MAX_DEFAULT_LOG Controls the maximum size in bytes or amount of time that any log not explicitly specified using MAX_<SUBSYS>_LOG will be allowed to grow. When it is time to rotate a log file, it will be saved to a file with an ISO timestamp suffix. The oldest rotated file receives the ending .old. The .old files are overwritten each time the maximum number of rotated files (determined by the value of MAX_NUM_<SUBSYS>_LOG) is exceeded. The default value is 10 MiB in size. A value of 0 specifies that the file may grow without bounds. A single integer value is specified; without a suffix, it defaults to specifying a size in bytes. A suffix is case insensitive, except for Mb and Min; these both start with the same letter, and the implementation attaches meaning to the letter case when only the first letter is present. Therefore, use the following suffixes to qualify the integer:

Bytes for bytes

Kb for KiB, 2^{10} numbers of bytes

Mb for MiB, 2^{20} numbers of bytes

Gb for GiB, 2^{30} numbers of bytes

Tb for TiB, 2^{40} numbers of bytes

Sec for seconds

Min for minutes

Hr for hours

Day for days

Wk for weeks

MAX_NUM_<SUBSYS>_LOG An integer that controls the maximum number of rotations a log file is allowed to perform before the oldest one will be rotated away. Thus, at most MAX_NUM_<SUBSYS>_LOG + 1 log files of the same program coexist at a given time. The default value is 1.

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>_LOG_KEEP_OPEN A boolean value that controls whether or not the log file is kept open between writes. When True, the daemon will not open and close the log file between writes. Instead the daemon will hold the log file open until the log needs to be rotated. When False, the daemon reverts to the previous behavior of opening and closing the log file between writes. When the $(<SUBSYS>_LOCK) macro is defined, setting $(<SUBSYS>_LOG_KEEP_OPEN) has no effect, as the daemon will unconditionally revert back to the open/close between writes behavior. On Windows platforms, the value defaults to True for all daemons. On Linux platforms, the value defaults to True for all daemons, except the *condor_shadow*, due to a global file descriptor limit.

<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.

JOB_QUEUE_LOG A full path and file name, specifying the job queue log. The default value, when not defined is `$(SPOOL)/job_queue.log`. This specification can be useful, if there is a solid state drive which is big enough to hold the frequently written to `job_queue.log`, but not big enough to hold the whole contents of the spool directory.

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.

LOCK_DEBUG_LOG_TO_APPEND A boolean value that defaults to `False`. This variable controls whether a daemon's debug lock is used when appending to the log. When `False`, the debug lock is only used when rotating the log file. This is more efficient, especially when many processes share the same log file. When `True`, the debug lock is used when writing to the log, as well as when rotating the log file. This setting is ignored under Windows, and the behavior of Windows platforms is as though this variable were `True`. Under Unix, the default value of `False` is appropriate when logging to file systems that support the POSIX semantics of `O_APPEND`. On non-POSIX-compliant file systems, it is possible for the characters in log messages from multiple processes sharing the same log to be interleaved, unless locking is used. Since HTCondor does not support sharing of debug logs between processes running on different machines, many non-POSIX-compliant file systems will still avoid interleaved messages without requiring HTCondor to use a lock. Tests of AFS and NFS have not revealed any problems when appending to the log without locking.

ENABLE_USERLOG_LOCKING A boolean value that defaults to `False` on Unix platforms and `True` on Windows platforms. When `True`, a user's job event log will be locked before being written to. If `False`, HTCondor will not lock the file before writing.

ENABLE_USERLOG_FSYNC A boolean value that is `True` by default. When `True`, writes to the user's job event log are sync-ed to disk before releasing the lock.

USERLOG_FILE_CACHE_MAX The integer number of job event log files that the *condor_schedd* will keep open for writing during an interval of time (specified by `USERLOG_FILE_CACHE_CLEAR_INTERVAL`). The default value is 0, causing no files to remain open; when 0, each job event log is opened, the event is written, and then the file is closed. Individual file descriptors are removed from this count when the *condor_schedd* detects that no jobs are currently using them. Opening a file is a relatively time consuming operation on a networked file system (NFS), and therefore, allowing a set of files to remain open can improve performance. The value of this variable needs to be set low enough such that the *condor_schedd* daemon process does not run out of file descriptors by leaving these job event log files open. The Linux operating system defaults to permitting 1024 assigned file descriptors per process; the *condor_schedd* will have one file descriptor per running job for the *condor_shadow*.

USERLOG_FILE_CACHE_CLEAR_INTERVAL The integer number of seconds that forms the time interval within which job event logs will be permitted to remain open when `USERLOG_FILE_CACHE_MAX` is greater than zero. The default is 60 seconds. When the interval has passed, all job event logs that the *condor_schedd* has permitted to stay open will be closed, and the interval within which job event logs may remain open between writes of events begins anew. This time interval may be set to a longer duration if the administrator determines that the *condor_schedd* will not exceed the maximum number of file descriptors; a longer interval may yield higher performance due to fewer files being opened and closed.

EVENT_LOG_COUNT_EVENTS A boolean value that is `False` by default. When `True`, upon rotation of the user's job event log, a count of the number of job events is taken by scanning the log, such that the newly created, post-rotation user job event log will have this count in its header. This configuration variable is relevant when rotation of the user's job event log is enabled.

CREATE_LOCKS_ON_LOCAL_DISK A boolean value utilized only for Unix operating systems, that defaults to `True`. This variable is only relevant if `ENABLE_USERLOG_LOCKING` is `True`. When `True`, lock files are written to a directory named `condorLocks`, thereby using a local drive to avoid known problems with locking on NFS. The location of the `condorLocks` directory is determined by

1. The value of `TEMP_DIR`, if defined.
2. The value of `TMP_DIR`, if defined and `TEMP_DIR` is not defined.
3. The default value of `/tmp`, if neither `TEMP_DIR` nor `TMP_DIR` is defined.

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.

DEBUG_TIME_FORMAT This string defines how to format the current time printed at the start of each line in the daemon log files. The value is a format string is passed to the C `strftime()` function, so see that manual page for platform-specific details. If not defined, the default value is

```
"%m/%d/%y %H:%M:%S"
```

<SUBSYS>_DEBUG All of the HTCondor 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 3.8.13 on UIDs in HTCondor 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 HTCondor'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 HTCondor 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 HTCondor 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 HTCondor to work because of problems with DNS, NIS or other host name resolving systems in use.

D_CKPT When this flag is set, the HTCondor 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 3.8.1 for more information about secure communication configuration.

D_PROCFAMILY HTCondor 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 3.6).

D_PROTOCOL Enable debug messages relating to the protocol for HTCondor's matchmaking and resource claiming framework.

D_STATS Enable debug messages relating to the TCP statistics for file transfers. Note that the shadow and starter, by default, log these statistics to special log files (see SHADOW_STATS_LOG section 3.5.10 and STARTER_STATS_LOG, section 3.5.11). Note that, as of version 8.5.6, C_GAHP_DEBUG defaults to D_STATS.

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, HTCondor will always print out the process identifier (PID) of the process writing each line to the log file. This is especially helpful for HTCondor 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, HTCondor 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 HTCondor's use of system file descriptors as it will generally track the number of file descriptors that HTCondor has open.

D_CATEGORY 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_CATEGORY is set, Condor will include the debugging level flags that were in effect for each line of output. This may be used to filter log output by the level or tag it, for example, identifying all logging output at level D_SECURITY, or D_ACCOUNTANT.

D_TIMESTAMP 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_TIMESTAMP is set, the time at the beginning of each line in the log file with be a number of seconds since the start of the Unix era. This form of timestamp can be more convenient for tools to process.

D_SUB_SECOND 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_SUB_SECOND is set, the time at the beginning of each line in the log file will contain a fractional part to the seconds field that is accurate to the millisecond.

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 HTCondor tools.

Log files may optionally be specified per debug level as follows:

<SUBSYS>_<LEVEL>_LOG The name of a log file for messages at a specific debug level for a specific subsystem. <LEVEL> is defined by any debug level, but without the D_ prefix. See section 3.5.2 for the list of debug levels. If the debug level is included in $(<SUBSYS>_DEBUG), then all messages of this debug level will be written both to the log file defined by <SUBSYS>_LOG and the the log file defined by <SUBSYS>_<LEVEL>_LOG. As examples, SHADOW_SYSCALLS_LOG specifies a log file for all remote system call debug messages, and

NEGOTIATOR_MATCH_LOG specifies a log file that only captures *condor_negotiator* debug events occurring with matches.

MAX_<SUBSYS>_<LEVEL>_LOG See section 3.5.2, the definition of 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 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 MiB. 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 effect.

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 effect.

EVENT_LOG_ROTATION_LOCK Specifies 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; its default value is $(LOCK)/EventLogLock. If an empty value is set, then the file that is used is the file path of the event log itself, with the string .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 configuration variable has no effect.

EVENT_LOG_FSYNC A boolean value that controls whether HTCondor 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 False on Unix platforms and True on Windows platforms. When True, the event log (as specified by EVENT_LOG) will be locked before being written to. When False, HTCondor 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 effect.

EVENT_LOG_JOB_AD_INFORMATION_ATTRS A comma separated list of job ClassAd attributes, whose evaluated values form a new event, the JobAdInformationEvent, given Event Number 028. This new event is placed in the event log in addition to each logged event. If EVENT_LOG is not defined, this configuration variable has no effect. This configuration variable is the same as the job ClassAd attribute JobAdInformationAttrs (see page 995), but it applies to the system Event Log rather than the user job log.

3.5.3 DaemonCore Configuration File Entries

Please read section 3.11 for details on DaemonCore. There are certain configuration file settings that DaemonCore uses which affect all HTCondor 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 HTCondor's host-based security. See section 3.8.9 on Setting up IP/host-based security in HTCondor 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, and which only affect the in-memory configuration variables. Because of the potential security implications of this feature, by default, HTCondor daemons will not honor these requests. To use this functionality, HTCondor 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. 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 HTCondor configuration files. Because of the potential security implications of this feature, by default, HTCondor daemons will not honor these requests. To use this functionality, HTCondor administrators must specifically enable it by setting ENABLE_PERSISTENT_CONFIG to True, creating a directory where the HTCondor 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. 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 HTCondor 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 HTCondor installations, though it can be shared by all HTCondor 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 HTCondor 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_<PERMISSION-LEVEL> 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 3.8.9 on Setting up IP/Host-Based Security in HTCondor for details on these macros and how to configure them. In particular, section 3.8.9 on page 421 contains details specific to these macros.

SHUTDOWN_GRACEFUL_TIMEOUT Determines how long HTCondor 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 HTCondor 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 3.3.12.

<SUBSYS>_SUPER_ADDRESS_FILE A complete path to a file that is to contain an IP address and port number for a command port that is serviced with priority for a daemon. Every HTCondor daemon that uses DaemonCore may have a higher priority command port where commands are sent. Any command that goes through *condor_sos*, and any command issued by the super user (root or local system) for a daemon on the local machine will have the command sent to this port. Default values are provided for the *condor_schedd* daemon at `$(SPOOL)/.schedd_address.super` and the *condor_collector* daemon at `$(LOG)/.collector_address.super`. When not defined for other DaemonCore daemons, there will be no higher priority command port.

<SUBSYS>_DAEMON_AD_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*.

<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 3.3.12.

`<SUBSYS>_EXPRS` is a historic setting that functions identically to `<SUBSYS>_ATTRS`. It may be removed in the future, so 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_ATTRS` not part of the `<SUBSYS>_ATTRS`, it is documented in section 3.5.12

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 = time() >= $(NUMBER) || $(BOOL1)
MY_STRING = "$(STRING)"
STARTD_ATTRS = MY_STRING, NUMBER, BOOL1, BOOL2
```

DAEMON_SHUTDOWN Starting with HTCondor 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.

NOTE: This functionality does not work in conjunction with HTCondor's high-availability support (see section 3.13 on page 456 for more information). If you enable high-availability for a particular daemon, you should not define this expression.

DAEMON_SHUTDOWN_FAST 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 A boolean value that controls how an HTCondor daemon creates a new process on Linux platforms. If set to the default value of `True`, 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, for example, a *condor_schedd* with a lot of jobs in the queue. Currently, the use of `clone` is available on Linux systems. If HTCondor detects that it is running under the *valgrind* analysis tools, this setting is ignored and treated as `False`, to work around incompatibilities.

MAX_TIME_SKIP When an HTCondor daemon notices the system clock skip forwards or backwards more than the number of seconds specified by this parameter, it may take special action. For instance, the *condor_master* will restart HTCondor in the event of a clock skip. Defaults to a value of 1200, which in effect means that HTCondor will restart if the system clock jumps by more than 20 minutes.

NOT_RESPONDING_TIMEOUT When an HTCondor daemon's parent process is another HTCondor 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 value with a default value of `False`. This parameter is for debugging purposes on Unix systems, and it controls the behavior of the parent process when the parent process determines that a child process is not responding. 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 the

configuration variable CREATE_CORE_FILES enabled, the child process will then generate a core dump. See NOT_RESPONDING_TIMEOUT on page 227, and CREATE_CORE_FILES on page 213 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.

SOCKET_LISTEN_BACKLOG An integer value that defaults to 500, which defines the backlog value for the `listen()` network call when a daemon creates a socket for incoming connections. It limits the number of new incoming network connections the operating system will accept for a daemon that the daemon has not yet serviced.

MAX_ACCEPTS_PER_CYCLE An integer value that defaults to 8. It is a rarely changed performance tuning parameter to limit the number of accepts of new, incoming, socket connect requests per DaemonCore event cycle. A value of zero or less means no limit. It has the most noticeable effect on the *condor_schedd*, and would be given a higher integer value for tuning purposes when there is a high number of jobs starting and exiting per second.

MAX_REAPS_PER_CYCLE An integer value that defaults to 0. It is a rarely changed performance tuning parameter that places a limit on the number of child process exits to process per DaemonCore event cycle. A value of zero or less means no limit.

CORE_FILE_NAME Defines the name of the core file created on Windows platforms. Defaults to `core.$(SUBSYSTEM).WIN32`.

PIPE_BUFFER_MAX The maximum number of bytes read from a `stdout` or `stdout` pipe. The default value is 10240. A rare example in which the value would need to increase from its default value is when a hook must output an entire ClassAd, and the ClassAd may be larger than the default.

3.5.4 Network-Related Configuration File Entries

More information about networking in HTCondor can be found in section 3.9 on page 431.

BIND_ALL_INTERFACES For systems with multiple network interfaces, if this configuration setting is `False`, HTCondor will only bind network sockets to the IP address specified with NETWORK_INTERFACE (described below). If set to `True`, the default value, HTCondor will listen on all interfaces. However, currently HTCondor 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 3.9.3 on page 437.

CCB_ADDRESS This is the address of a *condor_collector* that will serve as this daemon's HTCondor Connection Broker (CCB). Multiple addresses may be listed (separated by commas and/or spaces) for redundancy. The CCB server must authorize this daemon at DAEMON level for this configuration to succeed. It is highly recommended to also configure PRIVATE_NETWORK_NAME if you configure CCB_ADDRESS so communications originating within the same private network do not need to go through CCB. For more information about CCB, see page 440.

CCB_HEARTBEAT_INTERVAL This is the maximum number of seconds of silence on a daemon's connection to the CCB server after which it will ping the server to verify that the connection still works. The default is 5 minutes. This feature serves to both speed up detection of dead connections and to generate a guaranteed minimum frequency of activity to attempt to prevent the connection from being dropped. The special value 0 disables the heartbeat. The heartbeat is automatically disabled if the CCB server is older than HTCondor version 7.5.0. Having the heartbeat interval greater than the job ClassAd attribute `JobLeaseDuration` may cause unnecessary job disconnects in pools with network issues.

CCB_POLLING_INTERVAL In seconds, the smallest amount of time that could go by before CCB would begin another round of polling to check on already connected clients. While the value of this variable does not change, the actual interval used may be exceeded if the measured amount of time previously taken to poll to check on already connected clients exceeded the amount of time desired, as expressed with `CCB_POLLING_TIMESLICE`. The default value is 20 seconds.

CCB_POLLING_MAX_INTERVAL In seconds, the interval of time after which polling to check on already connected clients must occur, independent of any other factors. The default value is 600 seconds.

CCB_POLLING_TIMESLICE A floating point fraction representing the fractional amount of the total run time of CCB to set as a target for the maximum amount of CCB running time used on polling to check on already connected clients. The default value is 0.05.

CCB_READ_BUFFER The size of the kernel TCP read buffer in bytes for all sockets used by CCB. The default value is 2 KiB.

CCB_WRITE_BUFFER The size of the kernel TCP write buffer in bytes for all sockets used by CCB. The default value is 2 KiB.

CCB_SWEEP_INTERVAL The interval, in seconds, between times when the CCB server writes its information about open TCP connections to a file. Crash recovery is accomplished using the information. The default value is 1200 seconds (20 minutes).

CCB_RECONNECT_FILE The full path and file name of the file that the CCB server writes its information about open TCP connections to a file. Crash recovery is accomplished using the information. The default value is `$(SPOOL)/.ccb_reconnect`.

COLLECTOR_USES_SHARED_PORT A boolean value that specifies whether the *condor_collector* uses the *condor_shared_port* daemon. When true, the *condor_shared_port* will transparently proxy queries to the *condor_collector* so users do not need to be aware of the presence of the *condor_shared_port* when querying the collector and configuring other daemons. The default is `True`

SHARED_PORT_DEFAULT_ID When `COLLECTOR_USES_SHARED_PORT` is set to `True`, this is the shared port ID used by the *condor_collector*. This defaults to `collector` and will not need to be changed by most sites.

AUTO_INCLUDE_SHARED_PORT_IN_DAEMON_LIST A boolean value that specifies whether `SHARED_PORT` should be automatically inserted into *condor_master*'s `DAEMON_LIST` when `USE_SHARED_PORT` is `True`. The default for this setting is `True`.

<SUBSYS>_MAX_FILE_DESCRIPTORS This setting is identical to `MAX_FILE_DESCRIPTORS`, but it only applies to a specific subsystem. If the subsystem-specific setting is unspecified, `MAX_FILE_DESCRIPTORS` is used. For the *condor_collector* daemon, the value defaults to 10240, and for the *condor_schedd* daemon, the

value defaults to 4096. If the *condor_shared_port* daemon is in use, its value for this parameter should match the largest value set for the other daemons.

MAX_FILE_DESCRIPTORS Under Unix, this specifies the maximum number of file descriptors to allow the HTCondor daemon to use. File descriptors are a system resource used for open files and for network connections. HTCondor daemons that make many simultaneous network connections may require an increased number of file descriptors. For example, see page 440 for information on file descriptor requirements of CCB. Changes to this configuration variable require a restart of HTCondor in order to take effect. Also note that only if HTCondor is running as root will it be able to increase the limit above the hard limit (on maximum open files) that it inherits.

NETWORK_HOSTNAME The host name to use, overriding the value returned by `gethostname()`, which will be invoked by default to query the operating system to obtain the host name of the local machine. Among other things, the host name is used to identify daemons in an HTCondor pool, via the `Machine` and `Name` attributes of daemon ClassAds. This variable can be used when a machine has multiple network interfaces with different host names, to use a host name that is not the primary one.

NETWORK_INTERFACE An IP address of the form `123.123.123.123` or the name of a network device, as in the example `eth0`. The wild card character (`*`) may be used within either. For example, `123.123.*` would match a network interface with an IP address of `123.123.123.123` or `123.123.100.100`. The default value is `*`, which matches all network interfaces.

The effect of this variable depends on the value of `BIND_ALL_INTERFACES`. There are two cases:

If `BIND_ALL_INTERFACES` is `True` (the default), `NETWORK_INTERFACE` controls what IP address will be advertised as the public address of the daemon. If multiple network interfaces match the value and `ENABLE_ADDRESS_REWRITING` is `True` (the default), the IP address that is chosen to be advertised will be the one that is used to communicate with the *condor_collector*. If `ENABLE_ADDRESS_REWRITING` is `False`, the IP address that is chosen to be advertised will be the one associated with the first device (in system-defined order) that is in a public address space, or a private address space, or a loopback address, in that order of preference. If it is desired to advertise an IP address that is not associated with any local network interface, for example, when TCP forwarding is being used, then `TCP_FORWARDING_HOST` should be used instead of `NETWORK_INTERFACE`.

If `BIND_ALL_INTERFACES` is `False`, then `NETWORK_INTERFACE` specifies which IP address HTCondor should use for all incoming and outgoing communication. If more than one IP address matches the value, then the IP address that is chosen will be the one associated with the first device (in system-defined order) that is in a public address space, or a private address space, or a loopback address, in that order of preference.

More information about configuring HTCondor on machines with multiple network interfaces can be found in section 3.9.3 on page 437.

PRIVATE_NETWORK_NAME If two HTCondor 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 CCB 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 HTCondor daemon or tool is configured with the same `PRIVATE_NETWORK_NAME`, it will attempt to contact this daemon using its private network address. Even for sites using CCB, this is an important optimization, since it means that two daemons on the same network can communicate directly, without having to go through the broker. If CCB is enabled, and the `PRIVATE_NETWORK_NAME` is defined, the daemon's private address will be defined automatically. Otherwise, you can specify a particular private

IP address to use by defining the PRIVATE_NETWORK_INTERFACE setting (described below). The default is $(FULL_HOSTNAME). After changing this setting and running *condor_reconfig*, it may take up to one *condor_collector* update interval before the change becomes visible.

PRIVATE_NETWORK_INTERFACE For systems with multiple network interfaces, if this configuration setting and PRIVATE_NETWORK_NAME are both defined, HTCondor daemons will advertise some additional attributes in their ClassAds to help other HTCondor daemons and tools in the same private network to communicate directly.

PRIVATE_NETWORK_INTERFACE defines what IP address of the form 123.123.123.123 or name of a network device (as in the example eth0) a given multi-homed machine should use for the private network. The asterisk (*) may be used as a wild card character within either the IP address or the device name. If another HTCondor daemon or tool is configured with the same PRIVATE_NETWORK_NAME, it will attempt to contact this daemon using the IP address specified here. The syntax for specifying an IP address is identical to NETWORK_INTERFACE. Sites using CCB only need to define the PRIVATE_NETWORK_NAME, and the PRIVATE_NETWORK_INTERFACE will be defined automatically. Unless CCB is enabled, there is no default value for this variable. After changing this variable and running *condor_reconfig*, it may take up to one *condor_collector* update interval before the change becomes visible.

TCP_FORWARDING_HOST This specifies the host or IP address that should be used as the public address of this daemon. If a host name is specified, be aware that it will be resolved to an IP address by this daemon, not by the clients wishing to connect to it. It is the IP address that is advertised, not the host name. This setting is useful if HTCondor on this host may be reached through a NAT or firewall by connecting to an IP address that forwards connections to this host. It is assumed that the port number on the TCP_FORWARDING_HOST that forwards to this host is the same port number assigned to HTCondor on this host. This option could also be used when ssh port forwarding is being used. In this case, the incoming addresses of connections to this daemon will appear as though they are coming from the forwarding host rather than from the real remote host, so any authorization settings that rely on host addresses should be considered accordingly.

ENABLE_ADDRESS_REWRITING A boolean value that defaults to True. When NETWORK_INTERFACE matches only one IP address or TCP_FORWARDING_HOST is defined or NET_REMAP_ENABLE is True, this setting has no effect and the behavior is as though it had been set to False. When True, IP addresses published by HTCondor daemons are automatically rewritten to match the IP address of the network interface used to make the publication. For example, if the *condor_schedd* advertises itself to two pools via flocking, and the *condor_collector* for one pool is reached by the *condor_schedd* through a private network interface, while the *condor_collector* for the other pool is reached through a different network interface, the IP address published by the *condor_schedd* daemon will match the address of the respective network interfaces used in the two cases. The intention is to make it easier for HTCondor daemons to operate in a multi-homed environment.

HIGHPORT Specifies an upper limit of given port numbers for HTCondor to use, such that HTCondor is restricted to a range of port numbers. If this macro is not explicitly specified, then HTCondor will not restrict the port numbers that it uses. HTCondor 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 HTCondor to use, such that HTCondor is restricted to a range of port numbers. If this macro is not explicitly specified, then HTCondor will not restrict the port numbers that it uses. HTCondor 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 HTCondor to use on incoming connections (ports for listening), such that HTCondor 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 HTCondor to use on incoming connections (ports for listening), such that HTCondor 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 HTCondor to use on outgoing connections, such that HTCondor 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 HTCondor to use on outgoing connections, such that HTCondor 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 This boolean value controls whether TCP or UDP is used by daemons to send ClassAd updates to the *condor_collector*. Please read section 3.9.5 for more details and a discussion of when this functionality is needed. When using TCP in large pools, it is also necessary to ensure that the *condor_collector* has a large enough file descriptor limit using COLLECTOR_MAX_FILE_DESCRIPTORS. The default value is True.

UPDATE_VIEW_COLLECTOR_WITH_TCP This boolean value controls whether TCP or UDP is used by the *condor_collector* to forward ClassAd updates to the *condor_collector* daemons specified by CONDOR_VIEW_HOST. Please read section 3.9.5 for more details and a discussion of when this functionality is needed. The default value is False.

TCP_UPDATE_COLLECTORS The list of *condor_collector* daemons which will be updated with TCP instead of UDP when UPDATE_COLLECTOR_WITH_TCP or UPDATE_VIEW_COLLECTOR_WITH_TCP is False. Please read section 3.9.5 for more details and a discussion of when a site needs this functionality.

<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.

UDP_NETWORK_FRAGMENT_SIZE An integer value that defaults to 1000 and represents the maximum size in bytes of an outgoing UDP packet. If the outgoing message is larger than `$(UDP_NETWORK_FRAGMENT_SIZE)`, then the message will be split (fragmented) into multiple packets no larger than `$(UDP_NETWORK_FRAGMENT_SIZE)`. If the destination of the message is the loopback network interface, see `UDP_LOOPBACK_FRAGMENT_SIZE` below. For instance, the maximum payload size of a UDP packet over Ethernet is typically 1472 bytes, and thus if a UDP payload exceeds 1472 bytes the IP network stack on either hosts or forwarding devices (such as network routers) will have to perform message fragmentation on transmission and reassembly on receipt. Experimentation has shown that such devices are more likely to simply drop a UDP message under high-traffic scenarios if the message requires reassembly. HTCondor avoids this situation via the capability to perform UDP fragmentation and reassembly on its own.

UDP_LOOPBACK_FRAGMENT_SIZE An integer value that defaults to 60000 and represents the maximum size in bytes of an outgoing UDP packet that is being sent to the loopback network interface (e.g. 127.0.0.1). If the outgoing message is larger than `$(UDP_LOOPBACK_FRAGMENT_SIZE)`, then the message will be split (fragmented) into multiple packets no larger than `$(UDP_LOOPBACK_FRAGMENT_SIZE)`. If the destination of the message is not the loopback interface, see `UDP_NETWORK_FRAGMENT_SIZE` above.

ALWAYS_REUSEADDR A boolean value that, when `True`, tells HTCondor to set `SO_REUSEADDR` socket option, so that the schedd can run large numbers of very short jobs without exhausting the number of local ports needed for shadows. The default value is `True`. (Note that this represents a change in behavior compared to versions of HTCondor older than 8.6.0, which did not include this configuration macro. To restore the previous behavior, set this value to `False`.)

3.5.5 Shared File System Configuration File Macros

These macros control how HTCondor interacts with various shared and network file systems. If you are using AFS as your shared file system, be sure to read section 3.14.1 on Using HTCondor with AFS. For information on submitting jobs under shared file systems, see section 2.5.8.

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 HTCondor 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 HTCondor 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`. HTCondor 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, HTCondor will run the job as user `nobody`. For example, a job submission to the HTCondor pool at the UW Madison from flippy.example.com, claiming a `UID_DOMAIN` of 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 HTCondor 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 HTCondor 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 HTCondor sends e-mail about a job, HTCondor 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 HTCondor 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, HTCondor must not do this regular security check. If the TRUST_UID_DOMAIN setting is defined to True, HTCondor 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 HTCondor 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 HTCondor not to do this check, set this configuration variable to True. HTCondor will then run the job under the user's UID.

SLOT<N>_USER The name of a user for HTCondor 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. <N> is an integer associated with slots. On Windows, SLOT<N>_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. See Section 3.8.13 for more information.

STARTER_ALLOW_RUNAS_OWNER 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 SLOT<N>_USER or nobody (False). On Unix, this defaults to True. On Windows, it defaults to False. The job ClassAd may also contain the attribute RunAsOwner which is logically ANDed with the *condor_starter* daemon's boolean value. Under Unix, if the job does not specify it, this attribute defaults to True. Under Windows, the attribute 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 3.8.13 for more information.

DEDICATED_EXECUTE_ACCOUNT_REGEXP 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 SLOT<N>_USER to be a *different* account for each HTCondor slot, and no non-condor processes will ever be run by these accounts, then this pattern should match the names of all SLOT<N>_USER accounts. Jobs run under a dedicated execute account are reliably tracked by HTCondor, whereas other jobs, may spawn processes that HTCondor 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_REGEXP = 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"
```

EXECUTE_LOGIN_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_REGEXP instead.

A boolean value that defaults to False. When True, HTCondor knows that all jobs are being run by dedicated execution accounts (whether they are running as the job owner or as nobody or as SLOT<N>_USER). Therefore, when the job exits, all processes running under the same account will be killed.

FILESYSTEM_DOMAIN An arbitrary string that is used to decide if the two machines, a submit machine and an execute machine, share a file system. Although this configuration variable name contains the word "DOMAIN", its value 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. HTCondor currently has no way to express this automatically. A job can express the need to use a particular file system where machines advertise an additional ClassAd attribute and the job requires machines with the attribute, as described on the question within the https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToAdminRecipes page for how to run jobs on a subset of machines that have required software installed.

Note that if you do not set $(FILESYSTEM_DOMAIN), the value defaults to the fully qualified host name of the local machine. Since each machine will have a different $(FILESYSTEM_DOMAIN), they will not be considered to have shared file systems.

RESERVE_AFS_CACHE If your machine is running AFS and the AFS cache lives on the same partition as the other HTCondor directories, and you want HTCondor 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 HTCondor jobs running in the standard universe access their files. By default, HTCondor will redirect the file I/O requests of standard universe jobs from the executing machine to the

submitting machine. So, as an HTCondor 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 submitting machine, which in turn sends all the I/O back over the network to the NFS file server. Thus, all of the program's I/O is being sent over the network twice.

If this configuration variable is `True`, then HTCondor will attempt to read/write files directly on the executing machine 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 HTCondor 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 as 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 HTCondor jobs are most often heavy on CPU and light on I/O. It also ensures that a remote standard universe HTCondor job will always use HTCondor'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 `$(FILESYSTEM_DOMAIN)` claimed by the submit machine is indeed found to be a subset of what an inverse look up to a DNS (domain name server) reports as the fully qualified domain name for the submit machine's IP address (this security measure safeguards against the submit machine from lying), *then* the job will access files using a local system call, without redirecting them to the submitting machine (with NFS). Otherwise, the system call will get routed back to the submitting machine using HTCondor's remote system call mechanism. NOTE: When submitting a vanilla job, *condor_submit* will, by default, append requirements to the Job ClassAd that specify the machine to run the job must be in the same `$(FILESYSTEM_DOMAIN)` and the same `$(UID_DOMAIN)`.

This configuration variable similarly changes the semantics of Chirp file I/O when running in the vanilla, java or parallel universe. If this variable is set in those universes, Chirp will not send I/O requests over the network as requested, but perform them directly to the locally mounted file system. Other than Chirp file access, this variable is unused outside of the standard universe.

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 HTCondor 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 HTCondor 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 an HTCondor 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 as 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 HTCondor jobs are usually heavy on CPU and light on I/O. `False` ensures that a remote standard universe HTCondor job will always see the exact same file system that the user on 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 HTCondor job being there.

However, things may be different at your site, which is why the setting is there.

3.5.6 Checkpoint Server Configuration File Macros

These macros control whether or not HTCondor uses a checkpoint server. This section describes the settings that the checkpoint server itself needs defined. See section 3.10 on Installing a Checkpoint Server for details on installing and running a checkpoint server.

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 full path of the directory the checkpoint server should use to store checkpoint files. Depending on the size of the pool and the size of the jobs submitted, this directory and its subdirectories might need to store many MiB of data.

USE_CKPT_SERVER A boolean which determines if a given submit machine is to use a checkpoint server if one is available. If a checkpoint server is not available or the variable `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 *condor_shadow* daemon 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 variable 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 (usually 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.

CKPT_SERVER_INTERVAL The maximum number of seconds the checkpoint server waits for activity on network sockets before performing other tasks. The default value is 300 seconds.

CKPT_SERVER_CLASSAD_FILE A string that represents a file in the file system to which ClassAds will be written. The ClassAds denote information about stored checkpoint files, such as owner, shadow IP address, name of the file, and size of the file. This information is also independently recorded in the `TransferLog`. The default setting is undefined, which means a checkpoint server ClassAd file will not be kept.

CKPT_SERVER_CLEAN_INTERVAL The number of seconds that must pass until the ClassAd log file as described by the CKPT_SERVER_CLASSAD_FILE variable gets truncated. The default is 86400 seconds, which is one day.

CKPT_SERVER_REMOVE_STALE_CKPT_INTERVAL The number of seconds between attempts to discover and remove stale checkpoint files. It defaults to 86400 seconds, which is one day.

CKPT_SERVER_SOCKET_BUFSIZE The number of bytes representing the size of the TCP send/recv buffer on the socket file descriptor related to moving the checkpoint file to and from the checkpoint server. The default value is 0, which allows the operating system to decide the size.

CKPT_SERVER_MAX_PROCESSES The maximum number of child processes that could be working on behalf of the checkpoint server. This includes store processes and restore processes. The default value is 50.

CKPT_SERVER_MAX_STORE_PROCESSES The maximum number of child process strictly devoted to the storage of checkpoints. The default is the value of CKPT_SERVER_MAX_PROCESSES.

CKPT_SERVER_MAX_RESTORE_PROCESSES The maximum number of child process strictly devoted to the restoring of checkpoints. The default is the value of CKPT_SERVER_MAX_PROCESSES.

CKPT_SERVER_STALE_CKPT_AGE_CUTOFF The number of seconds after which if a checkpoint file has not been accessed, it is considered stale. The default value is 5184000 seconds, which is sixty days.

ALWAYS_USE_LOCAL_CKPT_SERVER A boolean value that defaults to False. When True, it forces all checkpoints to be read from a checkpoint server running on the same machine where the job is running. This is intended to be used when all checkpoint servers access a shared file system.

3.5.7 condor_master Configuration File Macros

These macros control the *condor_master*.

DAEMON_LIST 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 3.3.12). 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_DAEMON_LIST A list delimited by commas and/or spaces that lists the daemons in DAEMON_LIST which use the HTCondor DaemonCore library. The *condor_master* must differentiate between daemons that use DaemonCore and those that do not, so it uses the appropriate inter-process communication mechanisms. This list currently includes all HTCondor daemons except the checkpoint server by default.

As of HTCondor version 7.2.1, a daemon may be appended to the default DC_DAEMON_LIST value by placing the plus character (+) before the first entry in the DC_DAEMON_LIST definition. For example:

```
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 3.3.12.

<DaemonName>_ENVIRONMENT <DaemonName> is the name of a daemon listed in `DAEMON_LIST`. Defines changes to the environment that the daemon is invoked with. It should use the same syntax for specifying the environment as the environment specification in a submit description file. For example, to redefine the `TMP` and `CONDOR_CONFIG` environment variables seen by the *condor_schedd*, place the following in the configuration:

```
SCHEDD_ENVIRONMENT = "TMP=/new/value CONDOR_CONFIG=/special/config"
```

When the *condor_schedd* daemon is 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 899), 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 3.3.12.

<SUBSYS>_USERID The account name that should be used to run the `SUBSYS` process spawned by the *condor_master*. When not defined, the process is spawned as the same user that is running *condor_master*. When defined, the real user id of the spawned process will be set to the specified account, so if this account is not `root`, the process will not have `root` privileges. The *condor_master* must be running as root in order to start processes as other users. Example configuration:

```
COLLECTOR_USERID = condor
NEGOTIATOR_USERID = condor
```

The above example runs the *condor_collector* and *condor_negotiator* as the `condor` user with no `root` privileges. If we specified some account other than the `condor` user, as set by the (`CONDOR_IDS`) configuration variable, then we would need to configure the log files for these daemons to be in a directory that they can write to. When using GSI security or any other security method in which the daemon credential is owned by `root`, it is also necessary to make a copy of the credential, make it be owned by the account the daemons are using, and configure the daemons to use that copy.

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 HTCondor. This macro determines where the *condor_master* finds the *condor_preen* binary. If this macro is set to nothing, *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 the *condor_master* will immediately exit upon startup. This appears strange, but perhaps you do not want HTCondor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized set of files – setting START_MASTER to False for those machines would allow this. Note that START_MASTER is an entry you would most likely find in a local configuration file, not a global configuration file. If not defined, START_MASTER defaults to True.

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_RESTART Defines a mode of operation for the restart of the *condor_master*, when it notices that the *condor_master* binary has changed. Valid values are GRACEFUL, PEACEFUL, and NEVER, with a default value of GRACEFUL. On a GRACEFUL restart of the master, child processes are told to exit, but if they do not before a timer expires, then they are killed. On a PEACEFUL restart, child processes are told to exit, after which the *condor_master* waits until they do so.

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).

DEFAULT_MASTER_SHUTDOWN_SCRIPT A full path and file name of a program that the *condor_master* is to execute via the Unix `execl()` call, or the similar Win32 `_execl()` call, instead of the normal call to `exit()`. This allows the admin to specify a program to execute as root when the *condor_master* exits. Note that a successful call to the *condor_set_shutdown* program will override this setting; see the documentation for config knob `MASTER_SHUTDOWN_<Name>` below.

MASTER_SHUTDOWN_<Name> A full path and file name of a program that the *condor_master* is to execute via the Unix `execl()` call, or the similar Win32 `_execl()` 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 effect 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 (that is, just before it exits), the program is then executed as described above. The manual page for *condor_set_shutdown* on page 875 contains details on the use of this program.

NOTE: This program will be run with root privileges under Unix or administrator privileges under Windows. The administrator must ensure that this cannot be used in such a way as to violate system integrity.

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 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 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 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 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 crashes. 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:

1^{st} crash: $n = 0$, so: $t = 9 + 2^0 = 9 + 1 = 10\ seconds$

2^{nd} crash: $n = 1$, so: $t = 9 + 2^1 = 9 + 2 = 11\ seconds$

3^{rd} crash: $n = 2$, so: $t = 9 + 2^2 = 9 + 4 = 13\ seconds$

...

6^{th} crash: $n = 5$, so: $t = 9 + 2^5 = 9 + 32 = 41\ seconds$

...

9^{th} crash: $n = 8$, so: $t = 9 + 2^8 = 9 + 256 = 265\ seconds$

And, after the 13 crashes, it would be:

13^{th} 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, HTCondor 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 HTCondor 7.1.1, when the string included an `@` sign, HTCondor replaced whatever followed the `@` sign with the fully qualified host name of the local machine. HTCondor 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 3.5.9 on page 269). Since HTCondor 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, the `STARTD_NAME` should be set to uniquely identify the *condor_startd* daemons.

If an HTCondor daemon (master, schedd or startd) has been given a unique name, all HTCondor 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 3.5.3 as `<SUBSYS>_ATTRS`.

MASTER_DEBUG This macro is described in section 3.5.2 as `<SUBSYS>_DEBUG`.

MASTER_ADDRESS_FILE This macro is described in section 3.5.3 as `<SUBSYS>_ADDRESS_FILE`.

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_INSTANCE_LOCK Defines the name of a file for the *condor_master* daemon to lock in order to prevent multiple *condor_master*s 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 HTCondor 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 2) that represents the number of times the *condor_master* will retry to add firewall exceptions. When a Windows machine boots up, HTCondor 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 5 seconds) that the *condor_master* will retry to add firewall exceptions. A value of 0 means that HTCondor will retry indefinitely.

USE_PROCESS_GROUPS A boolean value that defaults to `True`. When `False`, HTCondor daemons on Unix machines will *not* create new sessions or process groups. HTCondor uses processes groups to help it track the descendants of processes it creates. This can cause problems when HTCondor is run under another job execution system.

DISCARD_SESSION_KEYRING_ON_STARTUP A boolean value that defaults to `True`. When `True`, the *condor_master* daemon will replace the kernel session keyring it was invoked with with a new keyring named `htcondor`. Various Linux system services, such as OpenAFS and eCryptFS, use the kernel session keyring to hold passwords and authentication tokens. By replacing the keyring on start up, the *condor_master* ensures these keys cannot be unintentionally obtained by user jobs.

ENABLE_KERNEL_TUNING Relevant only to Linux platforms, a boolean value that defaults to `True`. When `True`, the *condor_master* daemon invokes the kernel tuning script specified by configuration variable `LINUX_KERNEL_TUNING_SCRIPT` once as root when the *condor_master* daemon starts up.

KERNEL_TUNING_LOG A string value that defaults to `$(LOG)/KernelTuningLog`. If the kernel tuning script runs, its output will be logged to this file.

LINUX_KERNEL_TUNING_SCRIPT A string value that defaults to `$(LIBEXEC)/linux_kernel_tuning`. This is the script that the *condor_master* runs to tune the kernel when `ENABLE_KERNEL_TUNING` is `True`.

3.5.8 condor_startd Configuration File Macros

NOTE: If you are running HTCondor on a multi-CPU machine, be sure to also read section 3.7.1 on page 378 which describes how to set up and configure HTCondor on multi-core machines.

These settings control general operation of the *condor_startd*. Examples using these configuration macros, as well as further explanation is found in section 3.7 on Configuring The Startd Policy.

START A boolean expression that, when `True`, indicates that the machine is willing to start running an HTCondor 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 HTCondor to suspend running an HTCondor job. The machine may still be claimed, but the job makes no further progress, and HTCondor does not generate a load on the machine.

PREEMPT A boolean expression that, when `True`, causes HTCondor to stop a currently running job once `MAXJOBRETIREMENTTIME` has expired. This expression is not evaluated if `WANT_SUSPEND` is `True`. The default value is `False`, such that preemption is disabled.

WANT_HOLD A boolean expression that defaults to `False`. When `True` and the value of `PREEMPT` becomes `True` and `WANT_SUSPEND` is `False` and `MAXJOBRETIREMENTTIME` has expired, the job is put on hold for the reason (optionally) specified by the variables `WANT_HOLD_REASON` and `WANT_HOLD_SUBCODE`. As usual,

the job owner may specify **periodic_release** and/or **periodic_remove** expressions to react to specific hold states automatically. The attribute `HoldReasonCode` in the job ClassAd is set to the value 21 when `WANT_HOLD` is responsible for putting the job on hold.

Here is an example policy that puts jobs on hold that use too much virtual memory:

```
VIRTUAL_MEMORY_AVAILABLE_MB = (VirtualMemory*0.9)
MEMORY_EXCEEDED = ImageSize/1024 > $(VIRTUAL_MEMORY_AVAILABLE_MB)
PREEMPT = ($(PREEMPT)) || ($(MEMORY_EXCEEDED))
WANT_SUSPEND = ($(WANT_SUSPEND)) && ($(MEMORY_EXCEEDED)) =!= TRUE
WANT_HOLD = ($(MEMORY_EXCEEDED))
WANT_HOLD_REASON = \
   ifThenElse( $(MEMORY_EXCEEDED), \
               "Your job used too much virtual memory.", \
               undefined )
```

WANT_HOLD_REASON An expression that defines a string utilized to set the job ClassAd attribute `HoldReason` when a job is put on hold due to `WANT_HOLD`. If not defined or if the expression evaluates to `Undefined`, a default hold reason is provided.

WANT_HOLD_SUBCODE An expression that defines an integer value utilized to set the job ClassAd attribute `HoldReasonSubCode` when a job is put on hold due to `WANT_HOLD`. If not defined or if the expression evaluates to `Undefined`, the value is set to 0. Note that `HoldReasonCode` is always set to 21.

CONTINUE A boolean expression that, when `True`, causes HTCondor to continue the execution of a suspended job.

KILL A boolean expression that, when `True`, causes HTCondor to immediately stop the execution of a vacating job, without delay. The job is hard-killed, so any attempt by the job to checkpoint or clean up will be aborted. This expression should normally be `False`. When desired, it may be used to abort the graceful shutdown of a job earlier than the limit imposed by `MachineMaxVacateTime`.

PERIODIC_CHECKPOINT A boolean expression that, when `True`, causes HTCondor 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 HTCondor uses to compare potential jobs. A larger value for a specific job ranks that job above others with lower values for `RANK`.

ADVERTISE_PSLOT_ROLLUP_INFORMATION A boolean value that defaults to `True`, causing the *condor_startd* to advertise ClassAd attributes that may be used in partitionable slot preemption. The attributes are

- `ChildAccountingGroup`
- `ChildActivity`
- `ChildCPUs`
- `ChildCurrentRank`
- `ChildEnteredCurrentState`
- `ChildMemory`
- `ChildName`

- ChildRemoteOwner
- ChildRemoteUser
- ChildRetirementTimeRemaining
- ChildState
- PslotRollupInformation

STARTD_PARTITIONABLE_SLOT_ATTRS A list of additional from the above default attributes from dynamic slots that will be rolled up into a list attribute in their parent partitionable slot, prefixed with the name Child.

IS_VALID_CHECKPOINT_PLATFORM A boolean expression that is logically ANDed with the 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)
    )
  )
)
```

CHECKPOINT_PLATFORM A string used to override the automatically-generated machine ClassAd attribute CheckpointPlatform (see section 12), which is used to identify the platform upon which a job previously generated a checkpoint under the standard universe. This restricts the machine matches that may be considered for a job and where the job may resume. Overriding the value may be necessary for architectures that are the same in name, but actually have differences in instruction sets, such as the AVX extensions to the Intel processor.

WANT_SUSPEND A boolean expression that, when True, tells HTCondor to evaluate the SUSPEND expression to decide whether to suspend a running job. When True, the PREEMPT expression is not evaluated. When not explicitly set, the *condor_startd* exits with an error. When explicitly set, but the evaluated value is anything other than True, the value is utilized as if it were False.

WANT_VACATE A boolean expression that, when True, defines that a preempted HTCondor job is to be vacated, instead of killed. This means the job will be soft-killed and given time to checkpoint or clean up. The amount of time given depends on MachineMaxVacateTime and KILL. The default value is True.

ENABLE_VERSIONED_OPSYS A boolean expression that determines whether pre-7.7.2 strings used for the machine ClassAd attribute OpSys are used or not. Defaults to False on Windows platforms, meaning that the newer behavior of setting OpSys = "WINDOWS" and OpSysVer = 601 (for example), while OpSysAndVer = "WINNT61". On platforms *other* than Windows, the default value is True, meaning that the values for OpSys and OpSysAndVer are the same, implementing the pre-7.7.2 behavior.

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`.

STARTD_HISTORY A file name where the *condor_startd* daemon will maintain a job history file in an analogous way to that of the history file defined by the configuration variable `HISTORY`. It will be rotated in the same way, and the same parameters that apply to the `HISTORY` file rotation apply to the *condor_startd* daemon history as well. This can be read with the *condor_history* command by passing the name of the file to the -file option of *condor_history*.

```
condor_history -file `condor_config_val LOG`/startd_history
```

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)`.

KILLING_TIMEOUT The amount of time in seconds that the *condor_startd* should wait after sending a fast shutdown request to *condor_starter* before forcibly killing the job and *condor_starter*. The default value is 30 seconds.

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 3.7.1 on *condor_startd* states, section 3.7.1 on *condor_startd* Activities, and section 3.7.1 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).

UPDATE_OFFSET An integer value representing the number of seconds of delay that the *condor_startd* should wait before sending its initial update, and the first update after a *condor_reconfig* command is sent to the *condor_collector*. The time of all other updates sent after this initial update is determined by `$(UPDATE_INTERVAL)`. Thus, the first update will be sent after `$(UPDATE_OFFSET)` seconds, and the second update will be sent after `$(UPDATE_OFFSET) + $(UPDATE_INTERVAL)`. This is useful when used in conjunction with the `$RANDOM_INTEGER()` macro for large pools, to spread out the updates sent by a large number of *condor_startd* daemons. Defaults to zero. The example configuration

```
startd.UPDATE_INTERVAL = 300
startd.UPDATE_OFFSET   = $RANDOM_INTEGER(0,300)
```

causes the initial update to occur at a random number of seconds falling between 0 and 300, with all further updates occurring at fixed 300 second intervals following the initial update.

MachineMaxVacateTime An integer expression representing the number of seconds the machine is willing to wait for a job that has been soft-killed to gracefully shut down. The default value is 600 seconds (10 minutes). This expression is evaluated when the job starts running. The job may adjust the wait time by setting

JobMaxVacateTime. If the job's setting is less than the machine's, the job's specification is used. If the job's setting is larger than the machine's, the result depends on whether the job has any excess retirement time. If the job has more retirement time left than the machine's maximum vacate time setting, then retirement time will be converted into vacating time, up to the amount of JobMaxVacateTime. The KILL expression may be used to abort the graceful shutdown of the job at any time. At the time when the job is preempted, the WANT_VACATE expression may be used to skip the graceful shutdown of the job.

MAXJOBRETIREMENTTIME When the *condor_startd* wants to evict a job, a job which has run for less than the number of seconds specified by this expression will not be hard-killed. The *condor_startd* will wait for the job to finish or to exceed this amount of time, whichever comes sooner. Time spent in suspension does not count against the job. The default value of 0 (when the configuration variable is not present) means that the job gets no retirement time. If the job vacating policy grants the job X seconds of vacating time, a preempted job will be soft-killed X seconds before the end of its retirement time, so that hard-killing of the job will not happen until the end of the retirement time if the job does not finish shutting down before then. Note that in peaceful shutdown mode of the *condor_startd*, retirement time is treated as though infinite. In graceful shutdown mode, the job will not be preempted until the configured retirement time expires or SHUTDOWN_GRACEFUL_TIMEOUT expires. In fast shutdown mode, retirement time is ignored. See MAXJOBRETIREMENTTIME in section 3.7.1 for further explanation.

By default the *condor_negotiator* will not match jobs to a slot with retirement time remaining. This behavior is controlled by NEGOTIATOR_CONSIDER_EARLY_PREEMPTION.

There is no default value for this configuration variable.

CLAIM_WORKLIFE This expression specifies the number of seconds after which a claim will stop accepting additional jobs. The default is 1200, which is 20 minutes. Once the *condor_negotiator* gives a *condor_schedd* a claim to a slot, the *condor_schedd* will keep running jobs on that slot as long as it has more jobs with matching requirements, and CLAIM_WORKLIFE has not expired, and it is not preempted. Once CLAIM_WORKLIFE expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. When CLAIM_WORKLIFE is -1, this is treated as an infinite claim worklife, so claims may be held indefinitely (as long as they are not preempted and the user does not run out of jobs, 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 267). 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.

CONSOLE_DEVICES 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, modern versions of Linux do not update the access time of device files for USB devices. Thus, these files cannot be be used to determine when the console is in use. Instead, use the *condor_kbdd* daemon, which gets this information by connecting to the X server.

KBDD_BUMP_CHECK_SIZE The number of pixels that the mouse can move in the X and/or Y direction, while still being considered a bump, and not keyboard activity. If the movement is greater than this bump size then the move is not a transient one, and it will register as activity. The default is 16, and units are pixels. Setting the value to 0 effectively disables bump testing.

KBDD_BUMP_CHECK_AFTER_IDLE_TIME The number of seconds of keyboard idle time that will pass before bump testing begins. The default is 15 minutes.

STARTD_JOB_ATTRS 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. By default, the job ClassAd attributes JobUniverse, NiceUser, ExecutableSize and ImageSize are advertised into the machine ClassAd. This setting was formerly called `STARTD_JOB_EXPRS`. The older name is still supported, but support for the older name may be removed in a future version of HTCondor.

STARTD_ATTRS This macro is described in section 3.5.3 as `<SUBSYS>_ATTRS`.

STARTD_DEBUG This macro (and other settings related to debug logging in the *condor_startd*) is described in section 3.5.2 as `<SUBSYS>_DEBUG`.

STARTD_ADDRESS_FILE This macro is described in section 3.5.3 as `<SUBSYS>_ADDRESS_FILE`

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, HTCondor 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 HTCondor 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 HTCondor's automatic computation of the number of CPUs in the machine. Lying in this way can allow multiple HTCondor jobs to run on a single-CPU machine, by having that machine treated like a multi-core machine with multiple CPUs, which could have different HTCondor jobs running on each one. Or, a multi-core 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.

The default value is `$(DETECTED_CPUS)`.

The *condor_startd* only takes note of the value of this configuration variable on start up, therefore it cannot be changed with a simple reconfigure. To change this, restart the *condor_startd* daemon for the change to take effect. The command will be

```
condor_restart -startd
```

MAX_NUM_CPUS An integer value used as a ceiling for the number of CPUs detected by HTCondor 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 configuration variable controls how HTCondor sees hyper-threaded processors. When set to the default value of `True`, it includes virtual CPUs in the default value of `DETECTED_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, HTCondor will automatically detect the amount of physical memory available on your machine. Define `MEMORY` to tell HTCondor how much physical memory (in MB) your machine has, overriding the value HTCondor computes automatically. The actual amount of memory detected by HTCondor is always available in the pre-defined configuration macro `DETECTED_MEMORY`.

RESERVED_MEMORY How much memory would you like reserved from HTCondor? By default, HTCondor considers all the physical memory of your machine as available to be used by HTCondor jobs. If `RESERVED_MEMORY` is defined, HTCondor 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 3.5.7 on page 242 for defaults and composition of valid HTCondor daemon names.

RUNBENCHMARKS A boolean expression that specifies whether 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 once, when the *condor_startd* starts. To disable start up benchmarks, set `RunBenchmarks` to `False`.

DedicatedScheduler A string that identifies the dedicated scheduler this machine is managed by. Section 3.14.8 on page 476 details the use of a dedicated scheduler.

STARTD_NOCLAIM_SHUTDOWN The number of seconds to run without receiving a claim before shutting HTCondor down on this machine. Defaults to unset, which means to never shut down. This is primarily intended to facilitate glidein; use in other situations is not recommended.

STARTD_PUBLISH_WINREG A string containing a semicolon-separated list of Windows registry key names. For each registry key, the contents of the registry key are published in the machine ClassAd. All attribute names are prefixed with `WINREG_`. The remainder of the attribute name is formed in one of two ways. The first way explicitly specifies the name within the list with the syntax

```
STARTD_PUBLISH_WINREG = AttrName1 = KeyName1; AttrName2 = KeyName2
```

The second way of forming the attribute name derives the attribute names from the key names in the list. The derivation uses the last three path elements in the key name and changes each illegal character to an underscore character. Illegal characters are essentially any non-alphanumeric character. In addition, the percent character (`%`) is replaced by the string `Percent`, and the string `/sec` is replaced by the string `_Per_Sec`.

HTCondor expects that the hive identifier, which is the first element in the full path given by a key name, will be the valid abbreviation. Here is a list of abbreviations:

`HKLM` is the abbreviation for `HKEY_LOCAL_MACHINE`

`HKCR` is the abbreviation for `HKEY_CLASSES_ROOT`

`HKCU` is the abbreviation for `HKEY_CURRENT_USER`

`HKPD` is the abbreviation for `HKEY_PERFORMANCE_DATA`

`HKCC` is the abbreviation for `HKEY_CURRENT_CONFIG`

`HKU` is the abbreviation for `HKEY_USERS`

The `HKPD` key names are unusual, as they are not shown in *regedit*. Their values are periodically updated at the interval defined by `UPDATE_INTERVAL`. The others are not updated until *condor_reconfig* is issued.

Here is a complete example of the configuration variable definition,

```
STARTD_PUBLISH_WINREG = HKLM\Software\Perl\BinDir; \
 BATFile_RunAs_Command = HKCR\batFile\shell\RunAs\command; \
 HKPD\Memory\Available MBytes; \
 BytesAvail = HKPD\Memory\Available Bytes; \
 HKPD\Terminal Services\Total Sessions; \
 HKPD\Processor\% Idle Time; \
 HKPD\System\Processes
```

which generates the following portion of a machine ClassAd:

```
WINREG_Software_Perl_BinDir = "C:\Perl\bin\perl.exe"
WINREG_BATFile_RunAs_Command = "%SystemRoot%\System32\cmd.exe /C \"%1\" %*"
WINREG_Memory_Available_MBytes = 5331
WINREG_BytesAvail = 5590536192.000000
WINREG_Terminal_Services_Total_Sessions = 2
WINREG_Processor_Percent_Idle_Time = 72.350384
WINREG_System_Processes = 166
```

MOUNT_UNDER_SCRATCH A ClassAd expression, which when evaluated in the context of the job ClassAd evaluates to a comma separated list of directories. For each directory in the list, HTCondor creates a directory in the job's temporary scratch directory with that name, and makes it available at the given name using bind mounts. This is available on Linux systems which provide bind mounts and per-process tree mount tables, such as Red Hat Enterprise Linux 5. A bind mount is like a symbolic link, but is not globally visible to all processes. It is only visible to the job and the job's child processes. As an example:

```
MOUNT_UNDER_SCRATCH = ifThenElse(TARGET.UtsnameSysname ? "Linux", "/tmp,/var/tmp"
```

If the job is running on a Linux system, it will see the usual `/tmp` and `/var/tmp` directories, but when accessing files via these paths, the system will redirect the access. The resultant files will actually end up in directories named `tmp` or `var/tmp` under the the job's temporary scratch directory. This is useful, because the job's scratch directory will be cleaned up after the job completes, two concurrent jobs will not interfere with each other, and because jobs will not be able to fill up the real `/tmp` directory. Another use case might be for home directories, which some jobs might want to write to, but that should be cleaned up after each job run. The default value if not defined will be that no directories are mounted in the job's temporary scratch directory.

If the job's execute directory is encrypted, `/tmp` and `/var/tmp` are automatically added to `MOUNT_UNDER_SCRATCH` when the job is run (they will not show up if `MOUNT_UNDER_SCRATCH` is examined with *condor_config_val*).

Note that the MOUNT_UNDER_SCRATCH mounts do not take place until the PreCmd of the job, if any, completes. (See 11 for information on PreCmd.)

Also note that, if `MOUNT_UNDER_SCRATCH` is defined, it must either be a string or an expression that evaluates to a string.

For Docker Universe jobs, any directories that are mounted under scratch are also volume mounted on the same paths inside the container. That is, any reads or writes to files in those directories goes to the host filesytem under the scratch directory. This is useful if a container has limited space to grow a filesytem.

The following macros control if the *condor_startd* daemon should perform backfill computations whenever resources would otherwise be idle. See section 3.14.9 on page 479 on Configuring HTCondor 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`.

BACKFILL_SYSTEM 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 an HTCondor 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 an HTCondor resource is available to run normal HTCondor 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 3.7 beginning on page 352, especially sections 3.7.1, 3.7.1, and 3.7.1.

EVICT_BACKFILL A boolean expression that is evaluated whenever an HTCondor 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 3.7 beginning on page 352, especially sections 3.7.1, 3.7.1, and 3.7.1.

The following macros only apply to the *condor_startd* daemon when it is running on a multi-core machine. See section 3.7.1 on page 378 for details.

STARTD_RESOURCE_PREFIX A string which specifies what prefix to give the unique HTCondor resources that are advertised on multi-core machines. Previously, HTCondor 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, such as the ones created using tools like VMware or Xen, the old *virtual machine* terminology has been changed, and has become the term *slot*. Therefore, the default value of this prefix is now slot. If sites want to continue 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 a multi-core 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. This allows the *condor_startd* to notice console activity. Defaults to the number of slots in the machine, which is $(NUM_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). This defaults to all slots (N in a machine with N CPUs).

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 HTCondor 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 HTCondor 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 multi-core 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 multi-core host, and what attributes each one has. They are only needed if you do not want to have a multi-core machine report to HTCondor with a separate slot for each CPU, with all shared system resources evenly divided among them. Please read section 3.7.1 on page 378 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 searching 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 3.7.1 on page 378 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 3.7.1.

CLAIM_PARTITIONABLE_LEFTOVERS A boolean variable that defaults to True. When True within the configuration for both the *condor_schedd* and the *condor_startd*, and the *condor_schedd* claims a partitionable slot, the *condor_startd* returns the slot's ClassAd and a claim id for leftover resources. In doing so, the *condor_schedd* can claim multiple dynamic slots without waiting for a negotiation cycle.

MACHINE_RESOURCE_NAMES A comma and/or space separated list of resource names that represent custom resources specific to a machine. These resources are further intended to be statically divided or partitioned, and these resource names identify the configuration variables that define the partitioning. If used, custom resources without names in the list are ignored.

MACHINE_RESOURCE_<name> An integer that specifies the quantity of or list of identifiers for the customized local machine resource available for an SMP machine. The portion of this configuration variable's name identified with `<name>` will be used to label quantities of the resource allocated to a slot. If a quantity is specified, the resource is presumed to be fungible and slots will be allocated a quantity of the resource but specific instances will not be identified. If a list of identifiers is specified the quantity is the number of identifiers and slots will be allocated both a quantity of the resource and assigned specific resource identifiers.

OFFLINE_MACHINE_RESOURCE_<name> A comma and/or space separated list of resource identifiers for any customized local machine resources that are currently offline, and therefore should not be allocated to a slot. The identifiers specified here must match those specified by value of configuration variables `MACHINE_RESOURCE_<name>` or `MACHINE_RESOURCE_INVENTORY_<name>`, or the identifiers will be ignored. The `<name>` identifies the type of resource, as specified by the value of configuration variable `MACHINE_RESOURCE_NAMES`. This configuration variable is used to have resources that are detected and reported to exist by HTCondor, but not assigned to slots. A restart of the *condor_startd* is required for changes to this configuration variable to take effect.

MACHINE_RESOURCE_INVENTORY_<name> Specifies a command line that is executed upon start up of the *condor_startd* daemon. The script is expected to output an attribute definition of the form

```
Detected<xxx>=y
```

or of the form

```
Detected<xxx>="y, z, a, ..."
```

where `<xxx>` is the name of a resource that exists on the machine, and `y` is the quantity of the resource or `"y, z, a, ..."` is a comma and/or space separated list of identifiers of the resource that exist on the machine. This attribute is added to the machine ClassAd, such that these resources may be statically divided or partitioned. A script may be a convenient way to specify a calculated or detected quantity of the resource, instead of specifying a fixed quantity or list of the resource in the the configuration when set by `MACHINE_RESOURCE_<name>`.

ENVIRONMENT_FOR_Assigned<name> A space separated list of environment variables to set for the job. Each environment variable will be set to the list of assigned resources defined by the slot ClassAd attribute `Assigned<name>`. Each environment variable name may be followed by an equals sign and a Perl style regular expression that defines how to modify each resource ID before using it as the value of the environment variable. As a special case for CUDA GPUs, if the environment variable name is `CUDA_VISIBLE_DEVICES`, then the correct Perl style regular expression is applied automatically.

For example, with the configuration

```
ENVIRONMENT_FOR_AssignedGPUs = VISIBLE_GPUS=/^/gpuid:/
```

and with the machine ClassAd attribute `AssignedGPUs = "CUDA1, CUDA2"`, the job's environment will contain

```
VISIBLE_GPUS = gpuid:CUDA1, gpuid:CUDA2
```

ENVIRONMENT_VALUE_FOR_UnAssigned<name> Defines the value to set for environment variables specified in by configuration variable ENVIRONMENT_FOR_Assigned<name> when there is no machine ClassAd attribute Assigned<name> for the slot. This configuration variable exists to deal with the situation where jobs will use a resource that they have not been assigned because there is no explicit assignment. The CUDA runtime library (for GPUs) has this problem.

For example, where configuration is

```
ENVIRONMENT_FOR_AssignedGPUs = VISIBLE_GPUS
ENVIRONMENT_VALUE_FOR_UnAssignedGPUs = none
```

and there is *no* machine ClassAd attribute AssignedGPUs, the job's environment will contain

```
VISIBLE_GPUS = none
```

MUST_MODIFY_REQUEST_EXPRS A boolean value that defaults to False. When False, configuration variables whose names begin with MODIFY_REQUEST_EXPR are only applied if the job claim still matches the partitionable slot after modification. If True, the modifications always take place, and if the modifications cause the claim to no longer match, then the *condor_startd* will simply refuse the claim.

MODIFY_REQUEST_EXPR_REQUESTMEMORY An integer expression used by the *condor_startd* daemon to modify the evaluated value of the RequestMemory job ClassAd attribute, before it used to provision a dynamic slot. The default value is given by

```
quantize(RequestMemory,{128})
```

MODIFY_REQUEST_EXPR_REQUESTDISK An integer expression used by the *condor_startd* daemon to modify the evaluated value of the RequestDisk job ClassAd attribute, before it used to provision a dynamic slot. The default value is given by

```
quantize(RequestDisk,{1024})
```

MODIFY_REQUEST_EXPR_REQUESTCPUS An integer expression used by the *condor_startd* daemon to modify the evaluated value of the RequestCpus job ClassAd attribute, before it used to provision a dynamic slot. The default value is given by

```
quantize(RequestCpus,{1})
```

NUM_SLOTS_TYPE_<N> This macro controls how many of a given slot type are actually reported to HTCondor. There is no default.

NUM_SLOTS An integer value representing the number of slots reported when the multi-core machine is being evenly divided, and the slot type settings described above are not being used. The default is one slot for each CPU. This setting can be used to reserve some CPUs on a multi-core machine, which would not be reported to the HTCondor pool. This value cannot be used to make HTCondor advertise more slots than there are CPUs on the machine. To do that, use NUM_CPUS.

ALLOW_VM_CRUFT A boolean value that HTCondor sets and uses internally, currently defaulting to `True`. When `True`, HTCondor 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`, HTCondor does *not* look for variables named with the previously used string `VM` after searching unsuccessfully for the string `SLOT`.

The following variables set consumption policies for partitionable slots. Section 3.7.1 details consumption policies.

CONSUMPTION_POLICY A boolean value that defaults to `False`. When `True`, consumption policies are enabled for partitionable slots within the *condor_startd* daemon. Any definition of the form `SLOT_TYPE_<N>_CONSUMPTION_POLICY` overrides this global definition for the given slot type.

CONSUMPTION_<Resource> An expression that specifies a consumption policy for a particular resource within a partitionable slot. To support a consumption policy, each resource advertised by the slot must have such a policy configured. Custom resources may be specified, substituting the resource name for `<Resource>`. Any definition of the form `SLOT_TYPE_<N>_CONSUMPTION_<Resource>` overrides this global definition for the given slot type. CPUs, memory, and disk resources are always advertised by *condor_startd*, and have the default values:

```
CONSUMPTION_CPUS = quantize(target.RequestCpus,{1})
CONSUMPTION_MEMORY = quantize(target.RequestMemory,{128})
CONSUMPTION_DISK = quantize(target.RequestDisk,{1024})
```

Custom resources have no default consumption policy.

SLOT_WEIGHT An expression that specifies a slot's weight, used as a multiplier the *condor_negotiator* daemon during matchmaking to assess user usage of a slot, which affects user priority. Defaults to `Cpus`.

In the case of slots with consumption policies, the cost of each match is is assessed as the difference in the slot weight expression before and after the resources consumed by the match are deducted from the slot. Only Memory, Cpus and Disk are valid attributes for this parameter.

NUM_CLAIMS Specifies the number of claims a partitionable slot will advertise for use by the *condor_negotiator* daemon. In the case of slots with a defined consumption policy, the *condor_negotiator* may match more than one job to the slot in a single negotiation cycle. For partitionable slots with a consumption policy, `NUM_CLAIMS` defaults to the number of CPUs owned by the slot. Otherwise, it defaults to 1.

The following configuration variables support java universe jobs.

JAVA The full path to the Java interpreter (the Java Virtual Machine).

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.

The following configuration variables control .NET version advertisement.

STARTD_PUBLISH_DOTNET A boolean value that controls the advertising of the .NET framework on Windows platforms. When `True`, the *condor_startd* will advertise all installed versions of the .NET framework within the `DotNetVersions` attribute in the *condor_startd* machine ClassAd. The default value is `True`. Set the value to `false` to turn off .NET version advertising.

DOT_NET_VERSIONS A string expression that administrators can use to override the way that .NET versions are advertised. If the administrator wishes to advertise .NET installations, but wishes to do so in a format different than what the *condor_startd* publishes in its ClassAds, setting a string in this expression will result in the *condor_startd* publishing the string when `STARTD_PUBLISH_DOTNET` is `True`. No value is set by default.

These macros control the power management capabilities of the *condor_startd* to optionally put the machine in to a low power state and wake it up later. See section 3.18 on page 501 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"`. 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 A string expression that represents lower power state. When this state name evaluates to a valid state other than `"NONE"`, causes HTCondor to put the machine into the specified low power state. 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, this is Sleep (standby)

`"S2"`, `"2"`: On Windows, this is Sleep (standby)

`"S3"`, `"3"`, `"RAM"`, `"MEM"`, `"SUSPEND"`: On Windows, this is Sleep (standby)

`"S4"`, `"4"`, `"DISK"`, `"HIBERNATE"`: Hibernate

`"S5"`, `"5"`, `"SHUTDOWN"`, `"OFF"`: 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 notation was adopted because 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, such as "RAM" and "DISK", 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".

UNHIBERNATE A boolean expression that specifies when an offline machine should be woken up. The default value is MachineLastMatchTime =!= UNDEFINED. This expression does not do anything, unless there is an instance of *condor_rooster* running, or another program that evaluates the Unhibernate expression of offline machine ClassAds. In addition, the collecting of offline machine ClassAds must be enabled for this expression to work. The variable COLLECTOR_PERSISTENT_AD_LOG on page 290 detailed on page 260 explains this. The special attribute MachineLastMatchTime is updated in the ClassAds of offline machines when a job would have been matched to the machine if it had been online. For multi-slot machines, the offline ClassAd for slot1 will also contain the attributes slot<X>_MachineLastMatchTime, where X is replaced by the slot id of the other slots that would have been matched while offline. This allows the slot1 UNHIBERNATE expression to refer to all of the slots on the machine, in case that is necessary. By default, *condor_rooster* will wake up a machine if any slot on the machine has its UNHIBERNATE expression evaluate to True.

HIBERNATION_PLUGIN A string which specifies the path and executable name of the hibernation plug-in that the *condor_startd* should use in the detection of low power states and switching to the low power states. The default value is $(LIBEXEC)/power_state. A default executable in that location which meets these specifications is shipped with HTCondor.

The *condor_startd* initially invokes this plug-in with both the value defined for HIBERNATION_PLUGIN_ARGS and the argument *ad*, and expects the plug-in to output a ClassAd to its standard output stream. The *condor_startd* will use this ClassAd to determine what low power setting to use on further invocations of the plug-in. To that end, the ClassAd must contain the attribute HibernationSupportedStates, a comma separated list of low power modes that are available. The recognized mode strings are the same as those in the table for the configuration variable HIBERNATE. The optional attribute HibernationMethod specifies a string which describes the mechanism used by the plug-in. The default Linux plug-in shipped with HTCondor will produce one of the strings NONE, /sys, /proc, or pm-utils. The optional attribute HibernationRawMask is an integer which represents the bit mask of the modes detected.

Subsequent *condor_startd* invocations of the plug-in have command line arguments defined by HIBERNATION_PLUGIN_ARGS plus the argument **set** <*power-mode*>, where <*power-mode*> is one of the supported states as given in the attribute HibernationSupportedStates.

HIBERNATION_PLUGIN_ARGS Command line arguments appended to the command that invokes the plug-in. The additional argument *ad* is appended when the *condor_startd* initially invokes the plug-in.

HIBERNATION_OVERRIDE_WOL A boolean value that defaults to False. When True, it causes the *condor_startd* daemon's detection of the whether or not the network interface handles WOL packets to be ignored. When False, hibernation is disabled if the network interface does not use WOL packets to wake from hibernation. Therefore, when True hibernation can be enabled despite the fact that WOL packets are not used to wake machines.

LINUX_HIBERNATION_METHOD A string that can be used to override the default search used by HTCondor on Linux platforms to detect the hibernation method to use. This is used by the default hibernation plug-in executable that is shipped with HTCondor. 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 This configuration variable is no longer used. It has been replaced by `COLLECTOR_PERSISTENT_AD_LOG`.

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.

The following macros control the optional computation of resource availability statistics in the *condor_startd*.

STARTD_COMPUTE_AVAIL_STATS A boolean value that determines if the *condor_startd* computes resource availability statistics. The default is `False`.

If `STARTD_COMPUTE_AVAIL_STATS` is `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.

STARTD_AVAIL_CONFIDENCE 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, so 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.

DOCKER Defines the path and executable name of the Docker CLI. The default value is /usr/bin/docker. Remember that the condor user must also be in the docker group for Docker Universe to work. See the Docker universe manual section for more details (3.16.2) An example of the configuration for running the Docker CLI:

```
DOCKER = /usr/bin/docker
```

DOCKER_VOLUMES A list of directories on the host execute machine to be volume mounted within the container. See the Docker Universe section for full details. (3.16.2)

DOCKER_IMAGE_CACHE_SIZE The number of most recently used Docker images that will be kept on the local machine. The default value is 20.

DOCKER_DROP_ALL_CAPABILITIES A class ad expression, which defaults to true. Evaluated in the context of the job ad and the machine ad, when true, runs the docker container with the command line option –drop-all-capabilities. Admins should be very careful with this setting, and only allow trusted users to run with full linux capabilities within the container.

OPENMPI_INSTALL_PATH The location of the Open MPI installation on the local machine. Referenced by `examples/openmpiscript`, which is used for running Open MPI jobs in the parallel universe. The Open MPI bin and lib directories should exist under this path. The default value is `/usr/lib64/openmpi`.

OPENMPI_EXCLUDE_NETWORK_INTERFACES A comma-delimited list of network interfaces that Open MPI should not use for MPI communications. Referenced by `examples/openmpiscript`, which is used for running Open MPI jobs in the parallel universe.

The list should contain any interfaces that your job could potentially see from any execute machine. The list may contain undefined interfaces without generating errors. Open MPI should exclusively use low latency/high speed networks it finds (e.g. InfiniBand) regardless of this setting. The default value is `docker0,virbr0`.

3.5.9 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 `TotalLocalJobsRunning < 200`. 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 HTCondor 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 HTCondor that existed before the **local** universe to one that includes the **local** universe, but without utilizing the newer, provided configuration files.

LOCAL_UNIV_EXECUTE A string value specifying the execute location for local universe jobs. Each running local universe job will receive a uniquely named subdirectory within this directory. If not specified, it defaults to `$(SPOOL)/local_univ_execute`.

START_SCHEDULER_UNIVERSE A boolean value that defaults to `TotalSchedulerJobsRunning < 200`. 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
```

SCHEDD_USES_STARTD_FOR_LOCAL_UNIVERSE A boolean value that defaults to false. When true, the *condor_schedd* will spawn a special startd process to run local universe jobs. This allows local universe jobs to run with both a condor_shadow and a condor_starter, which means that file transfer will work with local universe jobs.

MAX_JOBS_RUNNING An integer representing a limit on the number of *condor_shadow* processes spawned by a given *condor_schedd* daemon, for all job universes except grid, scheduler, and local universe. Limiting the number of running scheduler and local universe jobs can be done using `START_LOCAL_UNIVERSE` and `START_SCHEDULER_UNIVERSE`. The actual number of allowed *condor_shadow* daemons may be reduced, if the amount of memory defined by `RESERVED_SWAP` limits the number of *condor_shadow* daemons. A value for `MAX_JOBS_RUNNING` that is less than or equal to 0 prevents any new job from starting. Changing this setting to be below the current number of jobs that are running will cause running jobs to be aborted until the number running is within the limit.

Like all integer configuration variables, `MAX_JOBS_RUNNING` may be a ClassAd expression that evaluates to an integer, and which refers to constants either directly or via macro substitution. The default value is an expression that depends on the total amount of memory and the operating system. The default expression requires 1MByte of RAM per running job on the submit machine. In some environments and configurations, this is overly generous and can be cut by as much as 50%. On Windows platforms, the number of running jobs is capped at 2000. A 64-bit version of Windows is recommended in order to raise the value above the default. Under Unix, the maximum default is now 10,000. To scale higher, we recommend that the system ephemeral port range is extended such that there are at least 2.1 ports per running job.

Here are example configurations:

```
## Example 1:
MAX_JOBS_RUNNING = 10000

## Example 2:
## This is more complicated, but it produces the same limit as the default.
## First define some expressions to use in our calculation.
## Assume we can use up to 80% of memory and estimate shadow private data
## size of 800k.
MAX_SHADOWS_MEM = ceiling($(DETECTED_MEMORY)*0.8*1024/800)
## Assume we can use ~21,000 ephemeral ports (avg ~2.1 per shadow).
## Under Linux, the range is set in /proc/sys/net/ipv4/ip_local_port_range.
MAX_SHADOWS_PORTS = 10000
## Under windows, things are much less scalable, currently.
## Note that this can probably be safely increased a bit under 64-bit windows.
MAX_SHADOWS_OPSYS = ifThenElse(regexp("WIN.*","$(OPSYS)"),2000,100000)
## Now build up the expression for MAX_JOBS_RUNNING.  This is complicated
## due to lack of a min() function.
MAX_JOBS_RUNNING = $(MAX_SHADOWS_MEM)
MAX_JOBS_RUNNING = \
  ifThenElse( $(MAX_SHADOWS_PORTS) < $(MAX_JOBS_RUNNING), \
              $(MAX_SHADOWS_PORTS), \
              $(MAX_JOBS_RUNNING) )
MAX_JOBS_RUNNING = \
  ifThenElse( $(MAX_SHADOWS_OPSYS) < $(MAX_JOBS_RUNNING), \
              $(MAX_SHADOWS_OPSYS), \
              $(MAX_JOBS_RUNNING) )
```

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_JOBS_PER_OWNER This integer value limits the number of jobs any given owner (user) is permitted to have within a *condor_schedd* daemon's queue. A job submission fails if it would cause this limit on the number of jobs to be exceeded. The default value is 100000.

This configuration variable may be most useful in conjunction with MAX_JOBS_SUBMITTED, to ensure that no one user can dominate the queue.

MAX_RUNNING_SCHEDULER_JOBS_PER_OWNER This integer value limits the number of scheduler universe jobs that any given owner (user) can have running at one time. This limit will affect the number of running Dagman jobs, but not the number of nodes within a DAG. The default is no limit

MAX_JOBS_PER_SUBMISSION This integer value limits the number of jobs any single submission is permitted to add to a *condor_schedd* daemon's queue. The whole submission fails if the number of jobs would exceed this limit. The default value is 20000.

This configuration variable may be useful for catching user error, and for protecting a busy *condor_schedd* daemon from the excessively lengthy interruption required to accept a very large number of jobs at one time.

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.

CURB_MATCHMAKING A ClassAd expression evaluated by the *condor_schedd* in the context of the *condor_schedd* daemon's own ClassAd. While this expression evaluates to `True`, the *condor_schedd* will refrain from requesting more resources from a *condor_negotiator*. Defaults to `False`.

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.

FILE_TRANSFER_DISK_LOAD_THROTTLE This configures throttling of file transfers based on the disk load generated by file transfers. The maximum number of concurrent file transfers is specified by `MAX_CONCURRENT_UPLOADS` and `MAX_CONCURRENT_DOWNLOADS`. Throttling will dynamically reduce the level of concurrency further to attempt to prevent disk load from exceeding the specified level. Disk load is computed as the average number of file transfer processes conducting read/write operations at the same time. The throttle may be specified as a single floating point number or as a range. Syntax for the range is the smaller number followed by 1 or more spaces or tabs, the string `"to"`, 1 or more spaces or tabs, and then the larger number. Example:

```
FILE_TRANSFER_DISK_LOAD_THROTTLE = 5 to 6.5
```

If only a single number is provided, this serves as the upper limit, and the lower limit is set to 90% of the upper limit. When the disk load is above the upper limit, no new transfers will be started. When between the lower and upper limits, new transfers will only be started to replace ones that finish. There is no default value if this variable is not defined.

FILE_TRANSFER_DISK_LOAD_THROTTLE_WAIT_BETWEEN_INCREMENTS This rarely configured variable sets the waiting period between increments to the concurrency level set by `FILE_TRANSFER_DISK_LOAD_THROTTLE`. The default is 1 minute. A value that is too short risks starting too many transfers before their effect on the disk load becomes apparent.

FILE_TRANSFER_DISK_LOAD_THROTTLE_SHORT_HORIZON This rarely configured variable specifies the string name of the short monitoring time span to use for throttling. The named time span must exist in `TRANSFER_IO_REPORT_TIMESPANS`. The default is `1m`, which is 1 minute.

FILE_TRANSFER_DISK_LOAD_THROTTLE_LONG_HORIZON This rarely configured variable specifies the string name of the long monitoring time span to use for throttling. The named time span must exist in `TRANSFER_IO_REPORT_TIMESPANS`. The default is `5m`, which is 5 minutes.

TRANSFER_QUEUE_USER_EXPR This rarely configured expression specifies the user name to be used for scheduling purposes in the file transfer queue. The scheduler attempts to give equal weight to each user when there are multiple jobs waiting to transfer files within the limits set by MAX_CONCURRENT_UPLOADS and/or MAX_CONCURRENT_DOWNLOADS. When choosing a new job to allow to transfer, the first job belonging to the transfer queue user who has least number of active transfers will be selected. In case of a tie, the user who has least recently been given an opportunity to start a transfer will be selected. By default, a transfer queue user is identified as the job owner. A different user name may be specified by configuring TRANSFER_QUEUE_USER_EXPR to a string expression that is evaluated in the context of the job ad. For example, if this expression were set to a name that is the same for all jobs, file transfers would be scheduled in first-in-first-out order rather than equal share order. Note that the string produced by this expression is used as a prefix in the ClassAd attributes for per-user file transfer I/O statistics that are published in the *condor_schedd* ClassAd.

MAX_TRANSFER_INPUT_MB This integer expression specifies the maximum allowed total size in MiB of the input files that are transferred for a job. This expression does *not* apply to grid universe, standard universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value `-1` indicates no limit. The default value is -1. The job may override the system setting by specifying its own limit using the MaxTransferInputMB attribute. If the observed size of all input files at submit time is larger than the limit, the job will be immediately placed on hold with a HoldReasonCode value of 32. If the job passes this initial test, but the size of the input files increases or the limit decreases so that the limit is violated, the job will be placed on hold at the time when the file transfer is attempted.

MAX_TRANSFER_OUTPUT_MB This integer expression specifies the maximum allowed total size in MiB of the output files that are transferred for a job. This expression does *not* apply to grid universe, standard universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value `-1` indicates no limit. The default value is -1. The job may override the system setting by specifying its own limit using the MaxTransferOutputMB attribute. If the total size of the job's output files to be transferred is larger than the limit, the job will be placed on hold with a HoldReasonCode value of 33. The output will be transferred up to the point when the limit is hit, so some files may be fully transferred, some partially, and some not at all.

MAX_TRANSFER_QUEUE_AGE The number of seconds after which an aged and queued transfer may be dequeued from the transfer queue, as it is presumably hung. Defaults to 7200 seconds, which is 120 minutes.

TRANSFER_IO_REPORT_INTERVAL The sampling interval in seconds for collecting I/O statistics for file transfer. The default is 10 seconds. To provide sufficient resolution, the sampling interval should be small compared to the smallest time span that is configured in TRANSFER_IO_REPORT_TIMESPANS. The shorter the sampling interval, the more overhead of data collection, which may slow down the *condor_schedd*. See page 1030 for a description of the published attributes.

TRANSFER_IO_REPORT_TIMESPANS A string that specifies a list of time spans over which I/O statistics are reported, using exponential moving averages (like the 1m, 5m, and 15m load averages in Unix). Each entry in the list consists of a label followed by a colon followed by the number of seconds over which the named time span should extend. The default is `1m:60 5m:300 1h:3600 1d:86400`. To provide sufficient resolution, the smallest reported time span should be large compared to the sampling interval, which is configured by TRANSFER_IO_REPORT_INTERVAL. See page 1030 for a description of the published attributes.

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 8. If the limit is reached, the next query will be handled in the *condor_schedd*'s main process.

CONDOR_Q_USE_V3_PROTOCOL A boolean value that, when `True`, causes the *condor_schedd* to use an algorithm that responds to *condor_q* requests by *not* forking itself to handle each request. It instead handles the requests in a non-blocking way. The default value is `True`.

CONDOR_Q_DASH_BATCH_IS_DEFAULT A boolean value that, when `True`, causes *condor_q* to print the **-batch** output unless the **-nobatch** option is used or the other arguments to *condor_q* are incompatible with batch mode. For instance **-long** is incompatible with **-batch**. The default value is `True`.

CONDOR_Q_ONLY_MY_JOBS A boolean value that, when `True`, causes *condor_q* to request that only the current user's jobs be queried unless the current user is a queue superuser. It also causes the *condor_schedd* to honor that request. The default value is `True`. A value of `False` in either *condor_q* or the *condor_schedd* will result in the old behavior of querying all jobs.

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).

ABSENT_SUBMITTER_LIFETIME This macro determines the maximum time that the *condor_schedd* will remember a submitter after the last job for that submitter leaves the queue. It is defined in terms of seconds and defaults to 1 week.

ABSENT_SUBMITTER_UPDATE_RATE This macro can be used to set the maximum rate at which the *condor_schedd* sends updates to the *condor_collector* for submitters that have no jobs in the queue. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

WINDOWED_STAT_WIDTH The number of seconds that forms a time window within which performance statistics of the *condor_schedd* daemon are calculated. Defaults to 300 seconds.

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 setting 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_STOP_COUNT An integer value representing the number of jobs operated on at one time by the *condor_schedd* daemon, when throttling the rate at which jobs are stopped via *condor_rm*, *condor_hold*, or *condor_vacate_job*. The default and minimum values are both 1. This variable is ignored for grid and scheduler universe jobs.

JOB_STOP_DELAY An integer value representing the number of seconds delay utilized by the *condor_schedd* daemon, when throttling the rate at which jobs are stopped via *condor_rm*, *condor_hold*, or *condor_vacate_job*. The *condor_schedd* daemon stops `$(JOB_STOP_COUNT)` jobs at a time, then delays for `$(JOB_STOP_DELAY)` seconds before stopping the next set of jobs. This delay prevents a sudden, large load on resources required by the jobs when they are terminating. The resulting job stop rate averages as fast as `JOB_STOP_COUNT/JOB_STOP_DELAY` jobs per 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. The default value is 0, which means jobs will be stopped as fast as possible. This variable is ignored for grid and scheduler universe jobs.

JOB_IS_FINISHED_COUNT An integer value representing the number of jobs that the *condor_schedd* will let permanently leave the job queue each time that it examines the jobs that are ready to do so. The default value is 1.

JOB_IS_FINISHED_INTERVAL The *condor_schedd* maintains a list of jobs that are ready to permanently leave the job queue, for example, when they have completed or been removed. This integer-valued macro specifies a delay in seconds between instances of 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 248) 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.

STARTD_SENDS_ALIVES Note: This setting is deprecated, and may go away in a future version of HTCondor. This setting is mainly useful when running mixing very old *condor_schedd* daemons with newer pools. A boolean value that defaults to `True`, causing keep alive messages to be sent from the *condor_startd* to the *condor_schedd* by TCP during a claim. When `False`, the *condor_schedd* daemon sends keep alive signals to the *condor_startd*, reversing the direction. If both *condor_startd* and *condor_schedd* daemons are HTCondor version 7.5.4 or more recent, this variable is only used by the *condor_schedd* daemon. For earlier HTCondor versions, the variable must be set to the same value, and it must be set for both daemons.

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 `NEGOTIATOR_CONSIDER_EARLY_PREEMPTION` is `True`, and 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, where a small value is 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 The estimated private virtual memory size of each *condor_shadow* process in KiB. This value is only used if RESERVED_SWAP is non-zero. The default value is 800.

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 A comma and/or space separated list of user names on a given machine that are given *super-user access* to the job queue, meaning that they can modify or delete the job ClassAds of other users. When not on this list, users can only modify or delete their own ClassAds from the job queue. Whatever user name corresponds with the UID that HTCondor is running as – usually user condor – will automatically be included in this list, because that is needed for HTCondor's proper functioning. See section 3.8.13 on UIDs in HTCondor for more details on this. By default, the Unix user root and the Windows user administrator are given the ability to remove other user's jobs, in addition to user condor. In addition to a single user, Unix user groups may be specified by using a special syntax defined for this configuration variable; the syntax is the percent character (%) followed by the user group name. All members of the user group are given super-user access.

QUEUE_SUPER_USER_MAY_IMPERSONATE A regular expression that matches the user names (that is, job owner names) that the queue super user may impersonate when managing jobs. When not set, the default behavior is to allow impersonation of any user who has had a job in the queue during the life of the *condor_schedd*. For proper functioning of the *condor_shadow*, the *condor_gridmanager*, and the *condor_job_router*, this expression, if set, must match the owner names of all jobs that these daemons will manage. Note that a regular expression that matches only part of the user name is still considered a match. If acceptance of partial matches is not desired, the regular expression should begin with `^` and end with `$`.

SYSTEM_JOB_MACHINE_ATTRS This macro specifies a space and/or comma separated list of machine attributes that should be recorded in the job ClassAd. The default attributes are `Cpus` and `SlotWeight`. When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store is specified by the configuration variable `SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH`. A machine attribute named `X` will be inserted into the job ClassAd as an attribute named `MachineAttrX0`. The previous value of this attribute will be named `MachineAttrX1`, the previous to that will be named `MachineAttrX2`, and so on, up to the specified history length. A history of length 1 means that only `MachineAttrX0` will be recorded. Additional attributes to record may be specified on a per-job basis by using the **job_machine_attrs** submit file command. The value recorded in the job ClassAd is the evaluation of the machine attribute in the context of the job ClassAd when the *condor_schedd* daemon initiates the start up of the job. If the evaluation results in an `Undefined` or `Error` result, the value recorded in the job ClassAd will be `Undefined` or `Error` respectively.

SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH The integer number of run attempts to store in the job ClassAd when recording the values of machine attributes listed in `SYSTEM_JOB_MACHINE_ATTRS`. The default is 1. The history length may also be extended on a per-job basis by using the submit file command **job_machine_attrs_history_length**. The larger of the system and per-job history lengths will be used. A history length of 0 disables recording of machine attributes.

SCHEDD_LOCK 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 3.5.7 on page 242 for defaults and composition of valid HTCondor 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`.

SCHEDD_ATTRS This macro is described in section 3.5.3 as `<SUBSYS>_ATTRS`.

SCHEDD_DEBUG This macro (and other settings related to debug logging in the *condor_schedd*) is described in section 3.5.2 as `<SUBSYS>_DEBUG`.

SCHEDD_ADDRESS_FILE This macro is described in section 3.5.3 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 Defines a comma and/or space separated list of *condor_negotiator* host names for pools in which the *condor_schedd* should attempt to run jobs. If not set, the *condor_schedd* will query the *condor_collector* daemons for the addresses of the *condor_negotiator* daemons. If set, then the *condor_negotiator*

daemons must be specified in order, corresponding to the list set by FLOCK_COLLECTOR_HOSTS. In the typical case, where each pool has the *condor_collector* and *condor_negotiator* running on the same machine, $(FLOCK_NEGOTIATOR_HOSTS) should have the same definition as $(FLOCK_COLLECTOR_HOSTS). This configuration value is also typically used as a macro for adding the *condor_negotiator* to the relevant authorization lists.

FLOCK_COLLECTOR_HOSTS This macro defines a list of collector host names (not including the local $(COLLECTOR_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 3.5.3) must also be configured to allow negotiators from all of the pools to contact the *condor_schedd* at the NEGOTIATOR authorization level. Similarly, the central managers of the remote pools must be configured to allow this *condor_schedd* to join the pool (this requires ADVERTISE_SCHEDD authorization level, which defaults to WRITE).

FLOCK_INCREMENT This integer value controls how quickly flocking to various pools will occur. It defaults to 1, meaning that pools will be considered for flocking slowly. The first *condor_collector* daemon listed in FLOCK_COLLECTOR_HOSTS will be considered for flocking, and then the second, and so on. A larger value increases the number of *condor_collector* daemons to be considered for flocking. For example, a value of 2 will partition the FLOCK_COLLECTOR_HOSTS into sets of 2 *condor_collector* daemons, and each set will be considered for flocking.

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 an HTCondor submit file. By default, this value is 60 seconds. A value of 0 prevents the *condor_schedd* from performing the periodic evaluations.

MAX_PERIODIC_EXPR_INTERVAL This macro determines the maximum period, in seconds, between evaluation of periodic job control expressions, such as periodic_hold, periodic_release, and periodic_remove, given by the user in an HTCondor submit file. By default, this value is 1200 seconds. If HTCondor is behind on processing events, the actual period between evaluations may be higher than specified.

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) and the upper bound is configured with MAX_PERIODIC_EXPR_INTERVAL (default 1200 seconds).

SYSTEM_PERIODIC_HOLD This expression behaves identically to the job expression `periodic_hold`, but it is evaluated for every 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_HOLD_REASON This string expression is evaluated when the job is placed on hold due to `SYSTEM_PERIODIC_HOLD` evaluating to `True`. If it evaluates to a non-empty string, this value is used to set the job attribute `HoldReason`. Otherwise, a default description is used.

SYSTEM_PERIODIC_HOLD_SUBCODE This integer expression is evaluated when the job is placed on hold due to `SYSTEM_PERIODIC_HOLD` evaluating to `True`. If it evaluates to a valid integer, this value is used to set the job attribute `HoldReasonSubCode`. Otherwise, a default of 0 is used. The attribute `HoldReasonCode` is set to 26, which indicates that the job went on hold due to a system job policy expression.

SYSTEM_PERIODIC_RELEASE This expression behaves identically to a job's definition of a **periodic_release** expression in a submit description file, but it is evaluated for every 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 && (time() - EnteredCurrentStatus) > 1200 ) &&  \
    (HoldReasonCode == 6 && HoldReasonSubCode == 110)
```

SYSTEM_PERIODIC_REMOVE This expression behaves identically to the job expression `periodic_remove`, but it is evaluated for every job in the queue. As it is in the configuration file, it is easy for an administrator to set a remove policy that applies to all jobs. 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 && time() - 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 1200 (20 minutes).

GRACEFULLY_REMOVE_JOBS A boolean value that causes jobs to be gracefully removed when the default value of `True`. A submit description file command **want_graceful_removal** overrides the value set for this configuration variable.

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_ResidentSetSize = 25%
SCHEDD_ROUND_ATTR_ProportionalSetSizeKb = 25%
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 A boolean value that, when `True`, causes the *condor_schedd* to make a backup of the job queue as it starts. When `True`, the *condor_schedd* creates a host-specific backup of the current spool file to the spool directory. This backup file will be overwritten each time the *condor_schedd* starts. Defaults to `False`.

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_PREEMPTION_RANK 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 is a string which identifies a group of which this machine is a member. 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 HTCondor user, when a job leaves the *condor_schedd*'s queue, a copy of the job's 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`. HTCondor 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 files are written.

DEDICATED_SCHEDULER_USE_FIFO When this parameter is set to true (the default), parallel universe jobs will be scheduled in a first-in, first-out manner. When set to false, parallel jobs are scheduled using a best-fit algorithm. Using the best-fit algorithm is not recommended, as it can cause starvation.

DEDICATED_SCHEDULER_WAIT_FOR_SPOOLER A boolean value that when `True`, causes the dedicated scheduler to schedule parallel universe jobs in a very strict first-in, first-out manner. When the default value of `False`, parallel jobs that are being remotely submitted to a scheduler and are on hold, waiting for spooled input files to arrive at the scheduler, will not block jobs that arrived later, but whose input files have finished spooling. When `True`, jobs with larger cluster IDs, but that are in the Idle state will not be scheduled to run until all earlier jobs have finished spooling in their input files and have been scheduled.

DEDICATED_SCHEDULER_DELAY_FACTOR Limits the cpu usage of the dedicated scheduler within the *condor_schedd*. The default value of 5 is the ratio of time spent not in the dedicated scheduler to the time scheduling parallel jobs. Therefore, the default caps the time spent in the dedicated scheduler to 20%.

SCHEDD_SEND_VACATE_VIA_TCP A boolean value that defaults to `True`. 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. If the job cluster number reaches the value set by `SCHEDD_CLUSTER_MAXIMUM_VALUE` and wraps, it will be re-set to the value given by this variable. The default value is 1.

SCHEDD_CLUSTER_INCREMENT_VALUE A positive integer that defaults to 1, representing a stride used for the 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 variable.

SCHEDD_CLUSTER_MAXIMUM_VALUE An integer that specifies an upper bound on assigned job cluster id values. For value M, the maximum job cluster id assigned to any job will be $M - 1$. When the maximum id is reached, cluster ids will continue assignment using `SCHEDD_CLUSTER_INITIAL_VALUE`. The default value of this variable is zero, which represents the behavior of having no maximum cluster id value.

Note that HTCondor does not check for nor take responsibility for duplicate cluster ids for queued jobs. If `SCHEDD_CLUSTER_MAXIMUM_VALUE` is set to a non-zero value, the system administrator is responsible for ensuring that older jobs do not stay in the queue long enough for cluster ids of new jobs to wrap around and reuse the same id. With a low enough value, it is possible for jobs to be erroneously assigned duplicate cluster ids, which will result in a corrupt job queue.

GRIDMANAGER_SELECTION_EXPR By default, the *condor_schedd* daemon will start a new *condor_gridmanager* process for each discrete user that submits a grid universe job, that is, for each discrete value of job attribute `Owner` across all grid universe job ClassAds. For additional isolation and/or scalability of grid job management, additional *condor_gridmanager* processes can be spawned to share the load; to do so, set this variable to be a ClassAd expression. The result of the evaluation of this expression in the context of a grid universe job ClassAd will be treated as a hash value. All jobs that hash to the same value via this expression will go to the same *condor_gridmanager*. For instance, to spawn a separate *condor_gridmanager* process to manage each unique remote site, the following expression works:

```
GRIDMANAGER_SELECTION_EXPR = GridResource
```

CKPT_SERVER_CLIENT_TIMEOUT An integer which specifies how long in seconds the *condor_schedd* is willing to wait for a response from a checkpoint server before declaring the checkpoint server down. The value of 0 makes the schedd block for the operating system configured time (which could be a very long time) before the `connect()` returns on its own with a connection timeout. The default value is 20.

CKPT_SERVER_CLIENT_TIMEOUT_RETRY An integer which specifies how long in seconds the *condor_schedd* will ignore a checkpoint server that is deemed to be down. After this time elapses, the *condor_schedd* will try again in talking to the checkpoint server. The default is 1200.

SCHEDD_JOB_QUEUE_LOG_FLUSH_DELAY An integer which specifies an upper bound in seconds on how long it takes for changes to the job ClassAd to be visible to the HTCondor Job Router. The default is 5 seconds.

ROTATE_HISTORY_DAILY A boolean value that defaults to `False`. When `True`, the history file will be rotated daily, in addition to the rotations that occur due to the definition of `MAX_HISTORY_LOG` that rotate due to size.

ROTATE_HISTORY_MONTHLY A boolean value that defaults to `False`. When `True`, the history file will be rotated monthly, in addition to the rotations that occur due to the definition of `MAX_HISTORY_LOG` that rotate due to size.

SCHEDD_COLLECT_STATS_FOR_<Name> A boolean expression that when `True` creates a set of *condor_schedd* ClassAd attributes of statistics collected for a particular set. These attributes are named using the prefix of `<Name>`. The set includes each entity for which this expression is `True`. As an example, assume that *condor_schedd* statistics attributes are to be created for only user Einstein's jobs. Defining

```
SCHEDD_COLLECT_STATS_FOR_Einstein = (Owner=="einstein")
```

causes the creation of the set of statistics attributes with names such as `EinsteinJobsCompleted` and `EinsteinJobsCoredumped`.

SCHEDD_COLLECT_STATS_BY_<Name> Defines a string expression. The evaluated string is used in the naming of a set of *condor_schedd* statistics ClassAd attributes. The naming begins with `<Name>`, an underscore character, and the evaluated string. Each character not permitted in an attribute name will be converted to the underscore character. For example,

```
SCHEDD_COLLECT_STATS_BY_Host = splitSlotName(RemoteHost)[1]
```

a set of statistics attributes will be created and kept. If the string expression were to evaluate to `"storm.04.cs.wisc.edu"`, the names of two of these attributes will be `Host_storm_04_cs_wisc_edu_JobsCompleted` and `Host_storm_04_cs_wisc_edu_JobsCoredumped`.

SCHEDD_EXPIRE_STATS_BY_<Name> The number of seconds after which the *condor_schedd* daemon will stop collecting and discard the statistics for a subset identified by `<Name>`, if no event has occurred to cause any counter or statistic for the subset to be updated. If this variable is not defined for a particular `<Name>`, then the default value will be `60*60*24*7`, which is one week's time.

SIGNIFICANT_ATTRIBUTES A comma and/or space separated list of job ClassAd attributes that are to be added to the list of attributes for determining the sets of jobs considered as a unit (an auto cluster) in negotiation, when auto clustering is enabled. When defined, this list replaces the list that the *condor_negotiator* would define based upon machine ClassAds.

ADD_SIGNIFICANT_ATTRIBUTES A comma and/or space separated list of job ClassAd attributes that will always be added to the list of attributes that the *condor_negotiator* defines based upon machine ClassAds, for determining the sets of jobs considered as a unit (an auto cluster) in negotiation, when auto clustering is enabled.

REMOVE_SIGNIFICANT_ATTRIBUTES A comma and/or space separated list of job ClassAd attributes that are removed from the list of attributes that the *condor_negotiator* defines based upon machine ClassAds, for determining the sets of jobs considered as a unit (an auto cluster) in negotiation, when auto clustering is enabled.

SCHEDD_AUDIT_LOG The path and file name of the *condor_schedd* log that records user-initiated commands that modify the job queue. If not defined, there will be no *condor_schedd* audit log.

MAX_SCHEDD_AUDIT_LOG Controls the maximum amount of time that a log will be allowed to grow. When it is time to rotate a log file, it will be saved to a file with an ISO timestamp suffix. The oldest rotated file receives the file name suffix `.old`. The `.old` files are overwritten each time the maximum number of rotated files (determined by the value of `MAX_NUM_SCHEDD_AUDIT_LOG`) is exceeded. A value of 0 specifies that the file may grow without bounds. The following suffixes may be used to qualify the integer:

`Sec` for seconds

`Min` for minutes

`Hr` for hours

`Day` for days

`Wk` for weeks

MAX_NUM_SCHEDD_AUDIT_LOG The integer that controls the maximum number of rotations that the *condor_schedd* audit log is allowed to perform, before the oldest one will be rotated away. The default value is 1.

SCHEDD_USE_SLOT_WEIGHT A boolean that defaults to `False`. When `True`, the *condor_schedd* does use configuration variable `SLOT_WEIGHT` to weight running and idle job counts in the submitter ClassAd.

JOB_TRANSFORM_NAMES A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will contain a set of rules governing the transformation of jobs during submission. Each name in the list will be used in the name of configuration variable `JOB_TRANSFORM_<Name>`. Transforms are applied in the order in which names appear in this list. Names are not case-sensitive. There is no default value.

JOB_TRANSFORM_<Name> A single job transform specified as a set of transform rules in new classad syntax. The transform rules are applied to jobs that match the transform's `Requirements` expression as they are submitted. `<Name>` corresponds to a name listed in `JOB_TRANSFORM_NAMES`. Names are not case-sensitive. There is no default value.

SUBMIT_REQUIREMENT_NAMES A comma and/or space separated list of unique names, where each is used in the formation of a configuration variable name that will represent an expression evaluated to decide whether or not to reject a job submission. Each name in the list will be used in the name of configuration variable `SUBMIT_REQUIREMENT_<Name>`. There is no default value.

SUBMIT_REQUIREMENT_<Name> A boolean expression evaluated in the context of the *condor_schedd* daemon ClassAd, which is the `MY.` name space and the job ClassAd, which is the `TARGET.` name space. When `False`, it causes the *condor_schedd* to reject the submission of the job or cluster of jobs. `<Name>` corresponds to a name listed in `SUBMIT_REQUIREMENT_NAMES`. There is no default value.

SUBMIT_REQUIREMENT_<Name>_REASON An expression that evaluates to a string, to be printed for the job submitter when SUBMIT_REQUIREMENT_<Name> evaluates to False and the *condor_schedd* rejects the job. There is no default value.

SCHEDD_RESTART_REPORT The complete path to a file that will be written with report information. The report is written when the *condor_schedd* starts. It contains statistics about its attempts to reconnect to the *condor_startd* daemons for all jobs that were previously running. The file is updated periodically as reconnect attempts succeed or fail. Once all attempts have completed, a copy of the report is emailed to address specified by CONDOR_ADMIN. The default value is $(LOG)/ScheddRestartReport. If a blank value is set, then no report is written or emailed.

JOB_SPOOL_PERMISSIONS Control the permissions on the job's spool directory. Defaults to user which sets permissions to *0700*. Possible values are user, group, and world. If set to group, then the directory is group-accessible, with permissions set to *0750*. If set to world, then the directory is created with permissions set to *0755*.

CHOWN_JOB_SPOOL_FILES Prior to HTCondor 8.5.0 on unix, the condor_schedd would chown job files in the SPOOL directory between the condor account and the account of the job submitter. Now, these job files are always owned by the job submitter by default. To restore the older behavior, set this parameter to True. The default value is False.

IMMUTABLE_JOB_ATTRS A comma and/or space separated list of attributes provided by the administrator that cannot be changed, once they have committed values. No attributes are in this list by default.

SYSTEM_IMMUTABLE_JOB_ATTRS A predefined comma and/or space separated list of attributes that cannot be changed, once they have committed values. The hard-coded value is: Owner ClusterId ProcId MyType TargetType.

PROTECTED_JOB_ATTRS A comma and/or space separated list of attributes provided by the administrator that can only be altered by the queue super-user, once they have committed values. No attributes are in this list by default.

SYSTEM_PROTECTED_JOB_ATTRS A predefined comma and/or space separated list of attributes that can only be altered by the queue super-user, once they have committed values. The hard-code value is empty.

3.5.10 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 3.5.2 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 is no additional cost to the *condor_schedd*, but it means that *condor_q* will not 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 ClassAd, 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.

SHADOW_WORKLIFE The integer number of seconds after which the *condor_shadow* will exit when the current job finishes, instead of fetching a new job to manage. Having the *condor_shadow* continue managing jobs helps reduce overhead and can allow the *condor_schedd* to achieve higher job completion rates. The default is 3600, one hour. The value 0 causes *condor_shadow* to exit after running a single job.

COMPRESS_PERIODIC_CKPT A boolean value that when `True`, directs the *condor_shadow* to instruct applications to compress periodic checkpoints when possible. The default is `False`.

COMPRESS_VACATE_CKPT A boolean value that when `True`, directs the *condor_shadow* to 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_CKPT_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.

SHADOW_CHECKPROXY_INTERVAL The number of seconds between tests to see if the job proxy has been updated or should be refreshed. The default is 600 seconds (10 minutes). This variable's value should be small in comparison to the refresh interval required to keep delegated credentials from expiring (configured via `DELEGATE_JOB_GSI_CREDENTIALS_REFRESH` and `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`). If this variable's value is too small, proxy updates could happen very frequently, potentially creating a lot of load on the submit machine.

SHADOW_RUN_UNKNOWN_USER_JOBS A boolean that defaults to `False`. When `True`, it allows the *condor_shadow* daemon to run jobs as user `nobody` when remotely submitted and from users not in the local password file.

SHADOW_STATS_LOG The full path and file name of a file that stores TCP statistics for shadow file transfers. (Note that the shadow logs TCP statistics to this file by default. Adding `D_STATS` to the `SHADOW_DEBUG` value will cause TCP statistics to be logged to the normal shadow log file (`$(SHADOW_LOG)`).) If not defined, `SHADOW_STATS_LOG` defaults to `$(LOG)/XferStatsLog`. Setting `SHADOW_STATS_LOG` to `/dev/null` disables logging of shadow TCP file transfer statistics.

MAX_SHADOW_STATS_LOG Controls the maximum size in bytes or amount of time that the shadow TCP statistics log will be allowed to grow. If not defined, `MAX_SHADOW_STATS_LOG` defaults to `$(MAX_DEFAULT_LOG)`, which currently defaults to 10 MiB in size. Values are specified with the same syntax as `MAX_DEFAULT_LOG`.

3.5.11 condor_starter Configuration File Entries

These settings affect the *condor_starter*.

EXEC_TRANSFER_ATTEMPTS 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_RENICE_INCREMENT When the *condor_starter* spawns an HTCondor 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 HTCondor to always start jobs on the machines. HTCondor jobs would always run, but interactive response on the machines would never suffer. A user most likely will not notice HTCondor is running jobs. See section 3.7 on Startd Policy Configuration for more details on setting up a policy for starting and stopping jobs on a given machine.

The ClassAd expression is evaluated in the context of the job ad to an integer value, which 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 0, on Unix, and 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_LOG_NAME_APPEND A fixed value that sets the file name extension of the local log file used by the *condor_starter* daemon. Permitted values are `true`, `false`, `slot`, `cluster` and `jobid`. A value of `false`

will suppress the use of a file extension. A value of `true` gives the default behavior of using the slot name, unless there is only a single slot. A value of `slot` uses the slot name. A value of `cluster` uses the job's `ClusterId` ClassAd attribute. A value of `jobid` uses the job's `ProcId` ClassAd attribute. If `cluster` or `jobid` are specified, the resulting log files will persist until deleted by the user, so these two options should only be used to assist in debugging, not as permanent options.

STARTER_DEBUG This setting (and other settings related to debug logging in the starter) is described above in section 3.5.2 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 and file name of an executable or script. If specified, HTCondor never directly executes a job, but instead invokes this executable, allowing an administrator to specify the executable (wrapper script) that will handle the execution of all user jobs. 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 script must ultimately replace its image with the user job; thus, it must `exec()` the user job, not `fork()` it.

For Bourne type shells (*sh*, *bash*, *ksh*), the last line should be:

```
exec "$@"
```

For the C type shells (*csh*, *tcsh*), the last line should be:

```
exec $*:q
```

On Windows, the end should look like:

```
REM set some environment variables
set LICENSE_SERVER=192.168.1.202:5012
set MY_PARAMS=2

REM Run the actual job now
%*
```

This syntax is precise, to correctly handle program arguments which contain white space characters.

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`.

If the wrapper script encounters an error as it runs, and it is unable to run the user job, it is important that the wrapper script indicate this to the HTCondor system so that HTCondor does not assign the exit code of the

wrapper script to the job. To do this, the wrapper script should write a useful error message to the file named in the environment variable _CONDOR_WRAPPER_ERROR_FILE, and then the wrapper script should exit with a non-zero value. If this file is created by the wrapper script, HTCondor assumes that the wrapper script has failed, and HTCondor will place the job back in the queue marking it as Idle, such that the job will again be run. The *condor_starter* will also copy the contents of this error file to the *condor_starter* log, so the administrator can debug the problem.

When a wrapper script is in use, the executable of a job submission may be specified by a relative path, as long as the submit description file also contains:

```
+PreserveRelativeExecutable = True
```

For example,

```
# Let this executable be resolved by user's path in the wrapper
cmd = sleep
+PreserveRelativeExecutable = True
```

Without this extra attribute:

```
# A typical fully-qualified executable path
cmd = /bin/sleep
```

CGROUP_MEMORY_LIMIT_POLICY A string with possible values of `hard`, `soft` and `none`. The default value is `none`. If set to `hard`, the cgroup-based limit on the total amount of physical memory used by the sum of all processes in the job will not be allowed to exceed the limit given by the cgroup memory controller attribute memory.limit_in_bytes. If the processes try to allocate more memory, the allocation will succeed, and virtual memory will be allocated, but no additional physical memory will be allocated. If set to the default value `soft`, the cgroup-based limit on the total amount of physical memory used by the sum of all processes in the job will be allowed to go over the limit, if there is free memory available on the system. If set to `none`, no limit will be enforced, but the memory usage of the job will be accurately measured by a cgroup.

USE_VISIBLE_DESKTOP This boolean variable is only meaningful on Windows machines. If `True`, HTCondor 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 HTCondor. If `False`, HTCondor uses the default behavior of creating a new, non-visible desktop to run the job on. See section 7.2 for details on how HTCondor 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 901). 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*. When the user job and/or `STARTER_JOB_ENVIRONMENT` define an environment variable that is in the *condor_starter*'s environment, the setting from the *condor_starter*'s environment is overridden. This variable does not apply to standard universe jobs.

NAMED_CHROOT A comma and/or space separated list of full paths to one or more directories, under which the *condor_starter* may run a chroot-ed job. This allows HTCondor to invoke `chroot()` before launching a job, if the job requests such by defining the job ClassAd attribute `RequestedChroot` with a directory that matches one in this list. There is no default value for this variable.

STARTER_UPLOAD_TIMEOUT An integer value that specifies the network communication timeout 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.

ASSIGN_CPU_AFFINITY A boolean expression that defaults to `False`. When it evaluates to `True`, each job under this *condor_startd* is confined to using only as many cores as the configured number of slots. When using partitionable slots, each job will be bound to as many cores as requested by specifying **request_cpus**. When `True`, this configuration variable overrides any specification of `ENFORCE_CPU_AFFINITY`. The expression is evaluated in the context of the Job ClassAd.

ENFORCE_CPU_AFFINITY This configuration variable is replaced by `ASSIGN_CPU_AFFINITY`. Do not enable this configuration variable unless using glidein or another unusual setup.

A boolean value that defaults to `False`. When `False`, the CPU affinity of processes in a job is not enforced. When `True`, the processes in an HTCondor job maintain their affinity to a CPU. This means that this job will only run on that particular CPU, even if other CPU cores are idle.

If `True` and `SLOT<N>_CPU_AFFINITY` is not set, the CPU that the job is locked to is the same as `SlotID - 1`. Note that slots are numbered beginning with the value 1, while CPU cores are numbered beginning with the value 0.

When `True`, more fine grained affinities may be specified with `SLOT<N>_CPU_AFFINITY`.

SLOT<N>_CPU_AFFINITY This configuration variable is replaced by `ASSIGN_CPU_AFFINITY`. Do not enable this configuration variable unless using glidein or another unusual setup.

A comma separated list of cores to which an HTCondor job running on a specific slot given by the value of `<N>` show affinity. Note that slots are numbered beginning with the value 1, while CPU cores are numbered beginning with the value 0. This affinity list only takes effect when `ENFORCE_CPU_AFFINITY = True`.

ENABLE_URL_TRANSFERS A boolean value that when `True` causes the *condor_starter* for a job to invoke all plug-ins defined by `FILETRANSFER_PLUGINS` to determine their capabilities for handling protocols to be used in file transfer specified with a URL. When `False`, a URL transfer specified in a job's submit description file will cause an error issued by *condor_submit*. The default value is `True`.

FILETRANSFER_PLUGINS A comma separated list of full and absolute path and executable names for plug-ins that will accomplish the task of doing file transfer when a job requests the transfer of an input file by specifying a URL. See section 3.14.2 for a description of the functionality required of a plug-in.

RUN_FILETRANSFER_PLUGINS_WITH_ROOT A boolean value that affects only Unix platforms and defaults to `False`, causing file transfer plug-ins invoked for a job to run with both the real and the effective UID set to user that the job runs as. The user that the job runs as may be the job owner, `nobody`, or the slot user. The group is set to primary group of the user that the job runs as, and all supplemental groups are dropped. The default gives the behavior exhibited prior to the existence of this configuration variable. When set to `True`, file transfer

plug-ins are invoked with a real UID of 0 (`root`), provided the HTCondor daemons also run as `root`. The effective UID is set to the user that the job runs as.

This configuration variable can permit plug-ins to do privileged operations, such as access a credential protected by file system permissions. The default value is recommended unless privileged operations are required.

ENABLE_CHIRP A boolean value that defaults to `True`. An administrator would set the value to `False` to disable Chirp remote file access from execute machines.

ENABLE_CHIRP_UPDATES A boolean value that defaults to `True`. If `ENABLE_CHIRP` is `True`, and `ENABLE_CHIRP_UPDATES` is `False`, then the user job can only read job attributes from the submit side; it cannot change them or write to the job event log. If `ENABLE_CHIRP` is `False`, the setting of this variable does not matter, as no Chirp updates are allowed in that case.

ENABLE_CHIRP_IO A boolean value that defaults to `True`. If `False`, the file I/O *condor_chirp* commands are prohibited.

ENABLE_CHIRP_DELAYED A boolean value that defaults to `True`. If `False`, the *condor_chirp* commands **get_job_attr_delayed** and **set_job_attr_delayed** are prohibited.

CHIRP_DELAYED_UPDATE_PREFIX This string-valued parameter, which defaults to `"Chirp"`, defines the allowed prefixes for attribute names which can be used with the *condor_chirp* commands **set_job_attribute_delayed** and **get_job_attribute_delayed**. Because it must be set to the same value on both the submit and execute nodes, it is advised that this parameter not be changed from its built-in default.

CHIRP_DELAYED_UPDATE_MAX_ATTRS This integer-valued parameter, which defaults to 100, represents the maximum number of pending delayed chirp updates buffered by the *condor_starter*. If the number of unique attributes updated by the *condor_chirp* command **set_job_attr_delayed** exceeds this parameter, it is possible for these updates to be ignored.

USE_PSS A boolean value, that when `True` causes the *condor_starter* to measure the PSS (Proportional Set Size) of each HTCondor job. The default value is `True`. When running many short lived jobs, performance problems in the *condor_procd* have been observed, and a setting of `False` may relieve these problems.

MEMORY_USAGE_METRIC A ClassAd expression that produces an initial value for the job ClassAd attribute `MemoryUsage` in jobs that are *not* standard universe and *not* vm universe.

MEMORY_USAGE_METRIC_VM A ClassAd expression that produces an initial value for the job ClassAd attribute `MemoryUsage` in vm universe jobs.

STARTER_RLIMIT_AS An integer ClassAd expression, expressed in MiB, evaluated by the *condor_starter* to set the `RLIMIT_AS` parameter of the `setrlimit()` system call. This limits the virtual memory size of each process in the user job. The expression is evaluated in the context of both the machine and job ClassAds, where the machine ClassAd is the `MY.` ClassAd, and the job ClassAd is the `TARGET.` ClassAd. There is no default value for this variable. Since values larger than 2047 have no real meaning on 32-bit platforms, values larger than 2047 result in no limit set on 32-bit platforms.

USE_PID_NAMESPACES A boolean value that, when `True`, enables the use of per job PID namespaces for HTCondor jobs run on Linux kernels. Defaults to `False`.

PER_JOB_NAMESPACES A boolean value that defaults to `False`. Relevant only for Linux platforms using file system namespaces. The default value of `False` ensures that there will be no private mount points, because auto mounts done by *autofs* would use the wrong name for private file system mounts. A `True` value is useful when private file system mounts are permitted and *autofs* (for NFS) is not used.

DYNAMIC_RUN_ACCOUNT_LOCAL_GROUP For Windows platforms, a value that sets the local group to a group other than the default `Users` for the `condor-slot<X>` run account. Do *not* place the local group name within quotation marks.

JOB_EXECDIR_PERMISSIONS Control the permissions on the job's scratch directory. Defaults to `user` which sets permissions to *0700*. Possible values are `user`, `group`, and `world`. If set to `group`, then the directory is group-accessible, with permissions set to *0750*. If set to `world`, then the directory is created with permissions set to *0755*.

STARTER_STATS_LOG The full path and file name of a file that stores TCP statistics for starter file transfers. (Note that the starter logs TCP statistics to this file by default. Adding `D_STATS` to the `STARTER_DEBUG` value will cause TCP statistics to be logged to the normal starter log file (`$(STARTER_LOG)`).) If not defined, `STARTER_STATS_LOG` defaults to `$(LOG)/XferStatsLog`. Setting `STARTER_STATS_LOG` to `/dev/null` disables logging of starter TCP file transfer statistics.

MAX_STARTER_STATS_LOG Controls the maximum size in bytes or amount of time that the starter TCP statistics log will be allowed to grow. If not defined, `MAX_STARTER_STATS_LOG` defaults to `$(MAX_DEFAULT_LOG)`, which currently defaults to 10 MiB in size. Values are specified with the same syntax as `MAX_DEFAULT_LOG`.

SINGULARITY The path to the Singularity binary. The default value is `/usr/bin/singularity`.

SINGULARITY_JOB A boolean value specifying whether this startd should run jobs under Singularity. The default value is `False`.

SINGULARITY_IMAGE_EXPR The path to the Singularity container image file. The default value is `"SingularityImage"`.

SINGULARITY_TARGET_DIR A directory within the Singularity image to which `$_CONDOR_SCRATCH_DIR` on the host should be mapped. The default value is `""`.

SINGULARITY_BIND_EXPR A string value containing a list of bind mount specifications to be passed to Singularity. The default value is `"SingularityBind"`.

3.5.12 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.

JOB_DEFAULT_NOTIFICATION The default that sets email notification for jobs. This variable defaults to `NEVER`, such that HTCondor will not send email about events for jobs. Possible values are `NEVER`, `ERROR`, `ALWAYS`, or `COMPLETE`. If `ALWAYS`, the owner will be notified whenever the job produces a checkpoint, as well as when the

job completes. If COMPLETE, the owner will be notified when the job terminates. If ERROR, the owner will only be notified if the job terminates abnormally, or if the job is placed on hold because of a failure, and not by user request. If NEVER, the owner will not receive email.

JOB_DEFAULT_REQUESTMEMORY The amount of memory in MiB to acquire for a job, if the job does not specify how much it needs using the **request_memory** submit command. If this variable is not defined, then the default is defined by the expression

```
ifThenElse(MemoryUsage =!= UNDEFINED,MemoryUsage,(ImageSize+1023)/1024)
```

JOB_DEFAULT_REQUESTDISK The amount of disk in KiB to acquire for a job, if the job does not specify how much it needs using the **request_disk** submit command. If the job defines the value, then that value takes precedence. If not set, then then the default is defined as DiskUsage.

JOB_DEFAULT_REQUESTCPUS The number of CPUs to acquire for a job, if the job does not specify how many it needs using the **request_cpus** submit command. If the job defines the value, then that value takes precedence. If not set, then then the default is 1.

DEFAULT_JOB_MAX_RETRIES The default value for the maximum number of job retries, if the *condor_submit* retry feature is used. (Note that this value is only relevant if either **retry_until** or **success_exit_code** is defined in the submit file, and **max_retries** is not.) (See section 11 for more information.) The default value if not defined is 10.

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 HTCondor.

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 HTCondor 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, HTCondor will attempt to consolidate small read and write operations into large blocks. This macro specifies the default block size HTCondor 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 **-disable** command-line option is used. This tells *condor_submit* to disable file permission checks when submitting a job for read permissions on all input files, such as those defined by commands **input** and **transfer_input_files**, as well as write permission to output files, such as a log file defined by **log** and output files defined with **output** or **transfer_output_files**. 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_ATTRS A comma-separated and/or space-separated list of ClassAd attribute names for which the attribute and value will be inserted into all the job ClassAds that *condor_submit* creates. In this way, it is like the "+" syntax in a submit description file. Attributes defined in the submit description file with "+" will override attributes defined in the configuration file with SUBMIT_ATTRS. Note that adding an attribute to a job's ClassAd will *not* function as a method for specifying default values of submit description file commands forgotten in a job's submit description file. The command in the submit description file results in actions by *condor_submit*, while the use of SUBMIT_ATTRS adds a job ClassAd attribute at a later point in time. SUBMIT_EXPRS is a historic setting that functions identically to SUBMIT_ATTRS. It may be removed in the future, so use SUBMIT_ATTRS.

LOG_ON_NFS_IS_ERROR A boolean value that controls whether *condor_submit* prohibits job submit description files with job event 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.

ENABLE_DEPRECATION_WARNINGS A boolean value that defaults to `False`. When `True`, *condor_submit* issues warnings when a job requests features that are no longer supported.

INTERACTIVE_SUBMIT_FILE The path and file name of a submit description file that *condor_submit* will use in the specification of an interactive job. The default is `$(RELEASE_DIR)/libexec/interactive.sub` when not defined.

CRED_MIN_TIME_LEFT When a job uses an X509 user proxy, condor_submit will refuse to submit a job whose x509 expiration time is less than this many seconds in the future. The default is to only refuse jobs whose expiration time has already passed.

3.5.13 condor_preen Configuration File Entries

These macros affect *condor_preen*.

PREEN_ADMIN 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 A comma or space separated list of files that *condor_preen* considers valid files to find in the `$(SPOOL)` directory, such that *condor_preen* will not remove these files. There is no default value. *condor_preen* will add to the list files and directories that are normally present in the `$(SPOOL)` directory. A single asterisk (`*`) wild card character is permitted in each file item within the list.

SYSTEM_VALID_SPOOL_FILES A comma or space separated list of files that *condor_preen* considers valid files to find in the `$(SPOOL)` directory. The default value is all files known by HTCondor to be valid. This variable exists such that it can be queried; it should not be changed. *condor_preen* use it to initialize the the list files and directories that are normally present in the `$(SPOOL)` directory. A single asterisk (`*`) wild card character is permitted in each file item within the list.

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.

3.5.14 condor_collector Configuration File Entries

These macros affect the *condor_collector*.

CLASSAD_LIFETIME The default maximum age in seconds for ClassAds collected by the *condor_collector*. ClassAds older than the maximum age are discarded by the *condor_collector* as stale.

If present, the ClassAd attribute `ClassAdLifetime` specifies the ClassAd's lifetime in seconds. If `ClassAdLifetime` is not present in the ClassAd, the *condor_collector* will use the value of

`$(CLASSAD_LIFETIME)`. This variable is defined in terms of seconds, and it defaults to 900 seconds (15 minutes).

To ensure that the *condor_collector* does not miss any ClassAds, the frequency at which all other subsystems that report using an update interval must be tuned. The configuration variables that set these subsystems are

- `UPDATE_INTERVAL` (for the *condor_startd* daemon)
- `NEGOTIATOR_UPDATE_INTERVAL`
- `SCHEDD_INTERVAL`
- `MASTER_UPDATE_INTERVAL`
- `CKPT_SERVER_INTERVAL`
- `DEFRAG_UPDATE_INTERVAL`
- `HAD_UPDATE_INTERVAL`

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 HTCondor 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, HTCondor 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 Center for High Throughput Computing at the University of Wisconsin-Madison. The Center for High Throughput Computing uses these weekly status messages in order to have some idea as to how many HTCondor pools exist in the world. We appreciate getting the reports, as this is one way we can convince funding agencies that HTCondor is being used in the real world. If you do not wish this information to be sent to the Center for High Throughput Computing, 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, HTCondor follows its default behavior.

COLLECTOR_NAME 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 HTCondor 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 HTCondor 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.

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 the pool. Updates include only the number of total machines, the number of jobs submitted, the number of machines running jobs, the host name of the central manager, and the $(COLLECTOR_NAME). These updates help the Center for High Throughput Computing see how HTCondor is being used around the world. By default, they will be sent to condor.cs.wisc.edu. To discontinue sending updates, explicitly set this macro to NONE. If undefined or commented out in the configuration file, HTCondor follows its default behavior.

COLLECTOR_UPDATE_INTERVAL This variable is defined in seconds and defaults to 900 (every 15 minutes). It controls the frequency of the periodic updates sent to a central *condor_collector* at UW-Madison as defined by CONDOR_DEVELOPERS_COLLECTOR.

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.

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 MB).

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 A boolean value that controls whether or not the *condor_collector* daemon keeps update statistics on incoming updates. The default value is `True`. If enabled, the *condor_collector* will insert several attributes into the 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 HTCondor daemons which publish ClassAds publish these attributes).

The attributes inserted are `UpdatesTotal`, `UpdatesSequenced`, and `UpdatesLost`. `UpdatesTotal` is the total number of updates (of this ClassAd type) the *condor_collector* has received from this host. `UpdatesSequenced` is the number of updates that the *condor_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 *condor_collector* has detected as being lost. See page 1040 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 variable controls the size of the published update history that the *condor_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 variable's value is ignored, if `COLLECTOR_DAEMON_STATS` is not enabled.

If the value is a non-zero one, the *condor_collector* will insert attribute `UpdatesHistory` into the ClassAd (similar to `UpdatesTotal`). AttrUpdatesHistory 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 was 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 1040 for more information on the added attribute.

COLLECTOR_CLASS_HISTORY_SIZE This variable controls the size of the published update history that the *condor_collector* inserts into the *condor_collector* ClassAds it produces. The default value is zero.

If this variable has a non-zero value, the *condor_collector* will insert `UpdatesClassHistory` into the *condor_collector* ClassAd (similar to `UpdatesHistory`). These are added per class of ClassAd, however. The classes refer to the type of ClassAds. Additionally, there is a Total class created, which represents the history of all ClassAds that this *condor_collector* receives.

Note that the *condor_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.

COLLECTOR_QUERY_WORKERS This variable sets the maximum number of worker processes that the *condor_collector* can have. When receiving a query request, the Unix *condor_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 variable is ignored on Windows, and its default value is zero. The default configuration, however, has a value of 2.

COLLECTOR_DEBUG This macro (and other macros related to debug logging in the *condor_collector* is described in section 3.5.2 as `<SUBSYS>_DEBUG`.

CONDOR_VIEW_CLASSAD_TYPES Provides the ClassAd types that will be forwarded to the `CONDOR_VIEW_HOST`. The ClassAd types can be found with *condor_status* **-any**. The default forwarding behavior of the *condor_collector* is equivalent to

```
CONDOR_VIEW_CLASSAD_TYPES=Machine,Submitter
```

There is no default value for this variable.

COLLECTOR_FORWARD_FILTERING When this boolean variable is set to `True`, Machine and Submitter ad updates are not forwarded to the `CONDOR_VIEW_HOST` if certain attributes are unchanged from the previous update of the ad. The default is `False`, meaning all updates are forwarded.

COLLECTOR_FORWARD_WATCH_LIST When `COLLECTOR_FORWARD_FILTERING` is set to `True`, this variable provides the list of attributes that controls whether a Machine or Submitter ad update is forwarded to the `CONDOR_VIEW_HOST`. If all attributes in this list are unchanged from the previous update, then the new update is not forwarded. The default value is `State,Cpus,Memory,IdleJobs`.

COLLECTOR_FORWARD_INTERVAL When `COLLECTOR_FORWARD_FILTERING` is set to `True`, this variable limits how long forwarding of updates for a given ad can be filtered before an update must be forwarded. The default is one third of `CLASSAD_LIFETIME`.

The following macros control where, when, and for how long HTCondor persistently stores absent ClassAds. See section 3.12.2 on page 455 for more details.

ABSENT_REQUIREMENTS A boolean expression evaluated by the *condor_collector* when a machine ClassAd would otherwise expire. If `True`, the ClassAd instead becomes absent. If not defined, the implementation will behave as if `False`, and no absent ClassAds will be stored.

ABSENT_EXPIRE_ADS_AFTER The integer number of seconds after which the *condor_collector* forgets about an absent ClassAd. If 0, the ClassAds persist forever. Defaults to 30 days.

COLLECTOR_PERSISTENT_AD_LOG The full path and file name of a file that stores machine ClassAds for every hibernating or absent machine. This forms a persistent storage of these ClassAds, in case the *condor_collector* daemon crashes.

To avoid *condor_preen* 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`.

This configuration variable replaces `OFFLINE_LOG`, which is no longer used.

EXPIRE_INVALIDATED_ADS A boolean value that defaults to `False`. When `True`, causes all invalidated ClassAds to be treated as if they expired. This permits invalidated ClassAds to be marked absent, as defined in section 3.12.2.

3.5.15 condor_negotiator Configuration File Entries

These macros affect the *condor_negotiator*.

NEGOTIATOR_INTERVAL Sets how often the *condor_negotiator* starts a negotiation cycle. It is defined in seconds and defaults to 60 (1 minute).

NEGOTIATOR_UPDATE_INTERVAL This macro determines how often the *condor_negotiator* daemon sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 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 HTCondor experts.

NEGOTIATOR_TIMEOUT Sets the timeout that the negotiator uses on its network connections to the *condor_schedd* and *condor_startd*s. It is defined in seconds and defaults to 30.

NEGOTIATION_CYCLE_STATS_LENGTH Specifies how many recent negotiation cycles should be included in the history that is published in the *condor_negotiator*'s ad. The default is 3 and the maximum allowed value is 100. Setting this value to 0 disables publication of negotiation cycle statistics. The statistics about recent cycles are stored in several attributes per cycle. Each of these attribute names will have a number appended to it to indicate how long ago the cycle happened, for example: `LastNegotiationCycleDuration0`, `LastNegotiationCycleDuration1`, `LastNegotiationCycleDuration2`, ... The attribute numbered 0 applies to the most recent negotiation cycle. The attribute numbered 1 applies to the next most recent negotiation cycle, and so on. See page 1033 for a list of attributes that are published.

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 Sets the priority factor for local users as they first submit jobs, as described in section 3.6. Defaults to 1000.

NICE_USER_PRIO_FACTOR Sets the priority factor for nice users, as described in section 3.6. Defaults to 10000000000.

REMOTE_PRIO_FACTOR Defines the priority factor for remote users, which are those users who who do not belong to the local domain. See section 3.6 for details. Defaults to 10000000.

ACCOUNTANT_LOCAL_DOMAIN Describes the local UID domain. This variable is used to decide if a user is local or remote. A user is considered to be in the local domain if their UID domain matches the value of this variable. Usually, this variable is set to the local UID domain. If 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_schedd*s 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 `False`. When this is set to the default value of `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 3.5.1 on page 214 for details), the *condor_negotiator* should be configured not to attempt to contact these *condor_startd* daemons by using the default value.

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 explicitly set to be undefined, this expression has no effect on the ranking of matches. The default value prefers to match multi-core jobs to dynamic slots in a best fit manner:

```
NEGOTIATOR_PRE_JOB_RANK = (10000000 * My.Rank) + \
 (1000000 * (RemoteOwner =?= UNDEFINED)) - (100000 * Cpus) - Memory
```

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

```
NEGOTIATOR_POST_JOB_RANK = \
 (RemoteOwner =?= UNDEFINED) * \
 (ifThenElse(isUndefined(KFlops), 1000, Kflops) - \
 SlotID - 1.0e10*(Offline=?=True))
```

PREEMPTION_REQUIREMENTS When considering user priorities, the negotiator will not preempt a job running on a given machine unless this 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 HTCondor configuration file, the default value for this expression is `False`. `PREEMPTION_REQUIREMENTS` should include the term `(SubmitterGroup =?= RemoteGroup)`, if a preemption policy that respects *group quotas* is desired. Note that this variable does not influence other potential causes of preemption, such as the `RANK` of the *condor_startd*, or `PREEMPT` expressions. See section 3.7.1 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`.

PREEMPTION_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. 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 value first considers the user's priority and chooses the user with the worst priority. Then, among the running jobs of that user, it chooses the job with the least accumulated run time:

```
PREEMPTION_RANK = (RemoteUserPrio * 1000000) - \
 ifThenElse(isUndefined(TotalJobRunTime), 0, TotalJobRunTime)
```

PREEMPTION_RANK_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_SLOT_CONSTRAINT An expression which constrains which machine ClassAds are fetched from the *condor_collector* by the *condor_negotiator* during a negotiation cycle.

NEGOTIATOR_TRIM_SHUTDOWN_THRESHOLD This setting is not likely to be customized, except perhaps within a glidein setting. An integer expression that evaluates to a value within the context of the *condor_negotiator* ClassAd, with a default value of 0. When this expression evaluates to an integer X greater than 0, the *condor_negotiator* will not make matches to machines that contain the ClassAd attribute `DaemonShutdown` which evaluates to `True`, when that shut down time is X seconds into the future. The idea here is a mechanism to prevent matching with machines that are quite close to shutting down, since the match would likely be a waste of time.

NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT or GROUP_DYNAMIC_MACH_CONSTRAINT This optional expression specifies which machine ClassAds should be counted when computing the size of the pool. It applies both for group quota allocation and when there are no groups. The default is to count all machine ClassAds. When extra slots exist for special purposes, as, for example, suspension slots or file transfer slots, this expression

can be used to inform the *condor_negotiator* that only normal slots should be counted when computing how big each group's share of the pool should be.

The name NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT replaces GROUP_DYNAMIC_MACH_CONSTRAINT as of HTCondor version 7.7.3. Using the older name causes a warning to be logged, although the behavior is unchanged.

NEGOTIATOR_DEBUG This macro (and other settings related to debug logging in the negotiator) is described in section 3.5.2 as <SUBSYS>_DEBUG.

NEGOTIATOR_MAX_TIME_PER_SUBMITTER The maximum number of seconds the *condor_negotiator* will spend with each individual submitter during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will skip over requests from this submitter until the next negotiation cycle. It defaults to the number of seconds in one year.

NEGOTIATOR_MAX_TIME_PER_SCHEDD The maximum number of seconds the *condor_negotiator* will spend with each individual *condor_schedd* during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will skip over requests from this *condor_schedd* until the next negotiation cycle. It defaults to the number of seconds in one year.

NEGOTIATOR_MAX_TIME_PER_CYCLE The maximum number of seconds the *condor_negotiator* will spend in total across all submitters during one negotiation cycle. Once this time limit has been reached, the *condor_negotiator* will skip over requests from all submitters until the next negotiation cycle. It defaults to the number of seconds in one year.

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 3.6.6 on page 347 for a description of a pie slice. It defaults to the number of seconds in one year.

USE_RESOURCE_REQUEST_COUNTS A boolean value that defaults to True. When True, the latency of negotiation will be reduced when there are many jobs next to each other in the queue with the same auto cluster, and many matches are being made. When True, the *condor_schedd* tells the *condor_negotiator* to send X matches at a time, where X equals number of consecutive jobs in the queue within the same auto cluster.

NEGOTIATOR_RESOURCE_REQUEST_LIST_SIZE An integer tuning parameter used by the *condor_negotiator* to control the number of resource requests fetched from a *condor_schedd* per network round-trip. With higher values, the latency of negotiation can be significantly be reduced when negotiating with a *condor_schedd* running HTCondor version 8.3.0 or more recent, especially over a wide-area network. Setting this value too high, however, could cause the *condor_schedd* to unnecessarily block on network I/O. The default value is 20. If USE_RESOURCE_REQUEST_COUNTS is set to False, then this variable will be unconditionally set to a value of 1.

NEGOTIATOR_MATCH_EXPRS A comma-separated 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 does not 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 *condor_negotiator* will contain the following attribute when they are sent to the *condor_startd*:

```
NegotiatorMatchExprNegotiatorName = "My Negotiator"
```

The expressions inserted by the *condor_negotiator* may be useful in *condor_startd* policy expressions, when the *condor_startd* belongs to multiple HTCondor pools.

NEGOTIATOR_MATCHLIST_CACHING A boolean value that defaults to `True`. When `True`, it enables an optimization in the *condor_negotiator* that works with auto clustering. In determining the sorted list of machines that a job might use, the job goes to the first machine off the top of the list. If `NEGOTIATOR_MATCHLIST_CACHING` is `True`, and if the next job is part of the same auto cluster, meaning that it is a very similar job, the *condor_negotiator* will reuse the previous list of machines, instead of recreating the list from scratch.

If matching grid resources, and the desire is for a given resource to potentially match multiple times per *condor_negotiator* pass, `NEGOTIATOR_MATCHLIST_CACHING` should be `False`. See section 5.3.10 on page 579 in the subsection on Advertising Grid Resources to HTCondor for an example.

NEGOTIATOR_CONSIDER_PREEMPTION For expert users only. A boolean value that defaults to `True`. When `False`, it can cause the *condor_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`.*

NEGOTIATOR_CONSIDER_EARLY_PREEMPTION A boolean value that when `False` (the default), prevents the *condor_negotiator* from matching jobs to claimed slots that cannot immediately be preempted due to `MAXJOBRETIREMENTTIME`.

ALLOW_PSLOT_PREEMPTION A boolean value that defaults to `False`. When set to `True` for the *condor_negotiator*, it enables a new matchmaking mode in which one or more dynamic slots can be preempted in order to make enough resources available in their parent partitionable slot for a job to successfully match to the partitionable slot.

STARTD_AD_REEVAL_EXPR A boolean value evaluated in the context of each machine ClassAd within a negotiation cycle that determines whether the ClassAd from the *condor_collector* is to replace the stashed ClassAd utilized during the previous negotiation cycle. When `True`, the ClassAd from the *condor_collector* does replace the stashed one. When not defined, the default value is to replace the stashed ClassAd if the stashed ClassAd's sequence number is older than its potential replacement.

NEGOTIATOR_UPDATE_AFTER_CYCLE A boolean value that defaults to `False`. When `True`, it will force the *condor_negotiator* daemon to publish an update to the *condor_collector* at the end of every negotiation cycle. This is useful if monitoring statistics for the previous negotiation cycle.

NEGOTIATOR_READ_CONFIG_BEFORE_CYCLE A boolean value that defaults to `False`. When `True`, the *condor_negotiator* will re-read the configuration prior to beginning each negotiation cycle. Note that this operation will update configured behaviors such as concurrency limits, but not data structures constructed during a full reconfiguration, such as the group quota hierarchy. A full reconfiguration, for example as accomplished with *condor_reconfig*, remains the best way to guarantee that all *condor_negotiator* configuration is completely updated.

<NAME>_LIMIT An integer value that defines the amount of resources available for jobs which declare that they use some consumable resource as described in section 3.14.15. <Name> is a string invented to uniquely describe the resource.

CONCURRENCY_LIMIT_DEFAULT An integer value that describes the number of resources available for any resources that are not explicitly named defined with the configuration variable <NAME>_LIMIT. If not defined, no limits are set for resources not explicitly identified using <NAME>_LIMIT.

CONCURRENCY_LIMIT_DEFAULT_<NAME> If set, this defines a default concurrency limit for all resources that start with <NAME>.

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 floating point value to represent a static quota specifying an integral number of machines for the hierarchical group identified by <groupname>. It is meaningless to specify a non integer value, since only integral numbers of machines can be allocated. Example:

```
GROUP_QUOTA_group_physics = 20
GROUP_QUOTA_group_chemistry = 10
```

When both static and dynamic quotas are defined for a specific group, the static quota is used and the dynamic quota is ignored.

GROUP_QUOTA_DYNAMIC_<groupname> A floating point value in the range 0.0 to 1.0, inclusive, representing a fraction of a pool's machines (slots) set as a dynamic quota for the hierarchical group identified by <groupname>. For example, the following specifies that a quota of 25% of the total machines are reserved for members of the group_biology group.

```
GROUP_QUOTA_DYNAMIC_group_biology = 0.25
```

The group name must be specified in the GROUP_NAMES list.

This section has not yet been completed

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. The user name that is used for accounting and prioritization purposes is still the group user as specified by AccountingGroup in the job ClassAd.

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.

GROUP_ACCEPT_SURPLUS A boolean value that, when True, specifies that groups should be allowed to use more than their configured quota when there is not enough demand from other groups to use all of the available machines. The default value is False.

GROUP_ACCEPT_SURPLUS_<groupname> A boolean value applied as a group-specific version of GROUP_ACCEPT_SURPLUS. When not specified, the value of GROUP_ACCEPT_SURPLUS applies to the named group.

GROUP_QUOTA_ROUND_ROBIN_RATE The maximum sum of weighted slots that should be handed out to an individual submitter in each iteration within a negotiation cycle. If slot weights are not being used by the *condor_negotiator*, as specified by NEGOTIATOR_USE_SLOT_WEIGHTS = False, then this value is just the (unweighted) number of slots. The default value is a very big number, effectively infinite. Setting the value to a number smaller than the size of the pool can help avoid starvation. An example of the starvation problem is when there are a subset of machines in a pool with large memory, and there are multiple job submitters who desire all of these machines. Normally, HTCondor will decide how much of the full pool each person should get, and then attempt to hand out that number of resources to each person. Since the big memory machines are only a subset of pool, it may happen that they are all given to the first person contacted, and the remainder requiring large memory machines get nothing. Setting GROUP_QUOTA_ROUND_ROBIN_RATE to a value that is small compared to the size of subsets of machines will reduce starvation at the cost of possibly slowing down the rate at which resources are allocated.

GROUP_QUOTA_MAX_ALLOCATION_ROUNDS An integer that specifies the maximum number of times within one negotiation cycle the *condor_negotiator* will calculate how many slots each group deserves and attempt to allocate them. The default value is 3. The reason it may take more than one round is that some groups may not have jobs that match some of the available machines, so some of the slots that were withheld for those groups may not get allocated in any given round.

NEGOTIATOR_USE_SLOT_WEIGHTS A boolean value with a default of `True`. When `True`, the *condor_negotiator* pays attention to the machine ClassAd attribute `SlotWeight`. When `False`, each slot effectively has a weight of 1.

NEGOTIATOR_USE_WEIGHTED_DEMAND A boolean value that defaults to `True`. When `False`, the behavior is the same as for HTCondor versions prior to 7.9.6. If `True`, when the *condor_schedd* advertises `IdleJobs` in the submitter ClassAd, which represents the number of idle jobs in the queue for that submitter, it will also advertise the total number of requested cores across all idle jobs from that submitter, `WeightedIdleJobs`. If partitionable slots are being used, and if hierarchical group quotas are used, and if any hierarchical group quotas set `GROUP_ACCEPT_SURPLUS` to `True`, and if configuration variable `SlotWeight` is set to the number of cores, then setting this configuration variable to `True` allows the amount of surplus allocated to each group to be calculated correctly.

GROUP_SORT_EXPR A floating point ClassAd expression that controls the order in which the *condor_negotiator* considers groups when allocating resources. The smallest magnitude positive value goes first. The default value is set such that group `<none>` always goes last when considering group quotas, and groups are considered in starvation order (the group using the smallest fraction of its resource quota is considered first).

NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION A boolean value that defaults to `True`. When `True`, the behavior of resource allocation when considering groups is more like it was in the 7.4 stable series of HTCondor. In implementation, when `True`, the static quotas of subgroups will *not* be scaled when the sum of these static quotas of subgroups sums to more than the group's static quota. This behavior is desirable when using static quotas, unless the sum of subgroup quotas is considerably less than the group's quota, as scaling is currently based on the number of machines available, not assigned quotas (for static quotas).

3.5.16 condor_procd Configuration File Macros

USE_PROCD This boolean variable determines whether the *condor_procd* will be used for managing process families. If the *condor_procd* is not used, each daemon will run the process family tracking logic on its own. Use of the *condor_procd* results in improved scalability because only one instance of this logic is required. The *condor_procd* is required when using group ID-based process tracking (see Section 3.14.11). In this case, the `USE_PROCD` setting will be ignored and a *condor_procd* will always be used. By default, the *condor_master* will start a *condor_procd* that all other daemons that need process family tracking will use. A daemon that uses the *condor_procd* will start a *condor_procd* for use by itself and all of its child daemons.

PROCD_MAX_SNAPSHOT_INTERVAL This setting determines the maximum time that the *condor_procd* will wait between probes of the system for information about the process families it is tracking.

PROCD_LOG Specifies a log file for the *condor_procd* to use. Note that by design, the *condor_procd* does not include most of the other logic that is shared amongst the various HTCondor daemons. This means that the *condor_procd* does not include the normal HTCondor logging subsystem, and thus multiple debug levels are not supported. `PROCD_LOG` defaults to `$(LOG)/ProcLog`. Note that enabling `D_PROCFAMILY` in the debug level for any other daemon will cause it to log all interactions with the *condor_procd*.

MAX_PROCD_LOG Controls the maximum length in bytes to which the *condor_procd* 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` file is

overwritten each time the log is saved, thus the maximum space devoted to logging will be twice the maximum length of this log file. A value of 0 specifies that the file may grow without bounds. The default is 10 MiB.

PROCD_ADDRESS This specifies the address that the *condor_procd* will use to receive requests from other HTCondor 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 and distribution: `$(LOCK)/procd_pipe` or `$(RUN)/procd_pipe` on Unix and `\\.\pipe\procd_pipe` on Windows.

USE_GID_PROCESS_TRACKING A boolean value that defaults to `False`. When `True`, a job's initial process is assigned a dedicated GID which is further used by the *condor_procd* to reliably track all processes associated with a job. When `True`, values for `MIN_TRACKING_GID` and `MAX_TRACKING_GID` must also be set, or HTCondor will abort, logging an error message. See section 3.14.11 on page 485 for a detailed description.

MIN_TRACKING_GID An integer value, that together with `MAX_TRACKING_GID` specify a range of GIDs to be assigned on a per slot basis for use by the *condor_procd* in tracking processes associated with a job. See section 3.14.11 on page 485 for a detailed description.

MAX_TRACKING_GID An integer value, that together with `MIN_TRACKING_GID` specify a range of GIDs to be assigned on a per slot basis for use by the *condor_procd* in tracking processes associated with a job. See section 3.14.11 on page 485 for a detailed description.

BASE_CGROUP The path to the directory used as the virtual file system for the implementation of Linux kernel cgroups. This variable defaults to the string `htcondor`, and is only used on Linux systems. To disable cgroup tracking, define this to an empty string. See section 3.14.12 on page 486 for a description of cgroup-based process tracking.

3.5.17 condor_credd Configuration File Macros

These macros affect the *condor_credd*.

CREDD_HOST The host name of the machine running the *condor_credd* daemon.

CREDD_POLLING_TIMEOUT An integer value that determines how long the *condor_credd* daemon will poll for credentials in seconds. The default value is 20.

CREDD_CACHE_LOCALLY A boolean value that defaults to `False`. When `True`, the first successful password fetch operation to the *condor_credd* 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_credd* daemon.

SKIP_WINDOWS_LOGON_NETWORK A boolean value that defaults to `False`. When `True`, Windows authentication skips trying authentication with the `LOGON_NETWORK` method first, and attempts authentication with `LOGON_INTERACTIVE` method. This can be useful if many authentication failures are noticed, potentially leading to users getting locked out.

3.5.18 condor_gridmanager Configuration File Entries

These macros affect the *condor_gridmanager*.

GRIDMANAGER_LOG Defines the path and file name for the log of the *condor_gridmanager*. The owner of the file is the `condor` user.

GRIDMANAGER_CHECKPROXY_INTERVAL The number of seconds between checks for an updated X509 proxy credential. The default is 10 minutes (600 seconds).

GRIDMANAGER_PROXY_REFRESH_TIME For GRAM jobs, the *condor_gridmanager* will not forward a refreshed proxy until the lifetime left for the proxy on the remote machine falls below this value. The value is in seconds and the default is 21600 (6 hours).

GRIDMANAGER_MINIMUM_PROXY_TIME 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, HTCondor-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 HTCondor-G will periodically check to see if the job's proxy has been refreshed, at which point management of the job will resume.

GRIDMANAGER_CONTACT_SCHEDD_DELAY The minimum number of seconds between connections to the *condor_schedd*. The default is 5 seconds.

GRIDMANAGER_JOB_PROBE_INTERVAL The number of seconds between active probes for the status of a submitted job. The default is 1 minute (60 seconds). Intervals specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example

```
GRIDMANAGER_JOB_PROBE_INTERVAL_GT5 = 300
```

GRIDMANAGER_JOB_PROBE_RATE The maximum number of job status probes per second that will be issued to a given remote resource. The time between status probes for individual jobs may be lengthened beyond `GRIDMANAGER_JOB_PROBE_INTERVAL` to enforce this rate. The default is 5 probes per second. Rates specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example

```
GRIDMANAGER_JOB_PROBE_RATE_GT5 = 15
```

GRIDMANAGER_RESOURCE_PROBE_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.

GRIDMANAGER_RESOURCE_PROBE_DELAY The number of seconds between pings of a remote resource that is currently down. The default is 5 minutes (300 seconds).

GRIDMANAGER_EMPTY_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, for which `JobStatus` is anything other than Held. Defaults to 300 seconds.

GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE An integer value that limits the number of jobs that a *condor_gridmanager* daemon will submit to a resource. A comma-separated list of pairs that follows this integer limit will specify limits for specific remote resources. Each pair is a host name and the job limit for that host. Consider the example:

```
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE = 200, foo.edu, 50, bar.com, 100
```

In this example, all resources have a job limit of 200, except foo.edu, which has a limit of 50, and bar.com, which has a limit of 100.

Limits specific to grid types can be set by appending the name of the grid type to the configuration variable name, as the example

```
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_CREAM = 300
```

In this example, the job limit for all CREAM resources is 300. Defaults to 1000.

GRIDMANAGER_MAX_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 head note, 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.

GAHP The full path to the binary of the GAHP server. This configuration variable is no longer used. Use `GT2_GAHP` at section 3.5.18 instead.

GAHP_ARGS Arguments to be passed to the GAHP server. This configuration variable is no longer used.

GAHP_DEBUG_HIDE_SENSITIVE_DATA A boolean value that determines when sensitive data such as security keys and passwords are hidden, when communication to or from a GAHP server is written to a daemon log. The default is `True`, hiding sensitive data.

GRIDMANAGER_GAHP_CALL_TIMEOUT The number of seconds after which a pending GAHP command should time out. The default is 5 minutes (300 seconds).

GRIDMANAGER_GAHP_RESPONSE_TIMEOUT The *condor_gridmanager* will assume a GAHP is hung if this many seconds pass without a response. The default is 20.

GRIDMANAGER_MAX_PENDING_REQUESTS The maximum number of GAHP commands that can be pending at any time. The default is 50.

GRIDMANAGER_CONNECT_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.

GRIDMANAGER_GLOBUS_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).

EC2_RESOURCE_TIMEOUT The number of seconds after which if an EC2 grid universe job fails to ping the EC2 service, the job will be put on hold. Defaults to -1, which implements an infinite length, such that a failure to ping the service will never put the job on hold.

EC2_GAHP_RATE_LIMIT The minimum interval, in whole milliseconds, between requests to the same EC2 service with the same credentials. Defaults to 100.

GRAM_VERSION_DETECTION A boolean value that defaults to `True`. When `True`, the *condor_gridmanager* treats grid types `gt2` and `gt5` identically, and queries each server to determine which protocol it is using. When `False`, the *condor_gridmanager* trusts the grid type provided in job attribute `GridResource`, and treats the server accordingly. Beware that identifying a `gt2` server as `gt5` can result in overloading the server, if a large number of jobs are submitted.

BATCH_GAHP_CHECK_STATUS_ATTEMPTS The number of times a failed status command issued to the *batch_gahp* should be retried. These retries allow the *condor_gridmanager* to tolerate short-lived failures of the underlying batch system. The default value is 5.

C_GAHP_LOG The complete path and file name of the HTCondor GAHP server's log. The default value is `/tmp/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 HTCondor GAHP worker process' log. The default value is `/temp/CGAHPWorkerLog.$(USERNAME)`.

C_GAHP_CONTACT_SCHEDD_DELAY The number of seconds that the *condor_C-gahp* daemon waits between consecutive connections to the remote *condor_schedd* in order to send batched sets of commands to be executed on that remote *condor_schedd* daemon. The default value is 5.

GLITE_LOCATION The complete path to the directory containing the Glite software. The default value is `$(LIBEXEC)/glite`. The necessary Glite software is included with HTCondor, and is required for grid-type batch jobs.

CONDOR_GAHP The complete path and file name of the HTCondor GAHP executable. The default value is `$(SBIN)/condor_c-gahp`.

EC2_GAHP The complete path and file name of the EC2 GAHP executable. The default value is `$(SBIN)/ec2_gahp`.

GT2_GAHP The complete path and file name of the GT2 GAHP executable. The default value is `$(SBIN)/gahp_server`.

BATCH_GAHP The complete path and file name of the batch GAHP executable, to be used for PBS, LSF, SGE, and similar batch systems. The default location is `$(GLITE_LOCATION)/bin/batch_gahp`.

PBS_GAHP The complete path and file name of the PBS GAHP executable. The use of the configuration variable BATCH_GAHP is preferred and encouraged, as this variable may no longer be supported in a future version of HTCondor. A value given with this configuration variable will override a value specified by BATCH_GAHP, and the value specified by BATCH_GAHP is the default if this variable is not defined.

LSF_GAHP The complete path and file name of the LSF GAHP executable. The use of the configuration variable BATCH_GAHP is preferred and encouraged, as this variable may no longer be supported in a future version of HTCondor. A value given with this configuration variable will override a value specified by BATCH_GAHP, and the value specified by BATCH_GAHP is the default if this variable is not defined.

UNICORE_GAHP The complete path and file name of the wrapper script that invokes the Unicore GAHP executable. The default value is $(SBIN)/unicore_gahp.

NORDUGRID_GAHP The complete path and file name of the wrapper script that invokes the NorduGrid GAHP executable. The default value is $(SBIN)/nordugrid_gahp.

CREAM_GAHP The complete path and file name of the CREAM GAHP executable. The default value is $(SBIN)/cream_gahp.

SGE_GAHP The complete path and file name of the SGE GAHP executable. The use of the configuration variable BATCH_GAHP is preferred and encouraged, as this variable may no longer be supported in a future version of HTCondor. A value given with this configuration variable will override a value specified by BATCH_GAHP, and the value specified by BATCH_GAHP is the default if this variable is not defined.

GCE_GAHP The complete path and file name of the GCE GAHP executable. The default value is $(SBIN)/gce_gahp.

BOINC_GAHP The complete path and file name of the BOINC GAHP executable. The default value is $(SBIN)/boinc_gahp.

3.5.19 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. The enclosing square brackets are optional.

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 an HTCondor 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_LOCK This specifies the name of a lock file that is used to ensure that multiple instances of condor_job_router never run with the same `JOB_ROUTER_NAME`. Multiple instances running with the same name could lead to mismanagement of routed jobs. The default value is `$(LOCK)/$(JOB_ROUTER_NAME)Lock`.

JOB_ROUTER_SOURCE_JOB_CONSTRAINT Specifies a global `Requirements` expression that must be true for all newly routed jobs, in addition to any `Requirements` specified within a routing table entry. In addition to the configurable constraints, the *condor_job_router* also has some hard-coded constraints. It avoids recursively routing jobs by requiring that the job's attribute `RoutedBy` does not match `JOB_ROUTER_NAME`. When not running as root, it also avoids routing jobs belonging to other users.

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 on the same machine. Each instance must have a different name, or all but the first to start up will refuse to run. The default is `"jobrouter"`.

Changing this value when routed jobs already exist is not currently gracefully handled. However, it can be done if one also uses *condor_qedit* to change the value of `ManagedManager` and `RoutedBy` from the old name to the new name. The following commands may be helpful:

```
condor_qedit -constraint 'RoutedToJobId =!= undefined && \
  ManagedManager == "insert_old_name"' \
  ManagedManager '"insert_new_name"'
condor_qedit -constraint 'RoutedBy == "insert_old_name"' \
  RoutedBy '"insert_new_name"'
```

JOB_ROUTER_RELEASE_ON_HOLD A boolean value that defaults to `True`. It controls how the *condor_job_router* handles the routed copy when it goes on hold. When `True`, the *condor_job_router* leaves the original job ClassAd in the same state as when claimed. When `False`, the *condor_job_router* does not attempt to reset the original job ClassAd to a pre-claimed state upon yielding control of the job.

JOB_ROUTER_SCHEDD1_SPOOL The path to the spool directory for the *condor_schedd* serving as the source of jobs for routing. If not specified, this defaults to `$(SPOOL)`. If specified, this parameter must point to the spool directory of the *condor_schedd* identified by `JOB_ROUTER_SCHEDD1_NAME`.

JOB_ROUTER_SCHEDD2_SPOOL The path to the spool directory for the *condor_schedd* to which the routed copy of the jobs are submitted. If not specified, this defaults to `$(SPOOL)`. If specified, this parameter must point to the spool directory of the *condor_schedd* identified by `JOB_ROUTER_SCHEDD2_NAME`. Note that when *condor_job_router* is running as `root` and is submitting routed jobs to a different *condor_schedd* than the source *condor_schedd*, it is required that *condor_job_router* have permission to impersonate the job owners of the routed jobs. It is therefore usually necessary to configure `QUEUE_SUPER_USER_MAY_IMPERSONATE` in the configuration of the target *condor_schedd*.

JOB_ROUTER_SCHEDD1_NAME The advertised daemon name of the *condor_schedd* serving as the source of jobs for routing. If not specified, this defaults to the local *condor_schedd*. If specified, this parameter must name the same *condor_schedd* whose spool is configured in `JOB_ROUTER_SCHEDD1_SPOOL`. If the named *condor_schedd* is not advertised in the local pool, `JOB_ROUTER_SCHEDD1_POOL` will also need to be set.

JOB_ROUTER_SCHEDD2_NAME The advertised daemon name of the *condor_schedd* to which the routed copy of the jobs are submitted. If not specified, this defaults to the local *condor_schedd*. If specified, this parameter must name the same *condor_schedd* whose spool is configured in `JOB_ROUTER_SCHEDD2_SPOOL`. If the named *condor_schedd* is not advertised in the local pool, `JOB_ROUTER_SCHEDD2_POOL` will also need to be set. Note that when *condor_job_router* is running as `root` and is submitting routed jobs to a different *condor_schedd* than the source *condor_schedd*, it is required that *condor_job_router* have permission to impersonate the job owners of the routed jobs. It is therefore usually necessary to configure `QUEUE_SUPER_USER_MAY_IMPERSONATE` in the configuration of the target *condor_schedd*.

JOB_ROUTER_SCHEDD1_POOL The Condor pool (*condor_collector* address) of the *condor_schedd* serving as the source of jobs for routing. If not specified, defaults to the local pool.

JOB_ROUTER_SCHEDD2_POOL The Condor pool (*condor_collector* address) of the *condor_schedd* to which the routed copy of the jobs are submitted. If not specified, defaults to the local pool.

3.5.20 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.GETADS_INTERVAL 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.UPDATE_INTERVAL 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.PRUNE_INTERVAL 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.DEBUG_ADS 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.MAX_LEASE_DURATION 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.MAX_TOTAL_LEASE_DURATION`.

LeaseManager.MAX_TOTAL_LEASE_DURATION An integer value representing seconds used to limit the *total* duration of leases, over all its renewals. The default value is 3600 seconds.

LeaseManager.DEFAULT_MAX_LEASE_DURATION 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.CLASSAD_LOG 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.QUERY_ADTYPE 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.QUERY_CONSTRAINTS 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.

3.5.21 Grid Monitor Configuration File Entries

These macros affect the Grid Monitor.

ENABLE_GRID_MONITOR A boolean value that when `True` enables the Grid Monitor. The Grid Monitor is used to reduce load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. The variable `GRID_MONITOR` must also be correctly configured. Defaults to `True`. See section 5.3.2 on page 568 for more information.

GRID_MONITOR The complete path name of the *grid_monitor.sh* tool used to reduce the 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` (the default value).

GRID_MONITOR_HEARTBEAT_TIMEOUT The integer number of seconds that may pass without hearing from a working Grid Monitor before 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 HTCondor notices a problem.

GRID_MONITOR_RETRY_DURATION When HTCondor-G attempts to start the Grid Monitor at a particular site, it will wait this many seconds to start hearing from the Grid Monitor. Defaults to 900 (15 minutes). If this duration passes without success, the Grid Monitor will be disabled for the site in question for the period of time set by `GRID_MONITOR_DISABLE_TIME`.

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 variable 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 900, which is 15 minutes.

GRID_MONITOR_DISABLE_TIME When an error occurs with a Grid Monitor job, this parameter controls how long the *condor_gridmanager* will wait before attempting to start a new Grid Monitor job. The value is in seconds and the default is 3600 (1 hour).

3.5.22 Configuration File Entries Relating to Grid Usage

These macros affect the HTCondor's usage of grid resources.

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.

GLEXEC_RETRIES An integer value that specifies the maximum number of times to retry a call to *glexec* when *glexec* exits with status 202 or 203, error codes that indicate a possible transient error condition. The default number of retries is 3.

GLEXEC_RETRY_DELAY An integer value that specifies the minimum number of seconds to wait between retries of a failed call to *glexec*. The default is 5 seconds. The actual delay to be used is determined by a random exponential backoff algorithm that chooses a delay with a minimum of the value of `GLEXEC_RETRY_DELAY` and a maximum of 100 times that value.

GLEXEC_HOLD_ON_INITIAL_FAILURE A boolean value that when `False` prevents a job from being put on hold when a failure is encountered during the glexec setup phase of managing a job. The default is `True`. *glexec* is invoked multiple times during each attempt to run a job. This configuration setting only disables putting the job on hold for the initial invocation. Subsequent failures during that run attempt always put the job on hold.

3.5.23 Configuration File Entries for DAGMan

These macros affect the operation of DAGMan and DAGMan jobs within HTCondor.

Note: Many, if not all, of these configuration variables will be most appropriately set on a per DAG basis, rather than in the global HTCondor configuration files. Per DAG configuration is explained in section 2.10.9. Also note that configuration settings of a running *condor_dagman* job are not changed by doing a *condor_reconfig*.

General

DAGMAN_CONFIG_FILE The path and name of the configuration file to be used by *condor_dagman*. This configuration variable is set automatically by *condor_submit_dag*, and it should not be explicitly set by the user. Defaults to the empty string.

DAGMAN_USE_STRICT An integer defining the level of strictness *condor_dagman* will apply when turning warnings into fatal errors, as follows:

- 0: no warnings become errors
- 1: severe warnings become errors
- 2: medium-severity warnings become errors
- 3: almost all warnings become errors

Using a strictness value greater than 0 may help find problems with a DAG that may otherwise escape notice. The default value if not defined is 1.

DAGMAN_STARTUP_CYCLE_DETECT A boolean value that defaults to `False`. 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. Note that setting this value to `True` will impose significant startup delays for large DAGs.

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 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. **Note: users should rarely change this setting.**

DAGMAN_USE_OLD_DAG_READER As of HTCondor version 8.3.3, this variable is no longer supported. Its value will always be False. A setting of True will result in a warning, and the setting will have no effect on how a DAG input file is read. The variable was previously used to change the reading of DAG input files to that of HTCondor versions prior to 8.0.6. **Note: users should never change this setting.**

DAGMAN_USE_SHARED_PORT A boolean value that controls whether *condor_dagman* will attempt to connect to the shared port daemon. If not defined, DAGMAN_USE_SHARED_PORT defaults to False. There is no reason to ever change this value; it was introduced to prevent spurious shared port-related error messages from appearing in dagman.out files. (Introduced in version 8.6.1.)

Throttling

DAGMAN_MAX_JOBS_IDLE An integer value that controls the maximum number of idle procs allowed within the DAG before *condor_dagman* temporarily stops submitting jobs. *condor_dagman* will resume submitting jobs once the number of idle procs falls below the specified limit. DAGMAN_MAX_JOBS_IDLE currently counts each individual proc within a cluster as a job, which is inconsistent with DAGMAN_MAX_JOBS_SUBMITTED. Note that submit description files that queue multiple procs can cause the DAGMAN_MAX_JOBS_IDLE limit to be exceeded. If a submit description file contains queue 5000 and DAGMAN_MAX_JOBS_IDLE is set to 250, this will result in 5000 procs being submitted to the *condor_schedd*, not 250; in this case, no further jobs will then be submitted by *condor_dagman* until the number of idle procs falls below 250. The default value is 1000. To disable this limit, set the value to 0. This configuration option can be overridden by the *condor_submit_dag* **-maxidle** *c*ommand-line argument (see 11).

DAGMAN_MAX_JOBS_SUBMITTED An integer value that controls the maximum number of node jobs (clusters) within the DAG that will be submitted to HTCondor at one time. A single invocation of *condor_submit* by *condor_dagman* counts as one job, even if the submit file produces a multi-proc cluster. The default value is 0 (unlimited). This configuration option can be overridden by the *condor_submit_dag* **-maxjobs** *c*ommand-line argument (see 11).

DAGMAN_MAX_PRE_SCRIPTS An integer defining the maximum number of PRE scripts that any given *condor_dagman* will run at the same time. The value 0 allows any number of PRE scripts to run. The default value if not defined is 20. Note that the DAGMAN_MAX_PRE_SCRIPTS value can be overridden by the *condor_submit_dag* **-maxpre** command line option.

DAGMAN_MAX_POST_SCRIPTS An integer defining the maximum number of POST scripts that any given *condor_dagman* will run at the same time. The value 0 allows any number of POST scripts to run. The default value if not defined is 20. Note that the DAGMAN_MAX_POST_SCRIPTS value can be overridden by the *condor_submit_dag* **-maxpost** command line option.

Priority, node semantics

DAGMAN_DEFAULT_PRIORITY An integer value defining the minimum priority of node jobs running under this *condor_dagman* job. Defaults to 0.

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 `DAGMAN_SUBMIT_DEPTH_FIRST` is set to `True`, consider also setting `DAGMAN_RETRY_SUBMIT_FIRST` and `DAGMAN_RETRY_NODE_FIRST` to `True`. If not defined, `DAGMAN_SUBMIT_DEPTH_FIRST` defaults to `False`.

DAGMAN_ALWAYS_RUN_POST A boolean value defining whether *condor_dagman* will ignore the return value of a PRE script when deciding whether to run a POST script. The default is `False`, which means that the failure of a PRE script causes the POST script to not be executed. Changing this to `True` will restore the previous behavior of *condor_dagman*, which is that a POST script is always executed, even if the PRE script fails. (The default for this value had originally been `False`, was changed to `True` in version 7.7.2, and then was changed back to `False` in version 8.5.4.)

Node job submission/removal

DAGMAN_USER_LOG_SCAN_INTERVAL An integer value representing the number of seconds that *condor_dagman* waits between checking the workflow log file for status updates. Setting this value lower than the default increases the CPU time *condor_dagman* spends checking files, perhaps fruitlessly, but increases responsiveness to nodes completing or failing. The legal range of values is 1 to INT_MAX. If not defined, it defaults to 5 seconds. (As of version 8.4.2, the default may be automatically decreased if `DAGMAN_MAX_JOBS_IDLE` is set to a small value. If so, this will be noted in the `dagman.out` file.)

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. (As of version 8.4.2, the default may be automatically decreased if `DAGMAN_MAX_JOBS_IDLE` is set to a small value. If so, this will be noted in the `dagman.out` file.)

Note: The maximum rate at which DAGMan can submit jobs is DAGMAN_MAX_SUBMITS_PER_INTERVAL / DAGMAN_USER_LOG_SCAN_INTERVAL.

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_MAX_JOB_HOLDS An integer value defining the maximum number of times a node job is allowed to go on hold. As a job goes on hold this number of times, it is removed from the queue. For example, if the value is 2, as the job goes on hold for the second time, it will be removed. At this time, this feature is not fully compatible with node jobs that have more than one `ProcID`. The number of holds of each process in the cluster count towards the total, rather than counting individually. So, this setting should take that possibility into account, possibly using a larger value. A value of 0 allows a job to go on hold any number of times. The default value if not defined is 100.

DAGMAN_HOLD_CLAIM_TIME An integer defining the number of seconds that *condor_dagman* will cause a hold on a claim after a job is finished, using the job ClassAd attribute `KeepClaimIdle`. The default value is 20. A value of 0 causes *condor_dagman* not to set the job ClassAd attribute.

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 any non negative integer. If defined with a value less than 0, the value 0 will be used.

DAGMAN_PROHIBIT_MULTI_JOBS A boolean value that controls whether *condor_dagman* prohibits node job submit description 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_MULTI_JOBS` defaults to `False`.

DAGMAN_GENERATE_SUBDAG_SUBMITS A boolean value specifying whether *condor_dagman* itself should create the `.condor.sub` files for nested DAGs. If set to `False`, nested DAGs will fail unless the `.condor.sub` files are generated manually by running *condor_submit_dag -no_submit* on each nested DAG, or the *-do_recurse* flag is passed to *condor_submit_dag* for the top-level DAG. DAG nodes specified with the `SUBDAG EXTERNAL` keyword or with submit description file names ending in `.condor.sub` are considered nested DAGs. The default value if not defined is `True`.

DAGMAN_REMOVE_NODE_JOBS A boolean value that controls whether *condor_dagman* removes its node jobs itself when it is removed (in addition to the *condor_schedd* removing them). Note that setting `DAGMAN_REMOVE_NODE_JOBS` to `True` is the safer option (setting it to `False` means that there is some chance of endig up with "orphan" node jobs). Setting `DAGMAN_REMOVE_NODE_JOBS` to `False` is a performance optimization (decreasing the load on the *condor_schedd* when a *condor_dagman* job is removed). Note that even if `DAGMAN_REMOVE_NODE_JOBS` is set to `False`, *condor_dagman* will remove its node jobs in some cases, such as a DAG abort triggered by an *ABORT-DAG-ON* command. Defaults to `True`.

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 followed by the period character (`.`) 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 named B, that node will internally be renamed to 2.B. If not defined, `DAGMAN_MUNGE_NODE_NAMES` defaults to `True`. **Note: users should rarely change this setting.**

DAGMAN_SUPPRESS_JOB_LOGS A boolean value specifying whether events should be written to a log file specified in a node job's submit description file. The default value is `False`, such that events are written to a log file specified by a node job.

DAGMAN_SUPPRESS_NOTIFICATION A boolean value defining whether jobs submitted by *condor_dagman* will use email notification when certain events occur. If `True`, all jobs submitted by *condor_dagman* will have the

equivalent of the submit command `notification = never` set. This does not affect the notification for events relating to the *condor_dagman* job itself. Defaults to `True`.

DAGMAN_CONDOR_SUBMIT_EXE The executable that *condor_dagman* will use to submit HTCondor jobs. If not defined, *condor_dagman* looks for *condor_submit* in the path. **Note: users should rarely change this setting.**

DAGMAN_CONDOR_RM_EXE The executable that *condor_dagman* will use to remove HTCondor jobs. If not defined, *condor_dagman* looks for *condor_rm* in the path. **Note: users should rarely change this setting.**

DAGMAN_ABORT_ON_SCARY_SUBMIT A boolean value that controls whether to abort a DAG upon detection of a scary submit event. An example of a scary submit event is one in which the HTCondor ID does not match the expected value. Note that in all HTCondor versions prior to 6.9.3, *condor_dagman* did *not* abort 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`. **Note: users should rarely change this setting.**

DAGMAN_STORK_SUBMIT_EXE This configuration variable is no longer used; as of HTCondor version 8.3.4, *condor_dagman* no longer supports Stork jobs. Setting this configuration variable will result in a warning from *condor_dagman* (which will be turned into a fatal error if `DAGMAN_USE_STRICT` is set to 1 or above). **Note: users should never change this setting.**

For completeness, here is the definition for historical purposes: The executable that *condor_dagman* will use to submit Stork jobs. If not defined, *condor_dagman* looks for *stork_submit* in the path.

DAGMAN_STORK_RM_EXE This configuration variable is no longer used; as of HTCondor version 8.3.4, *condor_dagman* no longer supports Stork jobs. Setting this configuration variable will result in a warning from *condor_dagman* (which will be turned into a fatal error if `DAGMAN_USE_STRICT` is set to 1 or above). **Note: users should never change this setting.**

For completeness, here is the definition for historical purposes: The executable that *condor_dagman* will use to remove Stork jobs. If not defined, *condor_dagman* looks for *stork_rm* in the path.

Rescue/retry

DAGMAN_AUTO_RESCUE A boolean value that controls whether *condor_dagman* automatically runs Rescue DAGs. If `DAGMAN_AUTO_RESCUE` is `True` and the DAG input file `my.dag` is submitted, and if a Rescue DAG such as the examples `my.dag.rescue001` or `my.dag.rescue002` exists, then the largest magnitude Rescue DAG will be run. If not defined, `DAGMAN_AUTO_RESCUE` defaults to `True`.

DAGMAN_MAX_RESCUE_NUM An integer value that controls the maximum Rescue DAG number that will be written, in the case that `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 prevents a Rescue DAG from being written at all, or automatically run. If not defined, `DAGMAN_MAX_RESCUE_NUM` defaults to 100.

DAGMAN_RESET_RETRIES_UPON_RESCUE A boolean value that controls whether node retries are reset in a Rescue DAG. If this value is `False`, the number of node retries written in a Rescue DAG is decreased, if any retries were used in the original run of the DAG; otherwise, the original number of retries is allowed when running the Rescue DAG. If not defined, `DAGMAN_RESET_RETRIES_UPON_RESCUE` defaults to `True`.

DAGMAN_WRITE_PARTIAL_RESCUE A boolean value that controls whether *condor_dagman* writes a partial or a full DAG file as a Rescue DAG. As of HTCondor version 7.2.2, writing a partial DAG is preferred. If not defined, DAGMAN_WRITE_PARTIAL_RESCUE defaults to True. **Note: users should rarely change this setting.**

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 had been the behavior of *condor_dagman*. 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 had been the behavior of *condor_dagman*. If not defined, it defaults to False.

DAGMAN_OLD_RESCUE This configuration variable is no longer used. **Note: users should never change this setting.**

Log files

DAGMAN_DEFAULT_NODE_LOG The default name of a file to be used as a job event log by all node jobs of a DAG.

This configuration variable uses a special syntax in which **@** instead of **$** indicates an evaluation of special variables. Normal HTCondor configuration macros may be used with the normal **$** syntax.

Special variables to be used only in defining this configuration variable:

- @(DAG_DIR): The directory in which the primary DAG input file resides. If more than one DAG input file is specified to *condor_submit_dag*, the primary DAG input file is the leftmost one on the command line.
- @(DAG_FILE): The name of the primary DAG input file. It does not include the path.
- @(CLUSTER): The ClusterId attribute of the *condor_dagman* job.
- @(OWNER): The user name of the user who submitted the DAG.
- @(NODE_NAME): For SUBDAGs, this is the node name of the SUBDAG in the upper level DAG; for a top-level DAG, it is the string "undef".

If not defined, @(DAG_DIR)/@(DAG_FILE).nodes.log is the default value.

Notes:

- Using $(LOG) in defining a value for DAGMAN_DEFAULT_NODE_LOG will not have the expected effect, because $(LOG) is defined as "." for *condor_dagman*. To place the default log file into the log directory, write the expression relative to a known directory, such as $(LOCAL_DIR)/log (see examples below).
- A default log file placed in the spool directory will need extra configuration to prevent *condor_preen* from removing it; modify VALID_SPOOL_FILES. Removal of the default log file during a run will cause severe problems.

- **The value defined for DAGMAN_DEFAULT_NODE_LOG must ensure that the file is unique for each DAG.** Therefore, the value should always include @(DAG_FILE). For example,

  ```
  DAGMAN_DEFAULT_NODE_LOG = $(LOCAL_DIR)/log/@(DAG_FILE).nodes.log
  ```

  is okay, but

  ```
  DAGMAN_DEFAULT_NODE_LOG = $(LOCAL_DIR)/log/dag.nodes.log
  ```

  will cause failure when more than one DAG is run at the same time on a given submit machine.

DAGMAN_LOG_ON_NFS_IS_ERROR A boolean value that controls whether *condor_dagman* prohibits a DAG workflow log from being on an NFS file system. This value is ignored if CREATE_LOCKS_ON_LOCAL_DISK and ENABLE_USERLOG_LOCKING are both True. If a DAG uses such a workflow log file file and DAGMAN_LOG_ON_NFS_IS_ERROR is True (and not ignored), the DAG will abort during the initialization process. If not defined, DAGMAN_LOG_ON_NFS_IS_ERROR defaults to False.

DAGMAN_ALLOW_LOG_ERROR This configuration variable is no longer used; as of HTCondor version 8.3.4, *condor_dagman* no longer supports Stork jobs. **Note: users should never change this setting.**

For completeness, here is the definition for historical purposes: A boolean value defining whether *condor_dagman* will still attempt to run a node job, even if errors are detected in the job event log specification. This setting has an effect only on nodes that are Stork jobs (not HTCondor jobs). The default value if not defined is False.

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. **Note: users should rarely change this setting.**

The DAGMAN_ALLOW_EVENTS value is a logical bitwise OR of the following values:

0 = allow no bad events

1 = allow all bad events, *except* the event "job re-run after terminated event"

2 = allow terminated/aborted event combination

4 = allow a "job re-run after terminated event" bug

8 = allow garbage or orphan events

16 = allow an execute or terminate event before job's submit event

32 = allow two terminated events per job, as sometimes seen with grid jobs

64 = allow duplicated events in general

The default value is 114, which allows terminated/aborted event combination, allows an execute and/or terminated event before job's submit event, allows double terminated events, and allows general duplicate events.

As examples, 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 will never abort the DAG because of a bad event. But this value should almost never be used, because the "job re-run after terminated event" bug breaks the semantics of the DAG.

DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION This configuration variable is no longer used. The improved functionality of the DAGMAN_ALLOW_EVENTS macro eliminates the need for this variable. **Note: users should never change this setting.**

For completeness, here is the definition for historical purposes: A boolean value that controls whether *condor_dagman* aborts or continues with a DAG in the rare case that HTCondor erroneously executes the job within a DAG node more than once. A bug in HTCondor 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 HTCondor 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_ALWAYS_USE_NODE_LOG As of HTCondor version 8.3.1, the value must always be the default value of True. Attempting to set it to False results in an error. This causes incompatibility with using a *condor_submit* executable that is older than HTCondor version 7.9.0. **Note: users should never change this setting.**

For completeness, here is the definition for historical purposes: A boolean value that when True causes *condor_dagman* to read events from its default node log file, as defined by DAGMAN_DEFAULT_NODE_LOG, instead of from the log file(s) defined in the node job submit description files. When True, *condor_dagman* will read events only from the default log file, and POST script terminated events will be written only to the default log file, and not to the log file(s) defined in the node job submit description files. The default value is True.

Debug output

DAGMAN_DEBUG This variable is described in section 3.5.2 as <SUBSYS>_DEBUG.

DAGMAN_VERBOSITY An integer value defining the verbosity of output to the dagman.out file, as follows (each level includes all output from lower debug levels):

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; output errors and warnings
- level = 3; normal output
- 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

The default value if not defined is 3.

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 representing the number of bytes of log lines 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_PENDING_REPORT_INTERVAL An integer value representing the number of seconds that controls how often *condor_dagman* will print a report of pending nodes to the `dagman.out` file. The report will only be printed if *condor_dagman* has been waiting at least `DAGMAN_PENDING_REPORT_INTERVAL` seconds without seeing any node job events, in order to avoid cluttering the `dagman.out` file. This feature is mainly intended to help diagnose *condor_dagman* processes that are stuck waiting indefinitely for a job to finish. If not defined, `DAGMAN_PENDING_REPORT_INTERVAL` defaults to 600 seconds (10 minutes).

MAX_DAGMAN_LOG This variable is described in section 3.5.2 as `MAX_<SUBSYS>_LOG`. If not defined, `MAX_DAGMAN_LOG` defaults to 0 (unlimited size).

HTCondor attributes

DAGMAN_COPY_TO_SPOOL A boolean value that when `True` copies the *condor_dagman* binary to the spool directory when a DAG is submitted. Setting this variable to `True` allows long-running DAGs to survive a DAGMan version upgrade. For running large numbers of small DAGs, leave this variable unset or set it to `False`. The default value if not defined is `False`. **Note: users should rarely change this setting.**

DAGMAN_INSERT_SUB_FILE A file name of a file containing submit description 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 *condor_submit_dag* **-insert_sub_file** command line option.

DAGMAN_ON_EXIT_REMOVE Defines the `OnExitRemove` ClassAd expression placed into the *condor_dagman* submit description file by *condor_submit_dag*. The default expression is designed to ensure that *condor_dagman* is automatically re-queued by the *condor_schedd* daemon if it exits abnormally or is killed (for example, during a reboot). If this results in *condor_dagman* staying in the queue when it should exit, consider changing to a less restrictive expression, as in the example

```
(ExitBySignal == false || ExitSignal =!= 9)
```

If not defined, `DAGMAN_ON_EXIT_REMOVE` defaults to the expression

```
( ExitSignal =?= 11 || (ExitCode =!= UNDEFINED && ExitCode >=0 && ExitCode <= 2))
```

Metrics

DAGMAN_PEGASUS_REPORT_METRICS The path to the *condor_dagman_metrics_reporter* executable, which is optionally used to anonymously report workflow metrics for Pegasus workflows. Defaults to `$(LIBEXEC)/condor_dagman_metrics_reporter`. **Note: users should rarely change this setting.**

DAGMAN_PEGASUS_REPORT_TIMEOUT An integer value specifying the maximum number of seconds that the *condor_dagman_metrics_reporter* will spend attempting to report metrics to the Pegasus metrics server. Defaults to 100.

3.5.24 Configuration File Entries Relating to Security

These macros affect the secure operation of HTCondor. Many of these macros are described in section 3.8 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_*_AUTHENTICATION_METHODS This section has not yet been written

SEC_*_CRYPTO_METHODS This section has not yet been written

GSI_DAEMON_NAME 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) used by Condor daemons to which this configuration of Condor will connect. The * character may be used as a wild card character. When `GSI_DAEMON_NAME` is defined, only certificates matching `GSI_DAEMON_NAME` pass the authentication step, and no check is performed to require that the host name of the daemon matches the host name in the daemon's certificate. When `GSI_DAEMON_NAME` is not defined, the host name of the daemon and certificate must match unless exempted by the use of `GSI_SKIP_HOST_CHECK` and/or `GSI_SKIP_HOST_CHECK_CERT_REGEX`.

GSI_SKIP_HOST_CHECK A boolean variable that controls whether a check is performed during GSI authentication of a Condor daemon. When the default value of `False`, the check is not skipped, so the daemon host name must match the host name in the daemon's certificate, unless otherwise exempted by the use of `GSI_DAEMON_NAME` or `GSI_SKIP_HOST_CHECK_CERT_REGEX`. When `True`, this check is skipped, and hosts will not be rejected due to a mismatch of certificate and host name.

GSI_SKIP_HOST_CHECK_CERT_REGEX This may be set to a regular expression. GSI certificates of Condor daemons with a subject name that are matched in full by this regular expression are not required to have a matching daemon host name and certificate host name. The default is an empty regular expression, which will not match any certificates, even if they have an empty subject name.

HOST_ALIAS Specifies the fully qualified host name that clients authenticating this daemon with GSI should expect the daemon's certificate to match. The alias is advertised to the *condor_collector* as part of the address of the daemon. When this is not set, clients validate the daemon's certificate host name by matching it against DNS A records for the host they are connected to. See GSI_SKIP_HOST_CHECK for ways to disable this validation step.

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. The value is unset by default.

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 HTCondor 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 HTCondor 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 HTCondor uses GSI_DAEMON_DIRECTORY to construct the directory path as

```
GSI_DAEMON_TRUSTED_CA_DIR  = $(GSI_DAEMON_DIRECTORY)/certificates
```

The EC2 GAHP may use this directory in the specification a trusted CA.

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.

GSI_AUTHZ_CONF A complete path and file name of the Globus mapping library that looks for the mapping call out configuration. There is no default value; as such, HTCondor uses the environment variable GSI_AUTHZ_CONF when this variable is not defined. Setting this variable to /dev/null disables callouts.

GSS_ASSIST_GRIDMAP_CACHE_EXPIRATION The length of time, in seconds, to cache the result of the Globus mapping lookup result when using Globus to map certificates to HTCondor user names. The lookup only occurs when the canonical name GSS_ASSIST_GRIDMAP is present in the HTCondor map file. The default value is 0 seconds, which is a special value that disables caching. The cache uses the DN and VOMS FQAN as a key; very rare Globus configurations that utilize other certificate attributes for the mapping may cause the cache to return a different user than Globus.

DELEGATE_JOB_GSI_CREDENTIALS A boolean value that defaults to `True` for HTCondor 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.

DELEGATE_FULL_JOB_GSI_CREDENTIALS A boolean value that controls whether HTCondor will delegate a full or limited GSI X.509 proxy. The default value of `False` indicates the limited GSI X.509 proxy.

DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME An integer value that specifies the maximum number of seconds for which delegated proxies should be valid. The default value is one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. The job may override this configuration setting by using the **delegate_job_GSI_credentials_lifetime** submit file command. This configuration variable currently only applies to proxies delegated for non-grid jobs and HTCondor-C jobs. It does not currently apply to globus grid jobs, which always behave as though the value is 0. This variable has no effect if `DELEGATE_JOB_GSI_CREDENTIALS` is `False`.

DELEGATE_JOB_GSI_CREDENTIALS_REFRESH A floating point number between 0 and 1 that indicates the fraction of a proxy's lifetime at which point delegated credentials with a limited lifetime should be renewed. The renewal is attempted periodically at or near the specified fraction of the lifetime of the delegated credential. The default value is 0.25. This setting has no effect if `DELEGATE_JOB_GSI_CREDENTIALS` is `False` or if `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME` is 0. For non-grid jobs, the precise timing of the proxy refresh depends on `SHADOW_CHECKPROXY_INTERVAL`. To ensure that the delegated proxy remains valid, the interval for checking the proxy should be, at most, half of the interval for refreshing it.

GSI_DELEGATION_KEYBITS The integer number of bits in the GSI key. If set to 0, the number of bits will be that preferred by the GSI library. If set to less than 1024, the value will be ignored, and the key size will be the default size of 1024 bits. Setting the value greater than 4096 is likely to cause long compute times.

GSI_DELEGATION_CLOCK_SKEW_ALLOWABLE The number of seconds of clock skew permitted for delegated proxies. The default value is 300 (5 minutes). This default value is also used if this variable is set to 0.

GRIDMAP The complete path and file name of the Globus Gridmap file. The Gridmap file is used to map X.509 distinguished names to HTCondor user ids.

SEC_<access-level>_SESSION_DURATION The amount of time in seconds before a communication session expires. 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. A short session duration increases the frequency with which daemons have to reauthenticate with each other, which may impact performance.

If the client and server are configured with different durations, the shorter of the two will be used. The default for daemons is 86400 seconds (1 day) and the default for command-line tools is 60 seconds. The shorter default for command-line tools is intended to prevent daemons from accumulating a large number of communication sessions from the short-lived tools that contact them over time. A large number of security sessions consumes a large amount of memory. It is therefore important when changing this configuration setting to preserve the small session duration for command-line tools.

One example of how to safely change the session duration is to explicitly set a short duration for tools and *condor_submit* and a longer duration for everything else:

```
SEC_DEFAULT_SESSION_DURATION = 50000
TOOL.SEC_DEFAULT_SESSION_DURATION = 60
SUBMIT.SEC_DEFAULT_SESSION_DURATION = 60
```

Another example of how to safely change the session duration is to explicitly set the session duration for a specific daemon:

```
COLLECTOR.SEC_DEFAULT_SESSION_DURATION = 50000
```

SEC_<access-level>_SESSION_LEASE The maximum number of seconds an unused security session will be kept in a daemon's session cache before being removed to save memory. The default is 3600. If the server and client have different configurations, the smaller one will be used.

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_REMOTE_DIR 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_EXECUTE_DIRECTORY A boolean value that, when `True`, causes the execute directory for jobs on Linux or Windows platforms to be encrypted. Defaults to `False`. Note that even if `False`, the user can require encryption of the execute directory on a per-job basis by setting **encrypt_execute_directory** to `True` in the job submit description file. Enabling this functionality requires that the HTCondor service is run as user root on Linux platforms, or as a system service on Windows platforms. On Linux platforms, the encryption method is *ecryptfs*, and therefore requires an installation of the `ecryptfs-utils` package. On Windows platforms, the encryption method is the EFS (Encrypted File System) feature of NTFS.

ENCRYPT_EXECUTE_DIRECTORY_FILENAMES A boolean value relevant on Linux platforms only. Defaults to `False`. On Windows platforms, file names are not encrypted, so this variable has no effect. When using an encrypted execute directory, the contents of the files will always be encrypted. On Linux platforms, file names may or may not be encrypted. There is some overhead and there are restrictions on encrypting file names (see the *ecryptfs* documentation). As a result, the default does not encrypt file names on Linux platforms, and the administrator may choose to enable encryption behavior by setting this configuration variable to `True`.

ECRYPTFS_ADD_PASSPHRASE The path to the *ecryptfs-add-passphrase* command-line utility. If the path is not fully-qualified, then safe system path subdirectories such as `/bin` and `/usr/bin` will be searched. The default value is `ecryptfs-add-passphrase`, causing the search to be within the safe system path subdirectories. This configuration variable is used on Linux platforms when a job sets **encrypt_execute_directory** to `True` in the submit description file.

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 HTCondor should attempt authenticating network connections before giving up. The default imposes no time limit, so the attempt never gives up. 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_PASSWORD_FILE 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.

CERTIFICATE_MAPFILE A path and file name of the unified map file.

CERTIFICATE_MAPFILE_ASSUME_HASH_KEYS For HTCondor version 8.5.8 and later. When this is true, the second field of the `CERTIFICATE_MAPFILE` is not interpreted as a regular expression unless it begins and ends with the slash / character.

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 (for example, on a pool with one *condor_schedd* daemon and thousands of *condor_startd* daemons on a network with a 0.1 second round trip time).

The default value is `True`. To have any effect, it must be `True` in the configuration of both the execute side (*condor_startd*) as well as the submit side (*condor_schedd*). When `True`, 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 *condor_startd* daemon's settings for `SEC_DAEMON_ENCRYPTION` and `SEC_DAEMON_INTEGRITY`; the configuration of these values in the *condor_schedd*, *condor_shadow*, and *condor_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 *condor_schedd* and *condor_shadow* will be authenticated as `submit-side@matchsession` when they talk to the *condor_startd* and *condor_starter*. The *condor_startd* and *condor_starter* will be authenticated as `execute-side@matchsession` when they talk to the *condor_schedd* and *condor_shadow*. These identities is automatically added to the DAEMON, READ, and CLIENT authorization levels in these daemons when needed.

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.

3.5.25 Configuration File Entries Relating to Virtual Machines

These macros affect how HTCondor 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*. The default value is `$(SBIN)/condor_vm-gahp`.

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 `kvm`, `xen` or `vmware`. There is no default value for this required configuration variable.

VM_MEMORY An integer specifying the maximum amount of memory in MiB to be shared among the VM universe jobs run on this machine.

VM_MAX_NUMBER An integer limit on the number of executing virtual machines. When not defined, the default value is the same `NUM_CPUS`. When it evaluates to `Undefined`, as is the case when not defined with a numeric value, no meaningful limit is imposed.

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 HTCondor 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, HTCondor 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 HTCondor 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.

VM_NETWORKING_BRIDGE_INTERFACE For Xen and KVM only, a required string if bridge networking is to be enabled. It specifies the networking interface that vm universe jobs will use.

LIBVIRT_XML_SCRIPT For Xen and KVM only, a path and executable specifying a program. When the *condor_vm-gahp* is ready to start a Xen or KVM **vm** universe job, it will invoke this program to generate the XML description of the virtual machine, which it then provides to the virtualization software. The job ClassAd will be provided to this program via standard input. This program should print the XML to standard output. If this configuration variable is not set, the *condor_vm-gahp* will generate the XML itself. The provided script in $(LIBEXEC)/libvirt_simple_script.awk will generate the same XML that the *condor_vm-gahp* would.

LIBVIRT_XML_SCRIPT_ARGS For Xen and KVM only, the command-line arguments to be given to the program specified by LIBVIRT_XML_SCRIPT.

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 HTCondor starts the virtual machine. This parameter is optional.

The following configuration variables are specific to the Xen virtual machine software.

XEN_BOOTLOADER A required full path and executable for the Xen bootloader, if the kernel image includes a disk image.

The following two macros affect the configuration of HTCondor where HTCondor is running on a host machine, the host machine is running an inner virtual machine, and HTCondor 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.

3.5.26 Configuration File Entries Relating to High Availability

These macros affect the high availability operation of HTCondor.

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 3.3.12). 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 3.3.12.

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_CONTROLLEE This variable is used to specify the name of the daemon which the *condor_had* daemon controls. This name should match the daemon name in the *condor_master* daemon's DAEMON_LIST definition. The default value is NEGOTIATOR.

HAD_CONNECTION_TIMEOUT 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 HTCondor) 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 MiB.

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. The default value is $(LOG)/HADLog.

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).

MAX_TRANSFER_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.

REPLICATION_ARGS 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_REPLICATION_LOG 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 MiB.

REPLICATION_DEBUG Logging level for the *condor_replication* daemon. See `<SUBSYS>_DEBUG` for values.

REPLICATION_LOG Full path and file name to the log file. The default value is `$(LOG)/ReplicationLog`.

TRANSFERER The full path and file name of the *condor_transferer* executable. The default value is `$(LIBEXEC)/condor_transferer`.

TRANSFERER_LOG Full path and file name to the log file. The default value is `$(LOG)/TransfererLog`.

TRANSFERER_DEBUG Logging level for the *condor_transferer* daemon. See `<SUBSYS>_DEBUG` for values.

MAX_TRANSFERER_LOG 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 MiB.

3.5.27 MyProxy Configuration File Macros

In some cases, HTCondor 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, HTCondor-G and *condor_credd* will have the capability to autonomously refresh proxy certificates. By default, this macro is undefined.

3.5.28 Configuration File Macros Affecting APIs

ENABLE_SOAP A boolean value that defaults to False. When True, HTCondor 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, HTCondor 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 expression 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. If provided, this expression will be logically ANDed with the default behavior of leaving the job in the queue until FilesRetrieved becomes True.

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 The 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. This parameter is required when SOAP over SSL is enabled. There is no default value.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.12.

SOAP_SSL_SERVER_KEYFILE The 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. This parameter is required when SOAP over SSL is enabled. 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 The 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. When SOAP over SSL is enabled, this parameter or SOAP_SSL_CA_DIR must be set. There is no default value. The EC2 GAHP may use this file to specify a trusted CA.

SOAP_SSL_CA_DIR The 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. When SOAP over SSL is enabled, this variable or the variable SOAP_SSL_CA_FILE must be defined. There is no default value. The EC2 GAHP may use this directory in the specification a trusted CA.

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.

SOAP_SSL_SKIP_HOST_CHECK When a SOAP server is authenticated via SSL, the server's host name is normally compared with the host name contained in the server's X.509 credential. If the two do not match, authentication fails. When this boolean variable is set to `True`, the host name comparison is disabled. The default value is `False`.

3.5.29 Configuration File Entries Relating to condor_ssh_to_job

These macros affect how HTCondor deals with *condor_ssh_to_job*, a tool that allows users to interactively debug jobs. With these configuration variables, the administrator can control who can use the tool, and how the *ssh* programs are invoked. The manual page for *condor_ssh_to_job* is at section 11.

ENABLE_SSH_TO_JOB A boolean expression read by the *condor_starter*, that when `True` allows the owner of the job or a queue super user on the *condor_schedd* where the job was submitted to connect to the job via *ssh*. The expression may refer to attributes of both the job and the machine ClassAds. The job ClassAd attributes may be referenced by using the prefix `TARGET.`, and the machine ClassAd attributes may be referenced by using the prefix `MY.`. When `False`, it prevents *condor_ssh_to_job* from starting an *ssh* session. The default value is `True`.

SCHEDD_ENABLE_SSH_TO_JOB A boolean expression read by the *condor_schedd*, that when `True` allows the owner of the job or a queue super user to connect to the job via *ssh* if the execute machine also allows *condor_ssh_to_job* access (see `ENABLE_SSH_TO_JOB`). The expression may refer to attributes of only the job ClassAd. When `False`, it prevents *condor_ssh_to_job* from starting an *ssh* session for all jobs managed by the *condor_schedd*. The default value is `True`.

SSH_TO_JOB_<SSH-CLIENT>_CMD A string read by the *condor_ssh_to_job* tool. It specifies the command and arguments to use when invoking the program specified by `<SSH-CLIENT>`. Values substituted for the placeholder `<SSH-CLIENT>` may be `SSH`, `SFTP`, `SCP`, or any other *ssh* client capable of using a command as a proxy for the connection to *sshd*. The entire command plus arguments string is enclosed in double quote marks. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in a *condor_submit* file. The following substitutions are made within the arguments:

`%h`: is substituted by the remote host

`%i`: is substituted by the ssh key

`%k`: is substituted by the known hosts file

`%u`: is substituted by the remote user

`%x`: is substituted by a proxy command suitable for use with the *OpenSSH* ProxyCommand option

`%%`: is substituted by the percent mark character

The default string is:
`"ssh -oUser=%u -oIdentityFile=%i -oStrictHostKeyChecking=yes -oUserKnownHostsFile=%`

When the `<SSH-CLIENT>` is *scp*, %h is omitted.

SSH_TO_JOB_SSHD The path and executable name of the *ssh* daemon. The value is read by the *condor_starter*. The default value is `/usr/sbin/sshd`.

SSH_TO_JOB_SSHD_ARGS A string, read by the *condor_starter* that specifies the command-line arguments to be passed to the *sshd* to handle an incoming ssh connection on its `stdin` or `stdout` streams in inetd mode. Enclose the entire arguments string in double quote marks. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in an HTCondor submit description file. Within the arguments, the characters `%f` are replaced by the path to the *sshd* configuration file the characters `%%` are replaced by a single percent character. The default value is the string `"-i -e -f %f"`.

SSH_TO_JOB_SSHD_CONFIG_TEMPLATE A string, read by the *condor_starter* that specifies the path and file name of an *sshd* configuration template file. The template is turned into an *sshd* configuration file by replacing macros within the template that specify such things as the paths to key files. The macro replacement is done by the script `$(LIBEXEC)/condor_ssh_to_job_sshd_setup`. The default value is `$(LIB)/condor_ssh_to_job_sshd_config_template`.

SSH_TO_JOB_SSH_KEYGEN A string, read by the *condor_starter* that specifies the path to *ssh_keygen*, the program used to create ssh keys.

SSH_TO_JOB_SSH_KEYGEN_ARGS A string, read by the *condor_starter* that specifies the command-line arguments to be passed to the *ssh_keygen* to generate an ssh key. Enclose the entire arguments string in double quotes. Individual arguments may be quoted with single quotes, using the same syntax as for arguments in an HTCondor submit description file. Within the arguments, the characters `%f` are replaced by the path to the key file to be generated, and the characters `%%` are replaced by a single percent character. The default value is the string `"-N '' -C '' -q -f %f -t rsa"`. If the user specifies additional arguments with the command `condor_ssh_to_job -keygen-options`, then those arguments are placed after the arguments specified by the value of `SSH_TO_JOB_SSH_KEYGEN_ARGS`.

3.5.30 condor_rooster Configuration File Macros

condor_rooster is an optional daemon that may be added to the *condor_master* daemon's `DAEMON_LIST`. It is responsible for waking up hibernating machines when their `UNHIBERNATE` expression becomes `True`. In the typical case, a pool runs a single instance of *condor_rooster* on the central manager. However, if the network topology requires that Wake On LAN packets be sent to specific machines from different locations, *condor_rooster* can be run on any machine(s) that can read from the pool's *condor_collector* daemon.

For *condor_rooster* to wake up hibernating machines, the collecting of offline machine ClassAds must be enabled. See variable `COLLECTOR_PERSISTENT_AD_LOG` on page 290 for details on how to do this.

ROOSTER_INTERVAL The integer number of seconds between checks for offline machines that should be woken. The default value is 300.

ROOSTER_MAX_UNHIBERNATE An integer specifying the maximum number of machines to wake up per cycle. The default value of 0 means no limit.

ROOSTER_UNHIBERNATE A boolean expression that specifies which machines should be woken up. The default expression is `Offline && Unhibernate`. If network topology or other considerations demand that some

machines in a pool be woken up by one instance of *condor_rooster*, while others be woken up by a different instance, ROOSTER_UNHIBERNATE may be set locally such that it is different for the two instances of *condor_rooster*. In this way, the different instances will only try to wake up their respective subset of the pool.

ROOSTER_UNHIBERNATE_RANK A ClassAd expression specifying which machines should be woken up first in a given cycle. Higher ranked machines are woken first. If the number of machines to be woken up is limited by ROOSTER_MAX_UNHIBERNATE, the rank may be used for determining which machines are woken before reaching the limit.

ROOSTER_WAKEUP_CMD A string representing the command line invoked by *condor_rooster* that is to wake up a machine. The command and any arguments should be enclosed in double quote marks, the same as **arguments** syntax in an HTCondor submit description file. The default value is "$(BIN)/condor_power -d -i". The command is expected to read from its standard input a ClassAd representing the offline machine.

3.5.31 condor_shared_port Configuration File Macros

These configuration variables affect the *condor_shared_port* daemon. For general discussion of the *condor_shared_port* daemon, see 435.

USE_SHARED_PORT A boolean value that specifies whether HTCondor daemons should rely on the *condor_shared_port* daemon for receiving incoming connections. Under Unix, write access to the location defined by DAEMON_SOCKET_DIR is required for this to take effect. The default is True.

SHARED_PORT_PORT The default TCP port used by the *condor_shared_port* daemon. If COLLECTOR_USES_SHARED_PORT is the default value of True, and the *condor_master* launches a *condor_collector* daemon, then the *condor_shared_port* daemon will ignore this value and use the TCP port assigned to the *condor_collector* via the COLLECTOR_HOST configuration variable.

The default value is $(COLLECTOR_PORT), which defaults to 9618. Note that this causes all HTCondor hosts to use TCP port 9618 by default, differing from previous behavior. The previous behavior has only the *condor_collector* host using a fixed port. To restore this previous behavior, set SHARED_PORT_PORT to 0, which will cause the *condor_shared_port* daemon to use a randomly selected port in the range LOWPORT - HIGHPORT, as defined in section 3.9.1.

SHARED_PORT_DAEMON_AD_FILE This specifies the full path and name of a file used to publish the address of *condor_shared_port*. This file is read by the other daemons that have USE_SHARED_PORT=True and which are therefore sharing the same port. The default typically does not need to be changed.

SHARED_PORT_MAX_WORKERS An integer that specifies the maximum number of sub-processes created by *condor_shared_port* while servicing requests to connect to the daemons that are sharing the port. The default is 50.

DAEMON_SOCKET_DIR This specifies the directory where Unix versions of HTCondor daemons will create named sockets so that incoming connections can be forwarded to them by *condor_shared_port*. If this directory does not exist, it will be created. The maximum length of named socket paths plus names is restricted by the operating system, so using a path that is longer than 90 characters may cause failures.

Write access to this directory grants permission to receive connections through the shared port. By default, the directory is created to be owned by HTCondor and is made to be only writable by HTCondor. One possible reason to broaden access to this directory is if execute nodes are accessed via CCB and the submit node is behind a firewall with only one open port, which is the port assigned to *condor_shared_port*. In this case, commands that interact with the execute node, such as *condor_ssh_to_job*, will not be able to operate unless run by a user with write access to DAEMON_SOCKET_DIR. In this case, one could grant tmp-like permissions to this directory so that all users can receive CCB connections back through the firewall. But, consider the wisdom of having a firewall in the first place, if it will be circumvented in this way.

On Linux platforms, daemons use abstract named sockets instead of normal named sockets. Abstract sockets are not not tied to a file in the file system. The *condor_master* picks a random prefix for abstract socket names and shares it privately with the other daemons. When searching for the recipient of an incoming connection, *condor_shared_port* will check for both an abstract socket and a named socket in the directory indicated by this variable. The named socket allows command-line tools such as *condor_ssh_to_job* to use *condor_shared_port* as described.

On Linux platforms, setting SHARED_PORT_AUDIT_LOG causes HTCondor to log the following information about each connection made through the DAEMON_SOCKET_DIR: the source address, the socket file name, and the target process's PID, UID, GID, executable path, and command line. An administrator may use this logged information to deter abuse.

The default value is auto, causing the use of the directory $(LOCK)/daemon_sock. On Unix platforms other than Linux, if that path is longer than the 90 characters maximum, then the *condor_master* will instead create a directory under /tmp with a name that looks like /tmp/condor_shared_port_<XXXXXX>, where <XXXXXX> is replaced with random characters. The *condor_master* then tells the other daemons the exact name of the directory it created, and they use it.

If a different value is set for DAEMON_SOCKET_DIR, then that directory is used, without regard for the length of the path name. Ensure that the length is not longer than 90 characters.

SHARED_PORT_ARGS Like all daemons started by the *condor_master* daemon, the command line arguments to the invocation of the *condor_shared_port* daemon can be customized. The arguments can be used to specify a non-default port number for the *condor_shared_port* daemon as in this example, which specifies port 4080:

```
SHARED_PORT_ARGS = -p 4080
```

It is recommended to use configuration variable SHARED_PORT_PORT to set a non-default port number, instead of using this configuration variable.

SHARED_PORT_AUDIT_LOG On Linux platforms, the path and file name of the *condor_shared_port* log that records connections made via the DAEMON_SOCKET_DIR. If not defined, there will be no *condor_shared_port* audit log.

MAX_SHARED_PORT_AUDIT_LOG On Linux platforms, controls the maximum amount of time that the *condor_shared_port* audit log will be allowed to grow. When it is time to rotate a log file, the log file will be saved to a file named with an ISO timestamp suffix. The oldest rotated file receives the file name suffix .old. The .old files are overwritten each time the maximum number of rotated files (determined by the value of MAX_NUM_SHARED_PORT_AUDIT_LOG) is exceeded. A value of 0 specifies that the file may grow without bounds. The following suffixes may be used to qualify the integer:

`Sec` for seconds

`Min` for minutes

`Hr` for hours

`Day` for days

`Wk` for weeks

MAX_NUM_SHARED_PORT_AUDIT_LOG On Linux platforms, the integer that controls the maximum number of rotations that the *condor_shared_port* audit log is allowed to perform, before the oldest one will be rotated away. The default value is 1.

3.5.32 Configuration File Entries Relating to Hooks

These macros control the various hooks that interact with HTCondor. Currently, there are two independent sets of hooks. One is a set of fetch work hooks, some of which are invoked by the *condor_startd* to optionally fetch work, and some are invoked by the *condor_starter*. See section 4.4.1 on page 539 on Job Hooks for more details. The other set replace functionality of the *condor_job_router* daemon. Documentation for the *condor_job_router* daemon is in section 5.4 on page 584.

SLOT<N>_JOB_HOOK_KEYWORD For the fetch work hooks, the keyword used to define which set of hooks a particular compute slot should invoke. The value of `<N>` is replaced by the slot identification number. For example, on slot 1, the variable name will 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 For the fetch work hooks, 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 job hooks must explicitly define the keyword and the corresponding hook paths.

<Keyword>_HOOK_FETCH_WORK For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* wants to fetch work. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_REPLY_FETCH For the fetch work hooks, the full path to the program to invoke when the hook defined by `<Keyword>_HOOK_FETCH_WORK` returns data and the the *condor_startd* decides if it is going to accept the fetched job or not. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_REPLY_CLAIM For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* finishes fetching a job and decides what to do with it. `<Keyword>` is the hook keyword defined to distinguish

between sets of hooks. There is no default.

<Keyword>_HOOK_PREPARE_JOB For the fetch work hooks, the full path to the program invoked by the *condor_starter* before it runs the job. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_UPDATE_JOB_INFO This configuration variable is used by both fetch work hooks and by *condor_job_router* hooks.

For the fetch work hooks, the full path to the program invoked by the *condor_starter* periodically as the job runs, allowing the *condor_starter* to present an updated and augmented job ClassAd to the program. See section 4.4.1 on page 540 for the list of additional attributes included. When the job is first invoked, the *condor_starter* will invoke the program after `$(STARTER_INITIAL_UPDATE_INTERVAL)` seconds. Thereafter, the *condor_starter* will invoke the program every `$(STARTER_UPDATE_INTERVAL)` seconds. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks.

As a Job Router hook, the full path to the program invoked when the Job Router polls the status of routed jobs at intervals set by `JOB_ROUTER_POLLING_PERIOD`. `<Keyword>` is the hook keyword defined by `JOB_ROUTER_HOOK_KEYWORD` to identify the hooks.

<Keyword>_HOOK_EVICT_CLAIM For the fetch work hooks, the full path to the program to invoke whenever the *condor_startd* needs to evict a fetched claim. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks. There is no default.

<Keyword>_HOOK_JOB_EXIT For the fetch work hooks, the full path to the program invoked by the *condor_starter* whenever a job exits, either on its own or when being evicted from an execution slot. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks.

<Keyword>_HOOK_JOB_EXIT_TIMEOUT For the fetch work hooks, the number of seconds the *condor_starter* will wait for the hook defined by `<Keyword>_HOOK_JOB_EXIT` hook to exit, before continuing with job clean up. Defaults to 30 seconds. `<Keyword>` is the hook keyword defined to distinguish between sets of hooks.

FetchWorkDelay An expression that defines the number of seconds that the *condor_startd* should wait after an invocation of `<Keyword>_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 544.

JOB_ROUTER_HOOK_KEYWORD For the Job Router hooks, the keyword used to define the set of hooks the *condor_job_router* is to invoke to replace functionality of routing translation. There is no default keyword. Use of these hooks requires the explicit definition of the keyword and the corresponding hook paths.

<Keyword>_HOOK_TRANSLATE_JOB A Job Router hook, the full path to the program 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. `<Keyword>` is the hook keyword defined by `JOB_ROUTER_HOOK_KEYWORD` to identify the hooks.

<Keyword>_HOOK_JOB_FINALIZE A Job Router hook, the full path to the program invoked when the Job Router has determined that the job completed. `<Keyword>` is the hook keyword defined by `JOB_ROUTER_HOOK_KEYWORD` to identify the hooks.

<Keyword>_HOOK_JOB_CLEANUP A Job Router hook, the full path to the program invoked when the Job Router finishes managing the job. `<Keyword>` is the hook keyword defined by `JOB_ROUTER_HOOK_KEYWORD` to identify the hooks.

The following macros describe the *Daemon ClassAd Hook* capabilities of HTCondor. The Daemon ClassAd Hook mechanism is used to run executables (called jobs) directly from the *condor_startd* and *condor_schedd* daemons. The output from the jobs is incorporated into the machine ClassAd generated by the respective daemon. The mechanism is described in section 4.4.3 on page 548.

STARTD_CRON_NAME and SCHEDD_CRON_NAME These variables will be honored through HTCondor versions 7.6, and support will be removed in HTCondor version 7.7. They are no longer documented as to their usage.

Defines a logical name to be used in the formation of related configuration macro names. This macro made other Daemon ClassAd Hook macros more readable and maintainable. A common example was

```
STARTD_CRON_NAME = HAWKEYE
```

This example allowed the naming of other related macros to contain the string `HAWKEYE` in their name, replacing the string `STARTD_CRON`.

The value of these variables may not be `BENCHMARKS`. The Daemon ClassAd Hook mechanism is used to implement a set of provided hooks that provide benchmark attributes.

STARTD_CRON_CONFIG_VAL and SCHEDD_CRON_CONFIG_VAL and BENCHMARKS_CONFIG_VAL This configuration variable can be used to specify the path and executable name of the *condor_config_val* program which the jobs (hooks) should use to get configuration information from the daemon. If defined, an environment variable by the same name with the same value will be passed to all jobs.

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 jobs produce output. Beware that enabling this setting can greatly increase the network traffic in an HTCondor 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 variable:

Never This default value causes the *condor_startd* to not automatically publish updates based on any jobs. Instead, updates rely on the usual behavior for sending updates, which is periodic, based on the `UPDATE_INTERVAL` configuration variable, or whenever a given slot changes state.

Always Causes the *condor_startd* to always send a new update to the *condor_collector* whenever any job exits.

If_Changed Causes the *condor_startd* to only send a new update to the *condor_collector* if the output produced by a given job is different than the previous output of the same job. The only exception is the `LastUpdate` attribute, which is automatically set for all jobs to be the timestamp when the job last ran. It is ignored when `STARTD_CRON_AUTOPUBLISH` is set to `If_Changed`.

STARTD_CRON_JOBLIST and SCHEDD_CRON_JOBLIST and BENCHMARKS_JOBLIST These configuration variables are defined by a comma and/or white space separated list of job names to run. Each is the logical name of a job. This name must be unique; no two jobs may have the same name.

STARTD_CRON_<JobName>_PREFIX and SCHEDD_CRON_<JobName>_PREFIX and BENCHMARKS_<JobName>_PREF
Specifies a string which is prepended by HTCondor to all attribute names that the job generates. The use of prefixes avoids the conflicts that would be caused by attributes of the same name generated and utilized by different jobs. For example, if a module prefix is `xyz_`, and an individual attribute is named `abc`, then the resulting attribute name will be `xyz_abc`. Due to restrictions on ClassAd names, a prefix is only permitted to contain alpha-numeric characters and the underscore character.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_SLOTS and BENCHMARKS_<JobName>_SLOTS Only the slots specified in this comma-separated list may incorporate the output of the job specified by <JobName>. If the list is not specified, any slot may. Whether or not a specific slot actually incorporates the output depends on the output; see 4.4.3.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_EXECUTABLE and SCHEDD_CRON_<JobName>_EXECUTABLE and BENCHMARKS_<JobNa
The full path and executable to run for this job. Note that multiple jobs may specify the same executable, although the jobs need to have different logical names.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_PERIOD and SCHEDD_CRON_<JobName>_PERIOD and BENCHMARKS_<JobName>_PERI
The period specifies time intervals at which the job should be run. For periodic jobs, this is the time interval that passes between starting the execution of the job. The value may be specified in seconds, minutes, or hours. Specify this time by appending the character s, m, or h to the value. As an example, 5m starts the execution of the job every five minutes. If no character is appended to the value, seconds are used as a default. In WaitForExit mode, the value has a different meaning: the period specifies the length of time after the job ceases execution and before it is restarted. The minimum valid value of the period is 1 second.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_MODE and SCHEDD_CRON_<JobName>_MODE and BENCHMARKS_<JobName>_MODE
A string that specifies a mode within which the job operates. Legal values are

- Periodic, which is the default.
- WaitForExit
- OneShot
- OnDemand

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

The default Periodic mode is used for most jobs. In this mode, the job is expected to be started by the *condor_startd* daemon, gather and publish its data, and then exit.

In WaitForExit mode the *condor_startd* daemon interprets the period as defined by STARTD_CRON_<JobName>_PERIOD differently. In this case, it refers to the amount of time to wait after the job exits before restarting it. With a value of 1, the job is kept running nearly continuously. In general, WaitForExit mode is for jobs that produce a periodic stream of updated data, but it can be used for other purposes, as well. The output data from the job is accumulated into a temporary ClassAd until the job exits or until it writes a line starting with dash (−) character. At that point, the temporary ClassAd replaces the active ClassAd for the job. The active ClassAd for the job is merged into the appropriate slot ClassAds whenever the slot ClassAds are published.

The OneShot mode is used for jobs that are run once at the start of the daemon. If the reconfig_rerun option is specified, the job will be run again after any reconfiguration.

The OnDemand mode is used only by the BENCHMARKS mechanism. All benchmark jobs must be be OnDemand jobs. Any other jobs specified as OnDemand will never run. Additional future features may allow for other OnDemand job uses.

STARTD_CRON_<JobName>_RECONFIG and SCHEDD_CRON_<JobName>_RECONFIG A boolean value that when True, causes the daemon to send an HUP signal to the job when the daemon is reconfigured. The job is expected to reread its configuration at that time.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST or SCHEDD_CRON_JOBLIST.

STARTD_CRON_<JobName>_RECONFIG_RERUN and SCHEDD_CRON_<JobName>_RECONFIG_RERUN A boolean value that when True, causes the daemon ClassAd hooks mechanism to re-run the specified job when the daemon is reconfigured via *condor_reconfig*. The default value is False.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST or SCHEDD_CRON_JOBLIST.

STARTD_CRON_<JobName>_JOB_LOAD and SCHEDD_CRON_<JobName>_JOB_LOAD and BENCHMARKS_<JobName>_
A floating point value that represents the assumed and therefore expected CPU load that a job induces on the system. This job load is then used to limit the total number of jobs that run concurrently, by not starting new jobs if the assumed total load from all jobs is over a set threshold. The default value for each individual STARTD_CRON or a SCHEDD_CRON job is 0.01. The default value for each individual BENCHMARKS job is 1.0.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_MAX_JOB_LOAD and SCHEDD_CRON_MAX_JOB_LOAD and BENCHMARKS_MAX_JOB_LOAD
A floating point value representing a threshold for CPU load, such that if starting another job would cause the sum of assumed loads for all running jobs to exceed this value, no further jobs will be started. The default value for STARTD_CRON or a SCHEDD_CRON hook managers is 0.1. This implies that a maximum of 10 jobs (using their default, assumed load) could be concurrently running. The default value for the BENCHMARKS hook manager is 1.0. This implies that only 1 BENCHMARKS job (at the default, assumed load) may be running.

STARTD_CRON_<JobName>_KILL and SCHEDD_CRON_<JobName>_KILL and BENCHMARKS_<JobName>_KILL
A boolean value applicable only for jobs with a MODE of anything other than WaitForExit. The default value is False.

This variable controls the behavior of the daemon hook manager when it detects that an instance of the job's executable is still running as it is time to invoke the job again. If True, the daemon hook manager will kill the currently running job and then invoke an new instance of the job. If False, the existing job invocation is allowed to continue running.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_ARGS and SCHEDD_CRON_<JobName>_ARGS and BENCHMARKS_<JobName>_ARGS
The command line arguments to pass to the job as it is invoked. The first argument will be <JobName>.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_ENV and SCHEDD_CRON_<JobName>_ENV and BENCHMARKS_<JobName>_ENV The environment string to pass to the job. The syntax is the same as that of <DaemonName>_ENVIRONMENT as defined at 3.5.7.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

STARTD_CRON_<JobName>_CWD and SCHEDD_CRON_<JobName>_CWD and BENCHMARKS_<JobName>_CWD The working directory in which to start the job.

<JobName> is the logical name assigned for a job as defined by configuration variable STARTD_CRON_JOBLIST, SCHEDD_CRON_JOBLIST, or BENCHMARKS_JOBLIST.

3.5.33 Configuration File Entries Only for Windows Platforms

These macros are utilized only on Windows platforms.

WINDOWS_RMDIR The complete path and executable name of the HTCondor version of the built-in *rmdir* program. The HTCondor version will not fail when the directory contains files that have ACLs that deny the SYSTEM process delete access. If not defined, the built-in Windows *rmdir* program is invoked, and a value defined for WINDOWS_RMDIR_OPTIONS is ignored.

WINDOWS_RMDIR_OPTIONS Command line options to be specified when configuration variable WINDOWS_RMDIR is defined. Defaults to **/S /C** when configuration variable WINDOWS_RMDIR is defined and its definition contains the string "condor_rmdir.exe".

3.5.34 condor_defrag Configuration File Macros

These configuration variables affect the *condor_defrag* daemon. A general discussion of *condor_defrag* may be found in section 3.7.1.

DEFRAG_NAME Used to give an alternative value to the Name attribute in the *condor_defrag*'s ClassAd. This esoteric configuration macro might be used in the situation where there are two *condor_defrag* 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 3.5.7 on page 242 for defaults and composition of valid HTCondor daemon names.

DEFRAG_DRAINING_MACHINES_PER_HOUR A floating point number that specifies how many machines should be drained per hour. The default is 0, so no draining will happen unless this setting is changed. Each *condor_startd* is considered to be one machine. The actual number of machines drained per hour may be less than this if draining is halted by one of the other defragmentation policy controls. The granularity in timing of draining initiation is controlled by DEFRAG_INTERVAL. The lowest rate of draining that is supported is one machine per day or one

machine per DEFRAG_INTERVAL, whichever is lower. A fractional number of machines contributing to the value of DEFRAG_DRAINING_MACHINES_PER_HOUR is rounded to the nearest whole number of machines on a per day basis.

DEFRAG_REQUIREMENTS An expression that specifies which machines to drain. The default is

```
PartitionableSlot && Offline=!=True
```

A machine, meaning a *condor_startd*, is matched if *any* of its slots match this expression. Machines are automatically excluded if they are already draining, or if they match DEFRAG_WHOLE_MACHINE_EXPR.

DEFRAG_CANCEL_REQUIREMENTS An expression that specifies which draining machines should have draining be canceled. This defaults to $(DEFRAG_WHOLE_MACHINE_EXPR). This could be used to drain partial rather than whole machines.

DEFRAG_RANK An expression that specifies which machines are more desirable to drain. The expression should evaluate to a number for each candidate machine to be drained. If the number of machines to be drained is less than the number of candidates, the machines with higher rank will be chosen. The rank of a machine, meaning a *condor_startd*, is the rank of its highest ranked slot. The default rank is -ExpectedMachineGracefulDrainingBadput.

DEFRAG_WHOLE_MACHINE_EXPR An expression that specifies which machines are already operating as whole machines. The default is

```
Cpus == TotalCpus && Offline=!=True
```

A machine is matched if *any* slot on the machine matches this expression. Each *condor_startd* is considered to be one machine. Whole machines are excluded when selecting machines to drain. They are also counted against DEFRAG_MAX_WHOLE_MACHINES.

DEFRAG_MAX_WHOLE_MACHINES An integer that specifies the maximum number of whole machines. When the number of whole machines is greater than or equal to this, no new machines will be selected for draining. Each *condor_startd* is counted as one machine. The special value -1 indicates that there is no limit. The default is -1.

DEFRAG_MAX_CONCURRENT_DRAINING An integer that specifies the maximum number of draining machines. When the number of machines that are draining is greater than or equal to this, no new machines will be selected for draining. Each draining *condor_startd* is counted as one machine. The special value -1 indicates that there is no limit. The default is -1.

DEFRAG_INTERVAL An integer that specifies the number of seconds between evaluations of the defragmentation policy. In each cycle, the state of the pool is observed and machines are drained, if specified by the policy. The default is 600 seconds. Very small intervals could create excessive load on the *condor_collector*.

DEFRAG_UPDATE_INTERVAL An integer that specifies the number of seconds between times that the *condor_defrag* daemon sends updates to the collector. (See section 12 for information about the attributes in these updates.) The default is 300 seconds.

DEFRAG_SCHEDULE A setting that specifies the draining schedule to use when draining machines. Possible values are graceful, quick, and fast. The default is graceful.

graceful Initiate a graceful eviction of the job. This means all promises that have been made to the job are honored, including `MaxJobRetirementTime`. The eviction of jobs is coordinated to reduce idle time. This means that if one slot has a job with a long retirement time and the other slots have jobs with shorter retirement times, the effective retirement time for all of the jobs is the longer one.

quick `MaxJobRetirementTime` is not honored. Eviction of jobs is immediately initiated. Jobs are given time to shut down and produce a checkpoint according to the usual policy, as given by `MachineMaxVacateTime`.

fast Jobs are immediately hard-killed, with no chance to gracefully shut down or produce a checkpoint.

DEFRAG_STATE_FILE The path to a file used to record information used by *condor_defrag* when it is restarted. This should only need to be modified if there will be multiple instances of the *condor_defrag* daemon running on the same machine. The default is `$(LOCK)/defrag_state`.

DEFRAG_LOG The path to the *condor_defrag* daemon's log file. The default log location is `$(LOG)/DefragLog`.

3.5.35 *condor_gangliad* Configuration File Macros

condor_gangliad is an optional daemon responsible for publishing information about HTCondor daemons to the Ganglia™ monitoring system. The Ganglia monitoring system must be installed and configured separately. In the typical case, a single instance of the *condor_gangliad* daemon is run per pool. A default set of metrics are sent. Additional metrics may be defined, in order to publish any information available in ClassAds that the *condor_collector* daemon has.

GANGLIAD_INTERVAL The integer number of seconds between consecutive sending of metrics to Ganglia. Daemons update the *condor_collector* every 300 seconds, and the Ganglia heartbeat interval is 20 seconds. Therefore, multiples of 20 between 20 and 300 makes sense for this value. Negative values inhibit sending data to Ganglia. The default value is 60.

GANGLIAD_VERBOSITY An integer that specifies the maximum verbosity level of metrics to be published to Ganglia. Basic metrics have a verbosity level of 0, which is the default. Additional metrics can be enabled by increasing the verbosity to 1. In the default configuration, there are no metrics with verbosity levels higher than 1. Some metrics depend on attributes that are not published to the *condor_collector* when using the default value of `STATISTICS_TO_PUBLISH`. For example, per-user file transfer statistics will only be published to Ganglia if `GANGLIA_VERBOSITY` is set to 1 or higher in the *condor_gangliad* configuration *and* `STATISTICS_TO_PUBLISH` in the *condor_schedd* configuration contains `TRANSFER:2`, or if the `STATISTICS_TO_PUBLISH_LIST` contains the desired attributes explicitly.

GANGLIAD_REQUIREMENTS An optional boolean ClassAd expression that may restrict the set of daemon ClassAds to be monitored. This could be used to monitor a subset of a pool's daemons or machines. The default is an empty expression, which has the effect of placing no restriction on the monitored ClassAds. Keep in mind that this expression is applied to all types of monitored ClassAds, not just machine ClassAds.

GANGLIAD_PER_EXECUTE_NODE_METRICS A boolean value that, when `False`, causes metrics from execute node daemons to not be published. Aggregate values from these machines will still be published. The default value is `True`. This option is useful for pools such that use glidein, in which it is not desired to record metrics for individual execute nodes.

GANGLIA_CONFIG The path and file name of the Ganglia configuration file. The default is `/etc/ganglia/gmond.conf`.

GANGLIA_GMETRIC The full path of the *gmetric* executable to use. If none is specified, `libganglia` will be used instead when possible, because the library interface is more efficient than invoking *gmetric*. Some versions of `libganglia` are not compatible. When a failure to use `libganglia` is detected, *gmetric* will be used, if *gmetric* can be found in HTCondor's `PATH` environment variable.

GANGLIA_GSTAT_COMMAND The full *gstat* command used to determine which hosts are monitored by Ganglia. For a *condor_gangliad* running on a host whose local *gmond* does not know the list of monitored hosts, change `localhost` to be the appropriate host name or IP address within this default string:

```
gstat --all --mpifile --gmond_ip=localhost --gmond_port=8649
```

GANGLIA_SEND_DATA_FOR_ALL_HOSTS A boolean value that when `True` causes data to be sent to Ganglia for hosts that it is not currently monitoring. The default is `False`.

GANGLIA_LIB The full path and file name of the `libganglia` shared library to use. If none is specified, and if configuration variable `GANGLIA_GMETRIC` is also not specified, then a search for `libganglia` will be performed in the directories listed in configuration variable `GANGLIA_LIB_PATH` or `GANGLIA_LIB64_PATH`. The special value `NOOP` indicates that *condor_gangliad* should not publish statistics to Ganglia, but should otherwise go through all the motions it normally does.

GANGLIA_LIB_PATH A comma-separated list of directories within which to search for the `libganglia` executable, if `GANGLIA_LIB` is not configured. This is used in 32-bit versions of HTCondor.

GANGLIA_LIB64_PATH A comma-separated list of directories within which to search for the `libganglia` executable, if `GANGLIA_LIB` is not configured. This is used in 64-bit versions of HTCondor.

GANGLIAD_DEFAULT_CLUSTER An expression specifying the default name of the Ganglia cluster for all metrics. The expression may refer to attributes of the machine.

GANGLIAD_DEFAULT_MACHINE An expression specifying the default machine name of Ganglia metrics. The expression may refer to attributes of the machine.

GANGLIAD_DEFAULT_IP An expression specifying the default IP address of Ganglia metrics. The expression may refer to attributes of the machine.

GANGLIAD_LOG The path and file name of the *condor_gangliad* daemon's log file. The default log is `$(LOG)/GangliadLog`.

GANGLIAD_METRICS_CONFIG_DIR Path to the directory containing files which define Ganglia metrics in terms of HTCondor ClassAd attributes to be published. All files in this directory are read, to define the metrics. The default directory `/etc/condor/ganglia.d/` is used when not specified.

3.6 User Priorities and Negotiation

HTCondor 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 HTCondor 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 HTCondor. Documentation on configuring and controlling priorities may be found in section 3.5.15.

3.6.1 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 has no running jobs, that user's RUP would be 5 one day later, 2.5 two days later, and so on.

3.6.2 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, an initial priority factor for all users as they first submit jobs is set by the configuration variable `DEFAULT_PRIO_FACTOR`, and defaults to the value 1000.0. If desired, the priority factors of specific users can be increased using *condor_userprio*, so that some 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 resources that A may be given, the available resources are repartitioned and distributed among remaining users according to the inverse ratio rule.

HTCondor supplies mechanisms to directly support two policies in which EUP may be useful:

Nice users A job may be submitted with the submit command **nice_user** set to `True`. This nice user job will have its RUP boosted by the `NICE_USER_PRIO_FACTOR` priority factor specified in the configuration, leading to a very large EUP. This corresponds to a low priority for resources, therefore using resources not used by other HTCondor users.

Remote Users HTCondor's flocking feature (see section 5.2) allows jobs to run in a pool other than the local one. In addition, the submit-only feature allows a user to submit jobs to another pool. In such situations, submitters from other domains can submit to the local pool. It may be desirable to have HTCondor treat local users preferentially over these remote users. If configured, HTCondor will boost the RUPs of remote users by `REMOTE_PRIO_FACTOR` specified in the configuration, 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 964.

3.6.3 Priorities in Negotiation and Preemption

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time, meaning during negotiation and matchmaking. Therefore, there are ClassAd attributes that take on defined values only during negotiation, making them ephemeral. In addition to allocation, HTCondor 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. When preemption is enabled, 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 3.7.1 for the current default policy on preemption.

The following ephemeral attributes may be used within policy definitions. Care should be taken when using these attributes, due to their ephemeral nature; they are not always defined, so the usage of an expression to check if defined such as

```
(RemoteUserPrio =?= UNDEFINED)
```

is likely necessary.

Within these attributes, those with names that contain the string `Submitter` refer to characteristics about the candidate job's user; those with names that contain 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 and `PREEMPTION_RANK_STABLE` exist to inform the *condor_negotiator* daemon that values may change. See section 3.5.15 on page 293 for definitions of these configuration variables.

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. This version of the attribute, with no slot represented in the attribute name, refers to the current slot being evaluated.

Slot<N>_RemoteUserPrio: A floating point value representing the user priority of the job currently running on the particular slot represented by <N> 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.

SubmitterGroup: The accounting group name of the requesting submitter.

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.

RemoteGroup: The accounting group name of the owner of the currently running job.

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.

SubmitterNegotiatingGroup: The accounting group name that the candidate job is negotiating under.

RemoteNegotiatingGroup: The accounting group name that the currently running job negotiated under.

SubmitterAutoregroup: Boolean attribute is `True` if candidate job is negotiated via autoregoup.

RemoteAutoregroup: Boolean attribute is `True` if currently running job negotiated via autoregoup.

3.6.4 Priority Calculation

This section may be skipped if the reader so feels, but for the curious, here is HTCondor'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 HTCondor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

3.6.5 Negotiation

Negotiation is the method HTCondor 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 3.6.2 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`.

3.6.6 The Layperson's Description of the Pie Spin and Pie Slice

HTCondor 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 HTCondor 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. When preemption is enabled, a reasonable policy 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 received 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.

3.6.7 Group Accounting

By default, HTCondor does all accounting on a per-user basis, and this accounting is primarily used to compute priorities for HTCondor'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 jobs with the same accounting group will be treated with the same priority. Jobs that do *not* specify an accounting group have all accounting and priority based on the user, which may be identified by the job ClassAd attribute `Owner`. Jobs that do specify an accounting group have all accounting and priority based on the specified accounting group. Therefore, accounting based on groups only works when the jobs correctly identify their group membership.

The preferred method for having a job associate itself with an accounting group adds a command to the submit description file that specifies the group name:

```
accounting_group = group_physics
```

This command causes the job ClassAd attribute `AcctGroup` to be set with this group name.

If the user name of the job submitter should be other than the `Owner` job ClassAd attribute, an additional command specifies the user name:

```
accounting_group_user = albert
```

This command causes the job ClassAd attribute `AcctGroupUser` to be set with this user name.

The previous method for defining accounting groups is no longer recommended. It inserted the job ClassAd attribute `AccountingGroup` by setting it in the submit description file using the syntax in this example:

```
+AccountingGroup = "group_physics.albert"
```

In this previous method for defining accounting groups, the `AccountingGroup` attribute is a string, and it therefore must be enclosed in double quote marks.

Much of the reason that the previous method for defining accounting groups is no longer recommended is that the name of an accounting is that it used the period (.) character to separate the group name from the user name. Therefore, the syntax did not work if a user name contained a period.

The name should *not* be qualified with a domain. Certain parts of the HTCondor 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

```
                                  Effective
User Name                         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 HTCondor. 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
```

HTCondor 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.

3.6.8 Accounting Groups with Hierarchical Group Quotas

An upper limit on the number of slots allocated to a group of users can be specified with group quotas. This policy may be desired when different groups provide their computers to create one large HTCondor pool, and want to restrict the number of jobs running from one group to the number of machines the group has provided.

Consider an example pool with thirty slots: twenty slots are owned by the physics group and ten are owned by the chemistry group. The desired policy is that no more than twenty concurrent jobs are ever running from the physicists, and only ten from the chemists. These machines are otherwise identical, so it does not matter which machines run which group's jobs. It only matters that the proportions of allocated slots are correct.

Instead of quotas, this could be implemented by configuring the `RANK` expression such that the twenty machines owned by the physics group prefer jobs submitted by the physics users. Likewise, the ten machines owned by the chemistry group are configured to prefer jobs submitted by the chemistry group. However, this steers jobs to execute on specific machines, instead of the desired policy which allocates numbers of machines, where these machines can be any of the pool's machines that are available.

Group quotas may implement this policy. Define the groups and set their quotas in the configuration of the central manager:

```
GROUP_NAMES = group_physics, group_chemistry
GROUP_QUOTA_group_physics =   20
GROUP_QUOTA_group_chemistry = 10
```

The implementation of quotas is hierarchical, such that quotas may be described for the tree of groups, subgroups, sub subgroups, etc. Group names identify the groups, such that the configuration can define the quotas in terms of limiting the number of cores allocated for a group or subgroup. Group names do not need to begin with `"group_"`, but that is the convention, which helps to avoid naming conflicts between groups and subgroups. The hierarchy is identified by using the period ('.') character to separate a group name from a subgroup name from a sub subgroup name, etc. Group names are case-insensitive for negotiation.

At the root of the tree that defines the hierarchical groups is the invented "<none>" group. The implied quota of the "<none>" group will be all available slots. This string will appear in the output of *condor_status*.

If the sum of the child quotas exceeds the parent, then the child quotas are scaled down in proportion to their relative sizes. For the given example, there were 30 original slots at the root of the tree. If a power failure removed half of the original 30, leaving fifteen slots, physics would be scaled back to a quota of ten, and chemistry to five. This scaling can be disabled by setting the *condor_negotiator* configuration variable `NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION` to `True`. If the sum of the child quotas is less than that of the parent, the child quotas remain intact; they are not scaled up. That is, if somehow the number of slots doubled from thirty to sixty, physics would still be limited to 20 slots, and chemistry would be limited to 10. This example in which the quota is defined by absolute values is called a static quota.

Each job must state which group it belongs to. Currently this is opt-in, and the system trusts each user to put the correct group in the submit description file. Jobs that do not identify themselves as a group member are negotiated for as part of the "<none>" group. Note that this requirement is per job, not per user. A given user may be a member of many groups. Jobs identify which group they are in by setting the **accounting_group** and **accounting_group_user**

commands within the submit description file, as specified in section 3.6.7. For example:

```
accounting_group = group_physics
accounting_group_user = einstein
```

The size of the quotas may instead be expressed as a proportion. This is then referred to as a dynamic group quota, because the size of the quota is dynamically recalculated every negotiation cycle, based on the total available size of the pool. Instead of using static quotas, this example can be recast using dynamic quotas, with one-third of the pool allocated to chemistry and two-thirds to physics. The quotas maintain this ratio even as the size of the pool changes, perhaps because of machine failures, because of the arrival of new machines within the pool, or because of other reasons. The job submit description files remain the same. Configuration on the central manager becomes:

```
GROUP_NAMES = group_physics, group_chemistry
GROUP_QUOTA_DYNAMIC_group_chemistry = 0.33
GROUP_QUOTA_DYNAMIC_group_physics =   0.66
```

The values of the quotas must be less than 1.0, indicating fractions of the pool's machines. As with static quota specification, if the sum of the children exceeds one, they are scaled down proportionally so that their sum does equal 1.0. If their sum is less than one, they are not changed.

Extending this example to incorporate subgroups, assume that the physics group consists of high-energy (hep) and low-energy (lep) subgroups. The high-energy sub-group owns fifteen of the twenty physics slots, and the low-energy group owns the remainder. Groups are distinguished from subgroups by an intervening period character (`.`) in the group's name. Static quotas for these subgroups extend the example configuration:

```
GROUP_NAMES = group_physics, group_physics.hep, group_physics.lep, group_chemistry
GROUP_QUOTA_group_physics     =   20
GROUP_QUOTA_group_physics.hep =   15
GROUP_QUOTA_group_physics.lep =    5
GROUP_QUOTA_group_chemistry   =   10
```

This hierarchy may be more useful when dynamic quotas are used. Here is the example, using dynamic quotas:

```
GROUP_NAMES = group_physics, group_physics.hep, group_physics.lep, group_chemistry
GROUP_QUOTA_DYNAMIC_group_chemistry   =   0.33334
GROUP_QUOTA_DYNAMIC_group_physics     =   0.66667
GROUP_QUOTA_DYNAMIC_group_physics.hep =   0.75
GROUP_QUOTA_DYNAMIC_group_physics.lep =   0.25
```

The fraction of a subgroup's quota is expressed with respect to its parent group's quota. That is, the high-energy physics subgroup is allocated 75% of the 66% that physics gets of the entire pool, however many that might be. If there are 30 machines in the pool, that would be the same 15 machines as specified in the static quota example.

High-energy physics users indicate which group their jobs should go in with the submit description file identification:

```
accounting_group = group_physics.hep
accounting_group_user = higgs
```

In all these examples so far, the hierarchy is merely a notational convenience. Each of the examples could be implemented with a flat structure, although it might be more confusing for the administrator. Surplus is the concept that creates a true hierarchy.

If a given group or sub-group accepts surplus, then that given group is allowed to exceed its configured quota, by using the leftover, unused quota of other groups. Surplus is disabled for all groups by default. Accepting surplus may be enabled for all groups by setting GROUP_ACCEPT_SURPLUS to True. Surplus may be enabled for individual groups by setting GROUP_ACCEPT_SURPLUS_<groupname> to True. Consider the following example:

```
GROUP_NAMES = group_physics, group_physics.hep, group_physics.lep, group_chemistry
GROUP_QUOTA_group_physics     =   20
GROUP_QUOTA_group_physics.hep =   15
GROUP_QUOTA_group_physics.lep =    5
GROUP_QUOTA_group_chemistry   =   10
GROUP_ACCEPT_SURPLUS = false
GROUP_ACCEPT_SURPLUS_group_physics = false
GROUP_ACCEPT_SURPLUS_group_physics.lep = true
GROUP_ACCEPT_SURPLUS_group_physics.hep = true
```

This configuration is the same as above for the chemistry users. However, GROUP_ACCEPT_SURPLUS is set to False globally, False for the physics parent group, and True for the subgroups group_physics.lep and group_physics.lep. This means that group_physics.lep and group_physics.hep are allowed to exceed their quota of 15 and 5, but their sum cannot exceed 20, for that is their parent's quota. If the group_physics had GROUP_ACCEPT_SURPLUS set to True, then either group_physics.lep and group_physics.hep would not be limited by quota.

Surplus slots are distributed bottom-up from within the quota tree. That is, any leaf nodes of this tree with excess quota will share it with any peers which accept surplus. Any subsequent excess will then be passed up to the parent node and over to all of its children, recursively. Any node that does not accept surplus implements a hard cap on the number of slots that the sum of it's children use.

After the *condor_negotiator* calculates the quota assigned to each group, possibly adding in surplus, it then negotiates with the *condor_schedd* daemons in the system to try to match jobs to each group. It does this one group at a time. By default, it goes in "starvation group order." That is, the group whose current usage is the smallest fraction of its quota goes first, then the next, and so on. The "<none>" group implicitly at the root of the tree goes last. This ordering can be replaced by defining configuration variable GROUP_SORT_EXPR. The *condor_negotiator* evaluates this ClassAd expression for each group ClassAd, sorts the groups by the floating point result, and then negotiates with the smallest positive value going first. Useful attributes to use are documented in section 3.6.3.

One possible group quota policy is strict priority. For example, a site prefers physics users to match as many slots as they can, and only when all the physics jobs are running, and idle slots remain, are chemistry jobs allowed to run. The default "starvation group order" can be used to implement this. By setting configuration variable NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION to True, and setting the physics quota to a number so large that it cannot ever be met, such as one million, the physics group will always be the "most starving" group, will always negotiate first, and will always be unable to meet the quota. Only when all the physics jobs are running will the chemistry jobs then run. If the chemistry quota is set to a value smaller than physics, but still larger than the pool, this policy can support a third, even lower priority group, and so on.

The *condor_userprio* command can show the current quotas in effect, and the current usage by group. For example:

```
$ condor_userprio -quotas
Last Priority Update: 11/12 15:18
Group                          Effective  Config     Use    Subtree  Requested
Name                             Quota     Quota   Surplus   Quota   Resources
------------------------------ --------- --------- ------- --------- ----------
group_physics.hep                  15.00     15.00 no          15.00         60
group_physics.lep                   5.00      5.00 no           5.00         60
------------------------------ --------- --------- ------- --------- ----------
Number of users: 2                                 ByQuota
```

This shows that there are two groups, each with 60 jobs in the queue. group_physics.hep has a quota of 15 machines, and group_physics.lep has 5 machines. Other options to *condor_userprio*, such as **-most** will also show the number of resources in use.

3.7 Policy Configuration for Execute Hosts and for Submit Hosts

Note: configuration templates make it easier to implement certain policies; see information on policy templates here: 3.4.2.

3.7.1 *condor_startd* Policy Configuration

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. This policy is the heart of HTCondor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to HTCondor). Please read this section carefully before changing any of the settings described here, as a wrong setting can have a severe impact on either the owners of machines in the pool or the users of the pool.

condor_startd Terminology

Understanding the configuration requires an understanding of ClassAd expressions, which are detailed in section 4.1.

Each machine runs one *condor_startd* daemon. Each machine may contain one or more cores (or CPUs). The HTCondor construct of a *slot* describes the unit which is matched to a job. Each slot may contain one or more integer number of cores. Each slot is represented by its own machine ClassAd, distinguished by the machine ClassAd attribute `Name`, which is of the form `slot<N>@hostname`. The value for `<N>` will also be defined with machine ClassAd attribute `SlotID`.

Each slot has its own machine ClassAd, and within that ClassAd, its own *state* and *activity*. Other policy expressions are propagated or inherited from the machine configuration by the *condor_startd* daemon, such that all slots have

the same policy from the machine configuration. This requires configuration expressions to incorporate the SlotID attribute when policy is intended to be individualized based on a slot. So, in this discussion of policy expressions, where a machine is referenced, the policy can equally be applied to a slot.

The *condor_startd* daemon represents the machine on which it is running to the HTCondor 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
```

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 or slot 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.

For platforms that support standard universe 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 ClassAd, 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 expression 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 3.7.1. However, the machine locally evaluates the IS_OWNER expression to determine if it is capable of running jobs for HTCondor. 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 HTCondor 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, HTCondor jobs always run and can never be kicked off from activity on the machine. However, because they would run with the low priority, interactive response on the machines will not suffer. A machine user probably would not notice that HTCondor was running the jobs, assuming you had enough free memory for the HTCondor jobs such that there was little swapping.

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 **and**ed 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 HTCondor configuration files.

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  (TARGET.JobUniverse =!= 1) ||

  (
    (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.

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 job ClassAd. 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 HTCondor pool in the department, and this small research group has spent a lot of money on really fast machines for the group. As part of the larger pool, but to implement a policy that gives priority on the fast machines to anyone in the small research group, set the RANK expression on the machines to reference the Owner attribute and prefer requests where that attribute matches one of the people in the 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 the group has priority on their fast machines. Its difference is that the machine owner has better priority on their own machine. To set this up for Garrison's machine (bass), place the following entry in the local configuration file of machine bass:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
  + ((Owner == "garrison") * 10) + (Owner == "jones")
```

Note that the parentheses in this expression are important, because the + operator has higher default precedence than ==.

The use of + instead of || allows us to distinguish which terms matched and which ones did not. If anyone not in the research group 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 the bass machine, assuming START evaluated to True for him at that time. The RANK would numerically evaluate to 1. Therefore, the Elvin Jones job could preempt the

HTCondor job currently running. Further assume that later Jimmy Garrison submits a job. The RANK evaluates to 10 on machine bass, since the boolean that matches gets multiplied by 10. Due to this, Jimmy Garrison's job could preempt Elvin Jones' job on the bass machine where Jimmy Garrison's jobs are preferred.

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. Perhaps configure this 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 the RANK on Coltrane's machine that prefers John Coltrane's jobs over those with the largest Imagesize:

```
RANK = (Owner == "coltrane" * 1000000000000) + Imagesize
```

This RANK does not work 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 HTCondor's only problem!

Machine States

A machine is assigned a *state* by HTCondor. The state depends on whether or not the machine is available to run HTCondor 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 HTCondor jobs. When the machine first starts up, it begins in this state.

Unclaimed The machine is available to run HTCondor 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 an HTCondor job. This state is only entered if the machine is specifically configured to enable backfill jobs.

Drained The machine is not running jobs, because it is being drained. One reason a machine may be drained is to consolidate resources that have been divided in a partitionable slot. Consolidating the resources gives large jobs a chance to run.

Figure 3.1 shows the states and the possible transitions between the states.

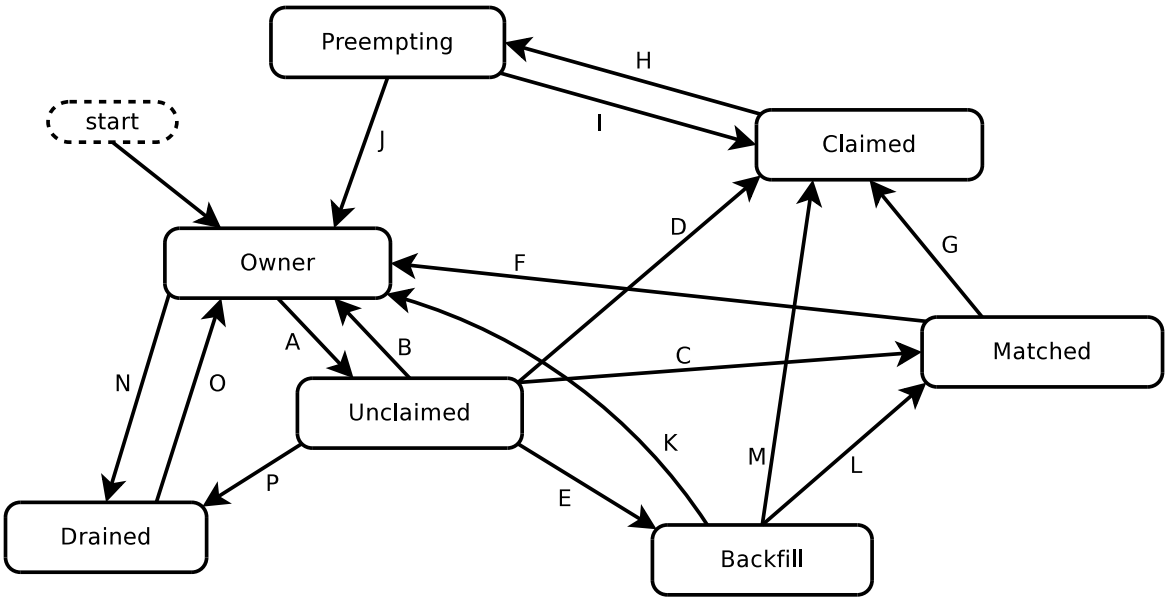

Figure 3.1: Machine States

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 an HTCondor job.

  N The machine switches from the Owner to the Drained state whenever draining of the machine is initiated, for example by *condor_drain* or by the *condor_defrag* daemon.

- 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 an HTCondor 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 an HTCondor job.

D The transition from Unclaimed directly to Claimed also happens if the *condor_negotiator* matches this resource with an HTCondor 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 3.14.9) and the `START_BACKFILL` expression evaluates to TRUE.

P The transition from Unclaimed to Drained happens if draining of the machine is initiated, for example by *condor_drain* or by the *condor_defrag* daemon.

- 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 an HTCondor 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*.

- Transitions out of the Drained state

O The transition from Drained to Owner state happens when draining is finalized or is canceled. When a draining request is made, the request either asks for the machine to stay in a Drained state until canceled, or it asks for draining to be automatically finalized once all slots have finished draining.

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 40 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*.

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 HTCondor is concerned the machine is Idle, since it is not doing anything for HTCondor.

- Unclaimed

  Idle This is the normal activity of Unclaimed machines. The machine is still Idle in that the machine owner is willing to let HTCondor jobs run, but HTCondor 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 HTCondor.

- 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 HTCondor is concerned.

  Suspended If the job is suspended by HTCondor, 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 HTCondor 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 an HTCondor 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 HTCondor job.

- Drained

  Idle All slots have been drained.

  Retiring This slot has been drained. It is waiting for other slots to finish draining.

Figure 3.2 on page 362 gives the overall view of all machine states and activities and shows the possible transitions from one to another within the HTCondor 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 HTCondor 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.

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, HTCondor 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
```

So, the machine will remain in the Owner state as long as the `START` expression locally evaluates to FALSE. Section 3.7.1 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 HTCondor system. From HTCondor'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 HTCondor 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

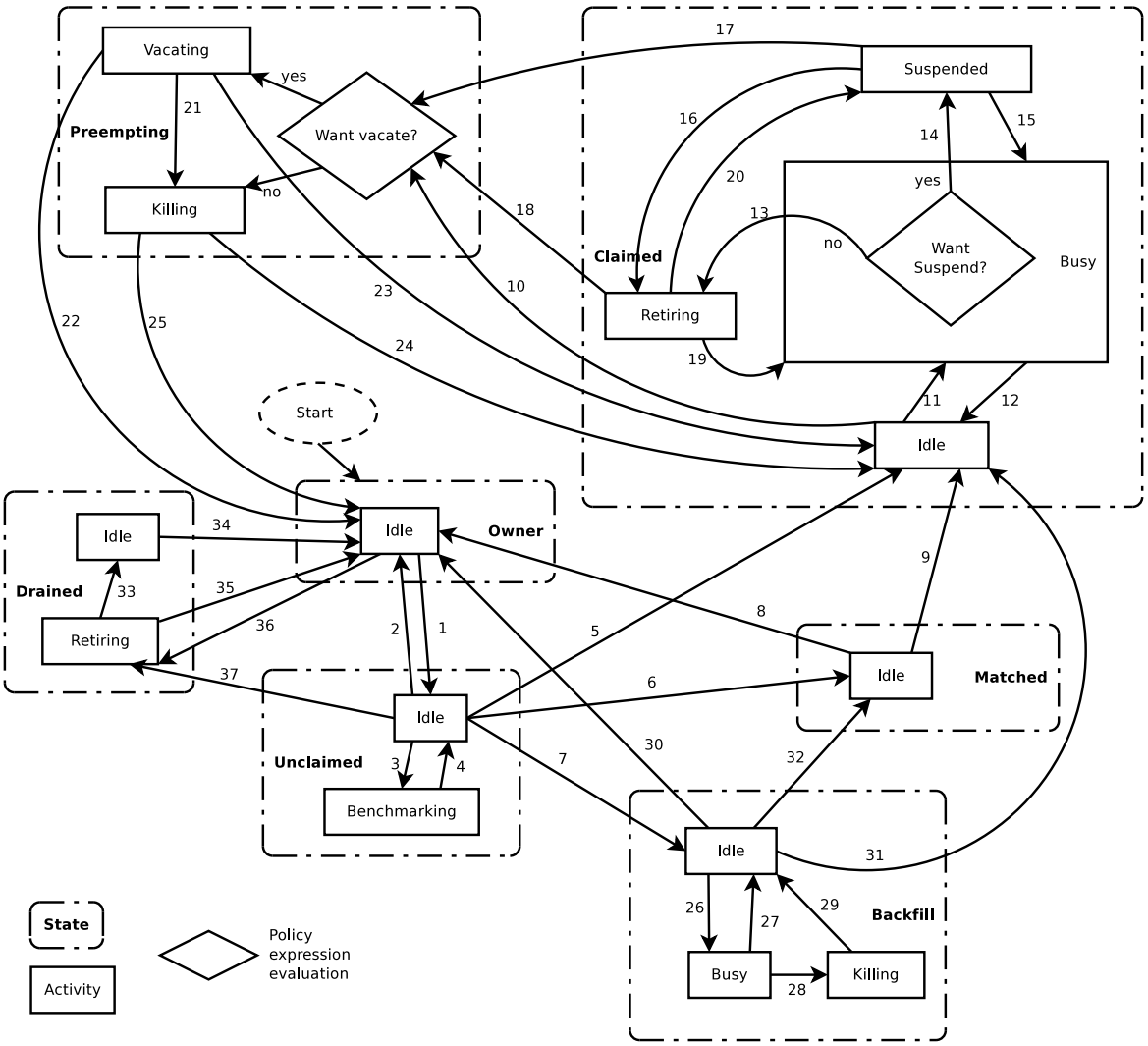

Figure 3.2: Machine States and Activities

the two states are how the resource shows up in *condor_status* and other reporting tools, and the fact that HTCondor 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.

If draining of the machine is initiated while in the Owner state, the slot transitions to Drained/Retiring (transition **36**).

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:

```
RunBenchmarks = (time() - LastBenchmark) >= (4 * $(HOUR))
```

This macro calculates 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 3.14.9), 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.

If draining of the machine is initiated while in the Unclaimed state, the slot transitions to Drained/Retiring (transition **37**).

Matched State

The Matched state is not very interesting to HTCondor. 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 resource 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 multi-core 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 HTCondor 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 HTCondor. 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 any value other than TRUE, 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 the value defined for the job ClassAd attribute `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 *condor_startd* daemon shutdown or restart, it is possible for the administrator to issue a *peaceful* shutdown command, which causes `MaxJobRetirementTime` to effectively be infinite, avoiding any killing of jobs. It is also possible for the administrator to issue a *fast* shutdown command, which causes `MaxJobRetirementTime` to be effectively 0.

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 *condor_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**). If the Vacating activity lasts for as long as the maximum vacating time, then the machine also enters the Killing activity. The maximum vacating time is determined by the configuration variable `MachineMaxVacateTime`. This may be adjusted by the setting of the job ClassAd attribute `JobMaxVacateTime`.

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 HTCondor job. Once the machine has begun to hard kill the HTCondor 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 HTCondor, see section 3.14.9 on page 479. 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 HTCondor 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 HTCondor). 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 multi-core 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 a multi-core 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 an HTCondor 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 an HTCondor 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 *condor_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**).

Drained State

The Drained state is used when the machine is being drained, for example by *condor_drain* or by the *condor_defrag* daemon, and the slot has finished running jobs and is no longer willing to run new jobs.

Slots initially enter the Drained/Retiring state. Once all slots have been drained, the slots transition to the Idle activity (transition **33**).

If draining is finalized or canceled, the slot transitions to Owner/Idle (transitions **34** and **35**).

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 HTCondor job.

RUNBENCHMARKS While in the Unclaimed state, the machine will run benchmarks whenever TRUE.

MATCH_TIMEOUT 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 any value other than True, the machine will 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.

CLAIM_WORKLIFE This expression specifies the number of seconds after which a claim will stop accepting additional jobs. This configuration macro is fully documented here: 3.5.8.

MachineMaxVacateTime When the machine enters the Preempting/Vacating state, this expression specifies the maximum time in seconds that the *condor_startd* will wait for the job to finish. The job may adjust the wait time by setting `JobMaxVacateTime`. If the job's setting is less than the machine's, the job's is used. If the job's setting is larger than the machine's, the result depends on whether the job has any excess retirement time. If the job has more retirement time left than the machine's maximum vacate time setting, then retirement time will be converted into vacating time, up to the amount of `JobMaxVacateTime`. Once the vacating time expires, the job is hard-killed. The `KILL` expression may be used to abort the graceful shutdown of the job at any time.

MAXJOBRETIREMENTTIME If the machine is in the Claimed/Retiring state, jobs which have run for less than the number of seconds specified by this expression will not be hard-killed. The *condor_startd* will wait for the job to finish or to exceed this amount of time, whichever comes sooner. Time spent in suspension does not count against the job. If the job vacating policy grants the job X seconds of vacating time, a preempted job will be soft-killed X seconds before the end of its retirement time, so that hard-killing of the job will not happen until the end of the retirement time if the job does not finish shutting down before then. The job may provide its own expression for `MaxJobRetirementTime`, but this can only be used to take *less* than the time granted by the *condor_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.

The machine enters the Preempting state with the goal of finishing shutting down the job by the end of the retirement time. If the job vacating policy grants the job X seconds of vacating time, the transition to the Preempting state will happen X seconds before the end of the retirement time, so that the hard-killing of the job will not happen until the end of the retirement time, if the job does not finish shutting down before then.

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.

By default the *condor_negotiator* will not match jobs to a slot with retirement time remaining. This behavior is controlled by `NEGOTIATOR_CONSIDER_EARLY_PREEMPTION`.

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 *condor_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 may preempt 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.

Examples of Policy Configuration

This section describes various policy configurations, including the default policy.

Default Policy

These settings are the default as shipped with HTCondor. 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 seconds in the current state.

ActivityTimer Amount of time in seconds in the current activity.

ActivationTimer Amount of time in seconds that 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 HTCondor load (the load generated by everything but HTCondor).

BackgroundLoad Amount of background load permitted on the machine and still start an HTCondor job.

HighLoad If the `$(NonCondorLoadAvg)` goes over this, the CPU is considered too busy, and eviction of the HTCondor job should start.

StartIdleTime Amount of time the keyboard must to be idle before HTCondor 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.

KeyboardBusy A boolean expression that evaluates to TRUE when the keyboard is being used.

CPUIdle 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 variable definitions exist in the example configuration file in order to help write legible expressions. They are not required, and perhaps will go unused by many configurations.

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (time() - EnteredCurrentState)
ActivityTimer   = (time() - EnteredCurrentActivity)
ActivationTimer = (time() - JobStart)
LastCkpt        = (time() - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)
```

Preemption is disabled as a default. Always desire to start jobs.

```
WANT_SUSPEND         = False
WANT_VACATE          = False
START                = True
SUSPEND              = False
CONTINUE             = True
PREEMPT              = False
# Kill jobs that take too long leaving gracefully.
MachineMaxVacateTime = 10 * $(MINUTE)
KILL                 = False
```

Periodic checkpointing specifies that for jobs smaller than 60 Mbytes, take a periodic checkpoint every 6 hours. For larger jobs, only take a 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))
```

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

HTCondor can be configured to only run jobs at certain times of the day. In general, we discourage configuring a system like this, since there will often be lots of good cycles on 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 produce checkpoints, 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, use the `ClockMin` and `ClockDay` 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.

To force HTCondor jobs to stay off of your machines during work hours:

```
# Only start jobs after hours.
START = $(AfterHours)

# Consider the machine busy during work hours, or if the keyboard or
# CPU are busy.
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

This `MachineBusy` macro is convenient if other than the default `SUSPEND` and `PREEMPT` expressions are used.

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 the global configuration file) 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 an HTCondor job
# 2) AND the user is not active (if a desktop)
START = ( ($(CPUIdle) || (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 = ( $(CPUIdle) && ($(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
MachineMaxVacateTime = 10 * $(MINUTE)
KILL = False
```

With this policy in the global configuration, 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 and Enabling Preemption

Preemption causes a running job to be suspended or killed, such that another job can run. As of HTCondor version 8.1.5, preemption is disabled by the default configuration. Previous versions of HTCondor had configuration that enabled preemption. Upon upgrade, the previous behavior will continue, if the previous configuration files are used. New configuration file examples disable preemption, but contain directions for enabling preemption.

Job Suspension

As new jobs are submitted that receive a higher priority than currently executing jobs, the executing jobs may be preempted. If the preempted jobs are not capable of writing checkpoints, they 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. An alternative to this is to use suspension to freeze the job while some other task runs, and then unfreeze it so that it can continue on from where it left off. This does not require any special handling in the job, unlike most strategies that take checkpoints. However, it does require a special configuration of HTCondor. This example implements a policy that allows the job to decide whether it should be evicted or suspended. The jobs announce their choice through the use of the invented job ClassAd attribute `IsSuspendableJob`, that is also utilized in the configuration.

The implementation of this policy utilizes two categories of slots, identified as suspendable or nonsuspendable. A job identifies which category of slot it wishes to run on. This affects two aspects of the policy:

- Of two jobs that might run on a slot, which job is chosen. The four cases that may occur depend on whether the currently running job identifies itself as suspendable or nonsuspendable, and whether the potentially running job identifies itself as suspendable or nonsuspendable.
    1. If the currently running job is one that identifies itself as suspendable, and the potentially running job identifies itself as nonsuspendable, the currently running job is suspended, in favor of running the nonsuspendable one. This occurs independent of the user priority of the two jobs.
    2. If both the currently running job and the potentially running job identify themselves as suspendable, then the relative priorities of the users and the preemption policy determines whether the new job will replace the existing job.
    3. If both the currently running job and the potentially running job identify themselves as nonsuspendable, then the relative priorities of the users and the preemption policy determines whether the new job will replace the existing job.
    4. If the currently running job is one that identifies itself as nonsuspendable, and the potentially running job identifies itself as suspendable, the currently running job continues running.
- What happens to a currently running job that is preempted. A job that identifies itself as suspendable will be suspended, which means it is frozen in place, and will later be unfrozen when the preempting job is finished. A job that identifies itself as nonsuspendable is evicted, which means it writes a checkpoint, when possible, and then is killed. The job will return to the idle state in the job queue, and it can try to run again in the future.

```
# Lie to HTCondor, to achieve 2 slots for each real slot
NUM_CPUS = $(DETECTED_CORES)*2
# There is no good way to tell HTCondor that the two slots should be treated
# as though they share the same real memory, so lie about how much
# memory we have.
MEMORY = $(DETECTED_MEMORY)*2

# Slots 1 through DETECTED_CORES are nonsuspendable and the rest are
# suspendable
IsSuspendableSlot = SlotID > $(DETECTED_CORES)

# If I am a suspendable slot, my corresponding nonsuspendable slot is
```

```
# my SlotID plus $(DETECTED_CORES)
NonSuspendableSlotState = eval(strcat("slot",SlotID-$(DETECTED_CORES),"_State")

# The above expression looks at slotX_State, so we need to add
# State to the list of slot attributes to advertise.
STARTD_SLOT_ATTRS = $(STARTD_SLOT_ATTRS) State

# For convenience, advertise these expressions in the machine ad.
STARTD_ATTRS = $(STARTD_ATTRS) IsSuspendableSlot NonSuspendableSlotState

MyNonSuspendableSlotIsIdle = \
  (NonSuspendableSlotState =!= "Claimed" && NonSuspendableSlotState =!= "Preempting")

# NonSuspendable slots are always willing to start jobs.
# Suspendable slots are only willing to start if the NonSuspendable slot is idle.
START = \
  IsSuspendableSlot!=True && IsSuspendableJob=!=True || \
  IsSuspendableSlot && IsSuspendableJob==True && $(MyNonSuspendableSlotIsIdle)

# Suspend the suspendable slot if the other slot is busy.
SUSPEND = \
  IsSuspendableSlot && $(MyNonSuspendableSlotIsIdle)!=True

WANT_SUSPEND = $(SUSPEND)

CONTINUE = ($(SUSPEND)) != True
```

Note that in this example, the job ClassAd attribute `IsSuspendableJob` has no special meaning to HTCondor. It is an invented name chosen for this example. To take advantage of the policy, a job that wishes to be suspended must submit the job so that this attribute is defined. The following line should be placed in the job's submit description file:

```
+IsSuspendableJob = True
```

Configuration for Interactive Jobs

Policy may be set based on whether a job is an interactive one or not. Each interactive job has the job ClassAd attribute

```
InteractiveJob = True
```

and this may be used to identify interactive jobs, distinguishing them from all other jobs.

As an example, presume that slot 1 prefers interactive jobs. Set the machine's `RANK` to show the preference:

```
RANK = ( (MY.SlotID == 1) && (TARGET.InteractiveJob =?= True) )
```

Or, if slot 1 should be reserved for interactive jobs:

```
START = ( (MY.SlotID == 1) && (TARGET.InteractiveJob =?= True) )
```

Multi-Core Machine Terminology

Machines with more than one CPU or core 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.

Multi-core machines are represented to the HTCondor system as shared resources broken up into individual *slots*. Each slot can be matched and claimed by users for jobs. Each slot is represented by an individual machine ClassAd. In this way, each multi-core machine will appear to the HTCondor system as a collection of separate slots. As an example, a multi-core machine named `vulture.cs.wisc.edu` would appear to HTCondor 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, such as RAM, disk space, and swap space, can be divided evenly among all the slots, with each slot assigned one core. Alternatively, *slot types* are defined by configuration, so that resources can be unevenly divided. Regardless of the scheme used, it is important to remember that the goal is to create a representative slot ClassAd, to be used for matchmaking with jobs.

HTCondor 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. HTCondor 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 policy that will evict a job that oversubscribes to shared resources, as described in section 3.7.1.

Dividing System Resources in Multi-core Machines

Within a machine the shared system resources of cores, RAM, swap space and disk space will be divided for use by the slots. There are two main ways to go about dividing the resources of a multi-core machine:

Evenly divide all resources. By default, the *condor_startd* will automatically divide the machine into slots, placing one core in each slot, and evenly dividing all shared resources among the slots. The only specification may be how many slots are reported at a time. By default, all slots are reported to HTCondor.

How many slots are reported at a time 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 cores within the machine. Variable `NUM_SLOTS` may not be used to make HTCondor advertise more slots than there are cores on the machine. The number of cores is defined by `NUM_CPUS`.

Define slot types. Instead of an even division of resources per slot, the machine may have definitions of *slot types*, where each type is provided with a fraction of shared system resources. Given the slot type definition, control how many of each type are reported at any given time with further configuration.

Configuration variables define the slot types, as well as variables that list how much of each system resource goes to each slot type.

Configuration variable `SLOT_TYPE_<N>`, where `<N>` is an integer (for example, `SLOT_TYPE_1`) defines the slot type. Note that there may be multiple slots of each type. The number of slots created of a given type is

configured with NUM_SLOTS_TYPE_<N>.

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 that includes a blanket value that serves as a default for any resources not explicitly specified in the list.

A simple fraction or percentage describes the allocation of the total system resources, including the number of CPUS or cores. A comma separated list allows a fine tuning of the amounts for specific resources.

The number of CPUs and the total amount of RAM in the machine do not change over time. 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.

```
SLOT_TYPE_1 = mem=64

SLOT_TYPE_1 = mem=1/2

SLOT_TYPE_1 = mem=50%

SLOT_TYPE_1 = mem=auto
```

Amounts of disk space and swap space are dynamic, as they change over time. 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 the 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 slot's EXECUTE or SLOT<N>_EXECUTE directory. 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 predefined attribute names are case insensitive when defining slot types. The first letter of the attribute name distinguishes between these attributes. The four attributes, with several examples of acceptable names for each:

- 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 machine with 4 cores and 256 Mbytes 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, all of these slot types would not be used together, because they add up to more than 100% of the various system resources. This configuration example also omits definitions of NUM_SLOTS_TYPE_<N>, to define the number of 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 explicitly allocated resources given 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`, number of cores 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%
```

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.

In addition to the standard resources of CPUs, memory, disk, and swap, the administrator may also define custom resources on a localized per-machine basis.

The resource names and quantities of available resources are defined using configuration variables of the form `MACHINE_RESOURCE_<name>`, as shown in this example:

```
MACHINE_RESOURCE_gpu = 16
MACHINE_RESOURCE_actuator = 8
```

If the configuration uses the optional configuration variable `MACHINE_RESOURCE_NAMES` to enable and disable local machine resources, also add the resource names to this variable. For example:

```
if defined MACHINE_RESOURCE_NAMES
  MACHINE_RESOURCE_NAMES = $(MACHINE_RESOURCE_NAMES) gpu actuator
endif
```

Local machine resource names defined in this way may now be used in conjunction with `SLOT_TYPE_<N>`, using all the same syntax described earlier in this section. The following example demonstrates the definition of static and partitionable slot types with local machine resources:

```
# declare one partitionable slot with half of the GPUs, 6 actuators, and
# 50% of all other resources:
SLOT_TYPE_1 = gpu=50%,actuator=6,50%
SLOT_TYPE_1_PARTITIONABLE = TRUE
NUM_SLOTS_TYPE_1 = 1

# declare two static slots, each with 25% of the GPUs, 1 actuator, and
# 25% of all other resources:
SLOT_TYPE_2 = gpu=25%,actuator=1,25%
SLOT_TYPE_2_PARTITIONABLE = FALSE
NUM_SLOTS_TYPE_2 = 2
```

A job may request these local machine resources using the syntax **request_<name>**, as described in section 3.7.1. This example shows a portion of a submit description file that requests GPUs and an actuator:

```
universe = vanilla

# request two GPUs and one actuator:
request_gpu = 2
request_actuator = 1

queue
```

The slot ClassAd will represent each local machine resource with the following attributes:

`Total<name>`: the total quantity of the resource identified by `<name>`

`Detected<name>`: the quantity detected of the resource identified by `<name>`; this attribute is currently equivalent to `Total<name>`

`TotalSlot<name>`: the quantity of the resource identified by `<name>` allocated to this slot

`<name>`: the amount of the resource identified by `<name>` available to be used on this slot

From the example given, the `gpu` resource would be represented by the ClassAd attributes `TotalGpu`, `DetectedGpu`, `TotalSlotGpu`, and `Gpu`. In the job ClassAd, the amount of the requested machine resource appears in a job ClassAd attribute named `Request<name>`. For this example, the two attributes will be `RequestGpu` and `RequestActuator`.

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. To change any slot type definitions, use *condor_restart*

```
condor_restart -startd
```

for that change to take effect.

Configuration Specific to Multi-core Machines

Each slot within a multi-core machine is treated as an independent machine, each with its own view of its state as represented by the machine ClassAd attribute `State`. The policy expressions for the multi-core machine as a whole are propagated from the *condor_startd* to the slot's machine ClassAd. This policy may consider a 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 multi-core machine 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. A multi-core machine with the default policy settings can add the keyboard and console to be noticed by only one slot. Assuming a reasonable load average, 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 continue running them, 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.

This example policy is controlled with the following configuration variables.

- `SLOTS_CONNECTED_TO_CONSOLE`, with definition at section 3.5.8
- `SLOTS_CONNECTED_TO_KEYBOARD`, with definition at section 3.5.8
- `DISCONNECTED_KEYBOARD_IDLE_BOOST`, with definition at section 3.5.8

Each slot has its own machine ClassAd. Yet, the policy expressions for the multi-core machine are propagated and inherited from configuration of the *condor_startd*. Therefore, the policy expressions for each slot are the same. This makes the implementation of certain types of policies impossible, because while evaluating the state of one slot within the multi-core machine, the state of other slots are not available. Decisions for one slot cannot be based on what other slots 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 by all of the slots on the 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 ClassAd attribute `State`, a different machine ClassAd, and if there is a job running, a separate job ClassAd. 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 ClassAd, it *always* refers to the ClassAd of the job running on the specific slot.

To set a different policy for the slots within a machine, incorporate the slot-specific machine ClassAd attribute `SlotID`. A `SUSPEND` policy that is different for each of the two slots 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 Multi-core Machines

Most operating systems define the load average for a multi-core machine as the total load on all cores. For example, a 4-core machine with 3 CPU-bound processes running at the same time will have a load of 3.0. In HTCondor, we maintain this view of the total load average and publish it in all resource ClassAds as `TotalLoadAvg`.

HTCondor also provides a per-core load average for multi-core machines. This nicely represents the model that each node on a multi-core machine is a slot, separate from the other nodes. All of the default, single-core policy expressions can be used directly on multi-core machines, without modification, since the `LoadAvg` and `CondorLoadAvg` attributes are the per-slot versions, not the total, multi-core wide versions.

The per-core load average on multi-core machines is an HTCondor invention. No system call exists to ask the operating system for this value. HTCondor already computes the load average generated by HTCondor on each slot. It does this by close monitoring of all processes spawned by any of the HTCondor daemons, even ones that are orphaned and then inherited by *init*. This HTCondor load average per slot is reported as the attribute `CondorLoadAvg` in all resource ClassAds, and the total HTCondor 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, HTCondor walks through all the slots and assigns out portions of the total load average to each one. First, HTCondor assigns the known HTCondor load average to each node that is generating load. If there is any load average left in the total system load, it is considered an owner load. Any slots HTCondor believes are in the Owner state, such as ones that have keyboard activity, are the first to get assigned this owner load. HTCondor hands out owner load in increments of at most 1.0, so generally speaking, no slot has a load average above 1.0. If HTCondor runs out of total load average before it runs out of slots, all the remaining machines believe that they have no load average at all. If, instead, HTCondor runs out of slots and it still has owner load remaining, HTCondor starts assigning that load to HTCondor nodes as well, giving individual nodes with a load average higher than 1.0.

Debug Logging in the Multi-Core *condor_startd* Daemon

This section describes how the *condor_startd* daemon handles its debugging messages for multi-core machines. In general, a given log message will either be something that is machine-wide, such as 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 word printed out in the entry with the slot number. 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-core machine with no HTCondor 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
```

Shared system resources are printed without the header, such as total swap space, and slot-specific messages, such as the load average or state of each slot, get the slot number appended.

Configuring GPUs

HTCondor supports incorporating GPU resources and making them available for jobs. First, GPUs must be detected as available resources. Then, machine ClassAd attributes advertise this availability. Both detection and advertisement are accomplished by having this configuration for each execute machine that has GPUs:

```
use feature : GPUs
```

Use of this configuration templdate invokes the *condor_gpu_discovery* tool to create a custom resource, with a custom resource name of `GPUs`, and it generates the ClassAd attributes needed to advertise the GPUs. *condor_gpu_discovery* is invoked in a mode that discovers and advertises both CUDA and OpenCL GPUs.

This configuration template refers to macro `GPU_DISCOVERY_EXTRA`, which can be used to define additional command line arguments for the *condor_gpu_discovery* tool. For example, setting

```
use feature : GPUs
GPU_DISCOVERY_EXTRA = -extra
```

causes the *condor_gpu_discovery* tool to output more attributes that describe the detected GPUs on the machine.

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. SLOT<N>_STARTD_ATTRS
4. SLOT<N>_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 Provisioning: Partitionable and Dynamic Slots

Dynamic provisioning, also referred to as partitionable or dynamic slots, allows HTCondor to use the resources of a slot in a dynamic way; these slots may be partitioned. This means that more than one job can occupy a single slot at any one time. Slots have a fixed set of resources which include the cores, memory and disk space. By partitioning the slot, the use of these resources becomes more flexible.

Here is an example that demonstrates how resources are divided as more than one job is or can be matched to a single slot. In this example, Slot1 is identified as a partitionable slot and 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 carved out is now known as a dynamic slot. This dynamic slot has its own machine ClassAd, and its `Name` attribute distinguishes itself as a dynamic slot with incorporating the substring `Slot1_1`.

After allocation, the partitionable Slot1 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`, `Slot1_3` etc., until the entire set of Slot1's available resources have been consumed by jobs.

To enable dynamic provisioning, define a slot type. and declare at least one slot of that type. Then, identify that slot type as partitionable by setting configuration variable `SLOT_TYPE_<N>_PARTITIONABLE` to `True`. The value of `<N>` within the configuration variable name is the same value as in slot type definition configuration variable `SLOT_TYPE_<N>`. For the most common cases the machine should be configured for one slot, managing all the resources on the machine. To do so, set the following configuration variables:

```
NUM_SLOTS = 1
NUM_SLOTS_TYPE_1 = 1
SLOT_TYPE_1 = 100%
SLOT_TYPE_1_PARTITIONABLE = TRUE
```

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

request_cpus = 3
request_memory = 1024
request_disk = 10240

queue
```

Each partitionable slot will have the ClassAd attributes

```
PartitionableSlot = True
SlotType = "Partitionable"
```

Each dynamic slot will have the ClassAd attributes

```
DynamicSlot = True
SlotType = "Dynamic"
```

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. If matched jobs consume all its resources, the partitionable slot will eventually show as having no available resources; this will prevent further matching of new jobs. The dynamic slots will show as running jobs. The dynamic slots can be preempted in the same way as all other slots.

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. A partial solution to this problem is provided by defragmentation accomplished by the *condor_defrag* daemon, as discussed in section 3.7.1.

Another partial solution is a new matchmaking algorithm in the negotiator, referred to as *partitionable slot preemption*, or *pslot preemption*. Without pslot preemption, when the negotiator searches for a match for a job, it

looks at each slot ClassAd individually. With pslot preemption, the negotiator looks at a partitionable slot and all of its dynamic slots as a group. If the partitionable slot does not have sufficient resources (memory, cpu, and disk) to be matched with the candidate job, then the negotiator looks at all of the related dynamic slots that the candidate job might preempt (following the normal preemption rules described elsewhere). The resources of each dynamic slot are added to those of the partitionable slot, one dynamic slot at a time. Once this partial sum of resources is sufficient to enable a match, the negotiator sends the match information to the *condor_schedd*. When the *condor_schedd* claims the partitionable slot, the dynamic slots are preempted, such that their resources are returned to the partitionable slot for use by the new job.

To enable pslot preemption, the following configuration variable must be set for the *condor_negotiator*:

```
ALLOW_PSLOT_PREEMPTION = True
```

When the negotiator examines the resources of dynamic slots, it sorts the slots by their `CurrentRank` attribute, such that slots with lower values are considered first. The negotiator only examines the cpu, memory and disk resources of the dynamic slots; custom resources are ignored.

Dynamic slots that have retirement time remaining are not considered eligible for preemption, regardless of how configuration variable `NEGOTIATOR_CONSIDER_EARLY_PREEMPTION` is set.

When pslot preemption is enabled, the negotiator will not preempt dynamic slots directly. It will preempt them only as part of a match to a partitionable slot.

When multiple partitionable slots match a candidate job and the various job rank expressions are evaluated to sort the matching slots, the ClassAd of the partitionable slot is used for evaluation. This may cause unexpected results for some expressions, as attributes such as `RemoteOwner` will not be present in a partitionable slot that matches with preemption of some of its dynamic slots.

Defaults for Partitionable Slot Sizes

If a job does not specify the required number of CPUs, amount of memory, or disk space, there are ways for the administrator to set default values for all of these parameters.

First, if any of these attributes are not set in the submit description file, there are three variables in the configuration file that condor_submit will use to fill in default values. These are

```
JOB_DEFAULT_REQUESTMEMORY
JOB_DEFAULT_REQUESTDISK
JOB_DEFAULT_REQUESTCPUS
```

The value of these variables can be ClassAd expressions. The default values for these variables, should they not be set are

```
JOB_DEFAULT_REQUESTMEMORY = ifThenElse(MemoryUsage =!= UNDEFINED, MemoryUsage, 1)
```

```
JOB_DEFAULT_REQUESTCPUS = 1

JOB_DEFAULT_REQUESTDISK = DiskUsage
```

Note that these default values are chosen such that jobs matched to partitionable slots function similar to static slots.

Once the job has been matched, and has made it to the execute machine, the *condor_startd* has the ability to modify these resource requests before using them to size the actual dynamic slots carved out of the partitionable slot. Clearly, for the job to work, the *condor_startd* daemon must create slots with at least as many resources as the job needs. However, it may be valuable to create dynamic slots somewhat bigger than the job's request, as subsequent jobs may be more likely to reuse the newly created slot when the initial job is done using it.

The *condor_startd* configuration variables which control this and their defaults are

```
MODIFY_REQUEST_EXPR_REQUESTCPUS = quantize(RequestCpus, {1})

MODIFY_REQUEST_EXPR_REQUESTMEMORY = quantize(RequestMemory, {128})

MODIFY_REQUEST_EXPR_REQUESTDISK = quantize(RequestDisk, {1024})
```

condor_negotiator-Side Resource Consumption Policies

For partitionable slots, the specification of a consumption policy permits matchmaking at the negotiator. A dynamic slot carved from the partitionable slot acquires the required quantities of resources, leaving the partitionable slot with the remainder. This differs from scheduler matchmaking in that multiple jobs can match with the partitionable slot during a single negotiation cycle.

All specification of the resources available is done by configuration of the partitionable slot. The machine is identified as having a resource consumption policy enabled with

```
CONSUMPTION_POLICY = True
```

A defined slot type that is partitionable may override the machine value with

```
SLOT_TYPE_<N>_CONSUMPTION_POLICY = True
```

A job seeking a match may always request a specific number of cores, amount of memory, and amount of disk space. Availability of these three resources on a machine and within the partitionable slot is always defined and have these default values:

```
CONSUMPTION_CPUS = quantize(target.RequestCpus,{1})
CONSUMPTION_MEMORY = quantize(target.RequestMemory,{128})
CONSUMPTION_DISK = quantize(target.RequestDisk,{1024})
```

Here is an example-driven definition of a consumption policy. Assume a single partitionable slot type on a multi-core machine with 8 cores, and that the resource this policy cares about allocating are the cores. Configuration for the machine includes the definition of the slot type and that it is partitionable.

```
SLOT_TYPE_1 = cpus=8
SLOT_TYPE_1_PARTITIONABLE = True
NUM_SLOTS_TYPE_1 = 1
```

Enable use of the *condor_negotiator*-side resource consumption policy, allocating the job-requested number of cores to the dynamic slot, and use `SLOT_WEIGHT` to assess the user usage that will affect user priority by the number of cores allocated. Note that the only attributes valid within the `SLOT_WEIGHT` expression are Cpus, Memory, and disk. This must the set to the same value on all machines in the pool.

```
SLOT_TYPE_1_CONSUMPTION_POLICY = True
SLOT_TYPE_1_CONSUMPTION_CPUS = TARGET.RequestCpus
SLOT_WEIGHT = Cpus
```

If custom resources are available within the partitionable slot, they may be used in a consumption policy, by specifying the resource. Using a machine with 4 GPUs as an example custom resource, define the resource and include it in the definition of the partitionable slot:

```
MACHINE_RESOURCE_NAMES = gpus
MACHINE_RESOURCE_gpus = 4
SLOT_TYPE_2 = cpus=8, gpus=4
SLOT_TYPE_2_PARTITIONABLE = True
NUM_SLOTS_TYPE_2 = 1
```

Add the consumption policy to incorporate availability of the GPUs:

```
SLOT_TYPE_2_CONSUMPTION_POLICY = True
SLOT_TYPE_2_CONSUMPTION_gpus = TARGET.RequestGpu
SLOT_WEIGHT = Cpus
```

Defragmenting Dynamic Slots

When partitionable slots are used, some attention must be given to the problem of the starvation of large jobs due to the fragmentation of resources. The problem is that over time the machine resources may become partitioned into slots suitable for running small jobs. If a sufficient number of these slots do not happen to become idle at the same time on a machine, then a large job will not be able to claim that machine, even if the large job has a better priority than the small jobs.

One way of addressing the partitionable slot fragmentation problem is to periodically drain all jobs from fragmented machines so that they become defragmented. The *condor_defrag* daemon implements a configurable policy for doing

that. Its implementation is targeted at machines configured to run whole-machine jobs and at machines that only have partitionable slots. The draining of a machine configured to have both partitionable slots and static slots would have a negative impact on single slot jobs running in static slots.

To use this daemon, `DEFRAG` must be added to `DAEMON_LIST`, and the defragmentation policy must be configured. Typically, only one instance of the *condor_defrag* daemon would be run per pool. It is a lightweight daemon that should not require a lot of system resources.

Here is an example configuration that puts the *condor_defrag* daemon to work:

```
DAEMON_LIST = $(DAEMON_LIST) DEFRAG
DEFRAG_INTERVAL = 3600
DEFRAG_DRAINING_MACHINES_PER_HOUR = 1.0
DEFRAG_MAX_WHOLE_MACHINES = 20
DEFRAG_MAX_CONCURRENT_DRAINING = 10
```

This example policy tells *condor_defrag* to initiate draining jobs from 1 machine per hour, but to avoid initiating new draining if there are 20 completely defragmented machines or 10 machines in a draining state. A full description of each configuration variable used by the *condor_defrag* daemon may be found in section 3.5.34.

By default, when a machine is drained, existing jobs are gracefully evicted. This means that each job will be allowed to use the remaining time promised to it by `MaxJobRetirementTime`. If the job has not finished when the retirement time runs out, the job will be killed with a soft kill signal, so that it has an opportunity to save a checkpoint (if the job supports this). No new jobs will be allowed to start while the machine is draining. To reduce unused time on the machine caused by some jobs having longer retirement time than others, the eviction of jobs with shorter retirement time is delayed until the job with the longest retirement time needs to be evicted.

There is a trade off between reduced starvation and throughput. Frequent draining of machines reduces the chance of starvation of large jobs. However, frequent draining reduces total throughput. Some of the machine's resources may go unused during draining, if some jobs finish before others. If jobs that cannot produce checkpoints are killed because they run past the end of their retirement time during draining, this also adds to the cost of draining.

To help gauge the costs of draining, the *condor_startd* advertises the accumulated time that was unused due to draining and the time spent by jobs that were killed due to draining. These are advertised respectively in the attributes `TotalMachineDrainingUnclaimedTime` and `TotalMachineDrainingBadput`. The *condor_defrag* daemon averages these values across the pool and advertises the result in its daemon ClassAd in the attributes `AvgDrainingBadput` and `AvgDrainingUnclaimed`. Details of all attributes published by the *condor_defrag* daemon are described in section 12.

The following command may be used to view the *condor_defrag* daemon ClassAd:

```
condor_status -l -any -constraint 'MyType == "Defrag"'
```

3.7.2 *condor_schedd* Policy Configuration

There are two types of schedd policy: job transforms (which change the ClassAd of a job at submission) and submit requirements (which prevent some jobs from entering the queue). These policies are explained below.

Job Transforms

The *condor_schedd* can transform jobs as they are submitted. Transformations can be used to guarantee the presence of required job attributes, to set defaults for job attributes the user does not supply, or to modify job attributes so that they conform to schedd policy; an example of this might be to automatically set accounting attributes based on the owner of the job while letting the job owner indicate a preference.

There can be multiple job transforms. Each transform can have a Requirements expression to indicate which jobs it should transform and which it should ignore. Transforms without a Requirements expression apply to all jobs. Job transforms are applied in order. The set of transforms and their order are configured using the Configuration variable `JOB_TRANSFORM_NAMES`.

For each entry in this list there must be a corresponding `JOB_TRANSFORM_<name>` configuration variable that specifies the transform rules. Transforms use the same syntax as *condor_job_router* transforms; although unlike the *condor_job_router* there is no default transform, and all matching transforms are applied - not just the first one. (See 5.4 for information on the *condor_job_router*.)

The following example shows a set of two transforms: one that automatically assigns an accounting group to jobs based on the submitting user, and one that shows one possible way to transform Vanilla jobs to Docker jobs.

```
JOB_TRANSFORM_NAMES = AssignGroup, SL6ToDocker

JOB_TRANSFORM_AssignGroup = [ eval_set_AccountingGroup = userMap("Groups",Owner,AccountingGroup); ]

JOB_TRANSFORM_SL6ToDocker @=end
[
   Requirements = JobUniverse==5 && WantSL6 && DockerImage =?= undefined;
   set_WantDocker = true;
   set_DockerImage = "SL6";
   copy_Requirements = "VanillaRequrements";
   set_Requirements = TARGET.HasDocker && VanillaRequirements
]
@end
```

The `AssignGroup` transform above assumes that a mapfile that can map an owner to one or more accounting groups has been configured via `SCHEDD_CLASSAD_USER_MAP_NAMES`, and given the name "Groups".

The `SL6ToDocker` transform above is most likely incomplete, as it assumes some custom attributes (`WantSL6` and `WantDocker` and `HasDocker`) that your pool may or may not use.

Submit Requirements

The *condor_schedd* may reject job submissions, such that rejected jobs never enter the queue. Rejection may be best for the case in which there are jobs that will never be able to run; an example of this might be all jobs that specify the standard universe in a queue with restricted networking. Another appropriate example might be to reject all jobs that do not request a minimum amount of memory. Or, it may be appropriate to prevent certain users from using a specific submit host.

Rejection criteria are configured. Configuration variable `SUBMIT_REQUIREMENT_NAMES` lists criteria, where each criterion is given a name. The chosen name is a major component of the default error message output if a user attempts to submit a job which fails to meet the requirements. Therefore, choose a descriptive name. For the three example submit requirements described:

```
SUBMIT_REQUIREMENT_NAMES = NotStandardUniverse, MinimalRequestMemory, NotChris
```

The criterion for each submit requirement is then specified in configuration variable `SUBMIT_REQUIREMENT_<Name>`, where `<Name>` matches the chosen name listed in `SUBMIT_REQUIREMENT_NAMES`. The value is a boolean ClassAd expression. The three example criterion result in these configuration variable definitions:

```
SUBMIT_REQUIREMENT_NotStandardUniverse = JobUniverse != 1
SUBMIT_REQUIREMENT_MinimalRequestMemory = RequestMemory > 512
SUBMIT_REQUIREMENT_NotChris = Owner != "chris"
```

Submit requirements are evaluated in the listed order; the first requirement that evaluates to `False` causes rejection of the job, terminates further evaluation of other submit requirements, and is the only requirement reported. Each submit requirement is evaluated in the context of the *condor_schedd* ClassAd, which is the `MY.` name space and the job ClassAd, which is the `TARGET.` name space. Note that `JobUniverse` and `RequestMemory` are both job ClassAd attributes.

Further configuration may associate a rejection reason with a submit requirement with the `SUBMIT_REQUIREMENT_<Name>_REASON`.

```
SUBMIT_REQUIREMENT_NotStandardUniverse_REASON = "This pool does not accept standard universe jobs."
SUBMIT_REQUIREMENT_MinimalRequestMemory_REASON = strcat( "The job only requested ", \
  RequestMemory, " Megabytes.  If that small amount is really enough, please contact ..." )
SUBMIT_REQUIREMENT_NotChris_REASON = "Chris, you may only submit jobs to the instructional pool."
```

The value must be a ClassAd expression which evaluates to a string. Thus, double quotes were required to make strings for both `SUBMIT_REQUIREMENT_NotStandardUniverse_REASON` and `SUBMIT_REQUIREMENT_NotChris_REASON`. The ClassAd function `strcat()` produces a string in the definition of `SUBMIT_REQUIREMENT_MinimalRequestMemory_REASON`.

Rejection reasons are sent back to the submitting program and will typically be immediately presented to the user. If an optional `SUBMIT_REQUIREMENT_<Name>_REASON` is not defined, a default reason will include the `<Name>` chosen for the submit requirement. Completing the presentation of the example submit requirements, upon an attempt to submit a standard universe job, *condor_submit* would print

```
Submitting job(s).
ERROR: Failed to commit job submission into the queue.
ERROR: This pool does not accept standard universe jobs.
```

Where there are multiple jobs in a cluster, if any job within the cluster is rejected due to a submit requirement, the entire cluster of jobs will be rejected.

3.8 Security

Security in HTCondor is a broad issue, with many aspects to consider. Because HTCondor'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 an HTCondor 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 HTCondor can defeat, and the kinds of attacks HTCondor cannot defeat. HTCondor cannot prevent security breaches of users that can elevate their privilege to the root or administrator account. HTCondor does not run user jobs in sandboxes (standard universe jobs are a partial exception to this), so HTCondor cannot defeat all malicious actions by user jobs. An example of a malicious job is one that launches a distributed denial of service attack. HTCondor assumes that users are trustworthy. HTCondor can prevent unauthorized access to the HTCondor pool, to help ensure that only trusted users have access to the pool. In addition, HTCondor provides encryption and integrity checking, to ensure that network transmissions are not examined or tampered with while in transit.

Broadly speaking, the aspects of security in HTCondor 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 HTCondor daemons become process owners. The HTCondor system prefers that HTCondor daemons are run as the user `root`, while other common operations are owned by a user of HTCondor. Operations that do not belong to either `root` or an HTCondor user are often owned by the `condor` user. See Section 3.8.13 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, HTCondor's authentication uses the user id (UID) to determine identity, but HTCondor 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 HTCondor itself. HTCondor provides authorization on either a per-user or on a per-machine basis.

Privacy HTCondor may encrypt data sent across the network, which prevents others from viewing the data. With persistence and sufficient computing power, decryption is possible. HTCondor 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 by default. However, see the `ENCRYPT_EXECUTE_DIRECTORY` setting for how to encrypt job data on the disk of an execute node.

Integrity The *man-in-the-middle* attack tampers with data without the awareness of either side of the communication. HTCondor'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. Also note that integrity checks are not performed upon job data files that are transferred by HTCondor via the File Transfer Mechanism described in section 2.5.9.

3.8.1 HTCondor's Security Model

At the heart of HTCondor's security model is the notion that communications are subject to various security checks. A request from one HTCondor daemon to another may require authentication to prevent subversion of the system. A request from a user of HTCondor 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 HTCondor 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 3.8.7 for a full list of HTCondor's access levels and their meanings.

There are two sides to any communication or command invocation in HTCondor. 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. HTCondor 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, HTCondor 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.

HTCondor's security model is implemented using configuration. Commands in HTCondor are executed over TCP/IP network connections. While network communication enables HTCondor to manage resources that are distributed across an organization (or beyond), it also brings in security challenges. HTCondor must have ways of ensuring that communications are being sent by trustworthy users and not tampered with in transit. These issues can be addressed with HTCondor'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 HTCondor. 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 HTCondor. 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 HTCondor administrator rights to the pool. It includes the ability to change user priorities with the command *condor_userprio*, as well as the ability to turn HTCondor on and off (as with the commands *condor_on* and *condor_off*). The *condor_fetchlog* tool also requires an ADMINISTRATOR access level. 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 HTCondor. 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 an HTCondor 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 HTCondor administrators. An example that requires the OWNER access level is the *condor_vacate* command. The command causes the *condor_startd* daemon to vacate any HTCondor 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 HTCondor. 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 HTCondor 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.

ADVERTISE_MASTER 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.

ADVERTISE_STARTD 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.

ADVERTISE_SCHEDD 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 an HTCondor daemon or tool is connecting *to* some other HTCondor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

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.

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 *condor_userprio* command.

COLLECTOR:

ADVERTISE_MASTER Commands that update the *condor_collector* daemon with new *condor_master* ClassAds.

ADVERTISE_SCHEDD Commands that update the *condor_collector* daemon with new *condor_schedd* ClassAds.

ADVERTISE_STARTD 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 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, HTCondor 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.

The commands that a *condor_startd* sends to the *condor_schedd* when the *condor_schedd* daemon's claim is being preempted and also when the lease on the claim is renewed. These operations only require `READ` access, rather than `DAEMON` in order to limit the level of trust that the *condor_schedd* must have for the *condor_startd*. Success of these commands is only possible if the *condor_startd* knows the secret claim id, so effectively, authorization for these commands is more specific than HTCondor's general security model implies. The *condor_schedd automatically* grants the *condor_startd* `READ` access for the duration of the claim. Therefore, if one desires to only authorize specific execute machines to run jobs, one must either limit which machines are allowed to advertise themselves to the pool (most common) or configure the *condor_schedd*'s `ALLOW_CLIENT` setting to only allow connections from the *condor_schedd* to the trusted execute machines.

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

3.8.2 Security Negotiation

Because of the wide range of environments and security demands necessary, HTCondor must be flexible. Configuration provides this flexibility. The process by which HTCondor 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 HTCondor 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 HTCondor daemon or tool initiates communication with another HTCondor 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.

As an example, consider Frida the scientist. Frida wants to avoid authentication when possible. She sets

```
SEC_DEFAULT_AUTHENTICATION = OPTIONAL
```

		Daemon Setting		
		NEVER	OPTIONAL	REQUIRED
Client Setting	NEVER	No	No	Fail
	REQUIRED	Fail	Yes	Yes

Table 3.1: Resolution of security negotiation.

		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.

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, HTCondor'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 HTCondor 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.

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 3.8.3 lists all implemented authentication methods. Section 3.8.5 lists all implemented encryption methods.

3.8.3 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. As not all of the authentication methods work on Windows platforms, which ones do *not* work on Windows are indicated in the following list of defined values:

```
GSI       (not available on Windows platforms)
SSL
KERBEROS
PASSWORD
FS        (not available on Windows platforms)
FS_REMOTE (not available on Windows platforms)
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 321 for details.

If the configuration for a machine does not define any variable for SEC_<access-level>_AUTHENTICATION, then HTCondor 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 HTCondor 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 HTCondor's use of terminology, and it illuminates the needed items that HTCondor 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 HTCondor 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 an HTCondor 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 HTCondor (the process owner of an HTCondor 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 HTCondor daemons For an HTCondor daemon, the certificate may be a single host certificate, and all HTCondor 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. HTCondor 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,...
```

HTCondor will also need a way to map an X.509 distinguished name to an HTCondor user id. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.8.4 to use HTCondor'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 X.509 Distinguished Name (DN) to HTCondor 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. 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 double quote marks (using the character "). The second item is the HTCondor 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
```

HTCondor 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, HTCondor looks for the map file in

```
$(GSI_DAEMON_DIRECTORY)/grid-mapfile
```

If GSI_DAEMON_DIRECTORY is not defined, then the third place HTCondor 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, HTCondor 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 HTCondor 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 HTCondor. 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 HTCondor. 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 an HTCondor UID domain is necessary. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.8.4 to use HTCondor'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 HTCondor assumes that the Kerberos realm is the same as the HTCondor 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.

HTCondor 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 HTCondor'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 HTCondor 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 HTCondor 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 7.2.3).

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 HTCondor 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 HTCondor 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 HTCondor'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 HTCondor 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 HTCondor 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 HTCondor and in the list of authentication methods for testing purposes only.

3.8.4 The Unified Map File for Authentication

HTCondor's unified map file allows the mappings from authenticated names to an HTCondor 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 name or a regular expression representing the authenticated name to be mapped.
3. The canonical HTCondor 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 HTCondor 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.

For HTCondor version 8.5.8 and later, the authenticated name field will be interpreted as a regular expression or as a simple string based on the value of the CERTIFICATE_MAPFILE_ASSUME_HASH_KEYS configuration variable. If this configuration varible is true, then the authenticated name field is a regular expression only when it begins and ends with the / character. If this configuration variable is false, or on HTCondor versions older than 8.5.8, the authenticated name field is always a regular expression.

A regular expression may need to contain spaces, and in this case the entire expression can be surrounded by double quote marks. If a double quote character also needs to appear in such an expression, it is preceded by a backslash.

The default behavior of HTCondor when no map file 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@unmapped
KERBEROS ([^/]*)/?[^@]*@(.*) \1@\2
NTSSPI (.*) \1
CLAIMTOBE (.*) \1
PASSWORD (.*) \1
```

For GSI (or SSL), the special name GSS_ASSIST_GRIDMAP instructs HTCondor to use the GSI grid map file (configured with GRIDMAP as shown in section 3.8.3) 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@unmapped.

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 3.8.3 for details.

If authentication did not happen or failed and was not required, then the user is given the name unauthenticated@unmapped.

With the integration of VOMS for GSI authentication, the interpretation of the regular expression representing the authenticated name may change. First, the full serialized DN and FQAN are used in attempting a match. If no match is found using the full DN and FQAN, then the DN is then used on its own without the FQAN. Using this, roles or user names from the VOMS attributes may be extracted to be used as the target for mapping. And, in this case the FQAN are verified, permitting reliance on their authenticity.

3.8.5 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_SCHEDD_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_CRYPTO_METHODS
SEC_CLIENT_CRYPTO_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_READ_CRYPTO_METHODS
SEC_WRITE_CRYPTO_METHODS
```

```
SEC_ADMINISTRATOR_CRYPTO_METHODS
SEC_CONFIG_CRYPTO_METHODS
SEC_OWNER_CRYPTO_METHODS
SEC_DAEMON_CRYPTO_METHODS
SEC_NEGOTIATOR_CRYPTO_METHODS
SEC_ADVERTISE_MASTER_CRYPTO_METHODS
SEC_ADVERTISE_STARTD_CRYPTO_METHODS
SEC_ADVERTISE_SCHEDD_CRYPTO_METHODS
```

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
```

3.8.6 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.

Note at this time, integrity checks are not performed upon job data files that are transferred by HTCondor via the File Transfer Mechanism described in section 2.5.9.

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.

3.8.7 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 3.8.9 on Setting Up IP/Host-Based Security in HTCondor 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 an HTCondor 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 HTCondor 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 HTCondor 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
```

The names ALLOW_CLIENT and DENY_CLIENT should be thought of as "when I am acting as a client, these are the servers I allow or deny." It should *not* be confused with the incorrect thought "when I am the server, these are the clients I allow or deny."

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.

For Unix platforms where netgroups are implemented, a netgroup may specify a set of fully qualified users by using an extension to the syntax for all configuration variables of the form `ALLOW_*` and `DENY_*`. The syntax is the plus sign character (+) followed by the netgroup name. Permissions are applied to all members of the netgroup.

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.

In addition, there are some hard-coded authorization rules that cannot be modified by configuration.

1. Connections with a name matching `*@unmapped` are not allowed to do any job management commands (e.g. submitting, removing, or modifying jobs). This prevents these operations from being done by unauthenticated users and users who are authenticated but lacking a name in the map file.
2. To simplify flocking, the *condor_schedd* automatically grants the *condor_startd* `READ` access for the duration of a claim so that claim-related communications are possible. The *condor_shadow* grants the *condor_starter* `DAEMON` access so that file transfers can be done. The identity that is granted access in both these cases is the authenticated name (if available) and IP address of the *condor_startd* when the *condor_schedd* initially connects to it to request the claim. It is important that only trusted *condor_startd*s are allowed to publish themselves to the collector or that the *condor_schedd*'s `ALLOW_CLIENT` setting prevent it from allowing connections to *condor_startd*s that it does not trust to run jobs.
3. When `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION` is true, `execute-side@matchsession` is automatically granted `READ` access to the *condor_schedd* and `DAEMON` access to the *condor_shadow*.

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 3.8.1.

```
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 HTCondor 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 HTCondor 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 HTCondor mentions in the explanations of why requests are granted or denied.

3.8.8 Security Sessions

To set up and configure secure communications in HTCondor, 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 HTCondor 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, HTCondor would not be able to scale well. Therefore, HTCondor 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 HTCondor 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 HTCondor. 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 HTCondor requires sessions when you have security turned on. On the other hand, if you are not using strong security in HTCondor, 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 HTCondor. HTCondor 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.

3.8.9 Host-Based Security in HTCondor

This section describes the mechanisms for setting up HTCondor'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 an HTCondor pool, which machines can find out information about your pool, and which machines within a pool can perform administrative commands. By default, HTCondor 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 HTCondor. It lists the different levels of access and what parts of HTCondor 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 HTCondor daemons or tools that use DaemonCore (see section 3.11 for details), most tasks are accomplished by sending commands to another HTCondor 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, HTCondor 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 HTCondor can be registered with:

READ Machines with `READ` access can read information from the HTCondor 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 an HTCondor pool; the machine can only view information about the pool.

WRITE Machines with `WRITE` access can write information to the HTCondor 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 an HTCondor pool, the machine must have both `WRITE` permission **AND** `READ` permission. `WRITE` permission is not enough.

ADMINISTRATOR Machines with ADMINISTRATOR access are granted additional HTCondor administrator rights to the pool. This includes the ability to change user priorities with the command *condor_userprio*, and the ability to turn HTCondor on and off using *condor_on* and *condor_off*. It is recommended that few machines be granted administrator access in a pool; typically these are the machines that are used by HTCondor 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 HTCondor 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 HTCondor administrators. For example, the *condor_vacate* command causes the *condor_startd* daemon to vacate any running HTCondor 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 HTCondor. 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 HTCondor 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.

ADVERTISE_MASTER 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.

ADVERTISE_STARTD 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.

ADVERTISE_SCHEDD 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 an HTCondor daemon or tool is connecting *to* some other HTCondor daemon. In other words, it specifies the policy of the client that is initiating the operation, rather than the server that is being contacted.

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.

3.8.10 Examples of Security Configuration

Here is a sample security configuration:

```
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
ALLOW_READ = *
ALLOW_WRITE = *
ALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
ALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
ALLOW_WRITE_COLLECTOR = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_WRITE_STARTD    = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_READ_COLLECTOR  = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_READ_STARTD     = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_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 there is an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.
- If there is a DENY, it means "deny these machines". No DENY means deny nobody.
- If there is 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 3.3.12 on Subsystem Names), which is one of: STARTD, SCHEDD, MASTER, NEGOTIATOR, or COLLECTOR. For example, to grant different read access for the *condor_schedd*:

```
ALLOW_READ_SCHEDD = <list of machines>
```

Here are more examples of configuration settings. Notice that ADMINISTRATOR access is only granted through an ALLOW setting to explicitly grant access to a small number of machines. We recommend this.

- Let any machine join the pool. Only the central manager has administrative access.

```
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA to join or view the pool. The central manager is the only machine with ADMINISTRATOR access.

```
ALLOW_READ = *.ncsa.uiuc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST)
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_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).

```
ALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
DENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
ALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_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.cs.wisc.edu has the read access it needs through the wild card setting in ALLOW_READ). This example also shows how to use the continuation character, \, 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.

```
ALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_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.

```
DENY_READ = *.mil
ALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
ALLOW_WRITE = *.ncsa.uiuc.edu
ALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
ALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
ALLOW_OWNER = $(FULL_HOSTNAME), $(ALLOW_ADMINISTRATOR)
```

3.8.11 Changing the Security Configuration

A new security feature introduced in HTCondor 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 760 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 HTCondor system and the security of the machines in an HTCondor pool, it is important to restrict the ability to change attributes remotely.

For each security access level described, the HTCondor 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 HTCondor daemon, or the administrator can define one list that is used by all daemons.

For each command that requests a change in configuration setting, HTCondor 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, HTCondor 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 HTCondor is exceedingly restrictive. HTCondor 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 HTCondor users to set attributes in the configuration file which are not used by HTCondor directly. These are custom attributes published by various HTCondor daemons with the `<SUBSYS>_ATTRS` setting described in section 3.5.3 on page 226. It is secure to grant access only to modify attributes that are used by HTCondor to publish information. Granting access to modify settings used to control the behavior of HTCondor is not secure. The goal is to ensure no one can use the power to change configuration attributes to compromise the security of your HTCondor 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 the `<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 HTCondor daemon (for example, STARTD, SCHEDD, and MASTER). There are many configuration settings that can be defined differently for each daemon that use this <SUBSYS> naming convention. See section 3.3.12 on page 199 for a list. If there is no daemon-specific value for a given daemon, HTCondor 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 HTCondor 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 HTCondor 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 an HTCondor job, to modify the HasDataSet variable for that host. HasDataSet is not used by HTCondor, it is an invented attribute included in the STARTD_ATTRS setting in order for this example to make sense.

3.8.12 Using HTCondor w/ Firewalls, Private Networks, and NATs

This topic is now addressed in more detail in section 3.9, which explains network communication in HTCondor.

3.8.13 User Accounts in HTCondor on Unix Platforms

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.

HTCondor 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 HTCondor 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 variable. In either case, the value should be the UID integer, followed by a period, followed by the GID integer. For example, if an HTCondor administrator does not want to create a `condor` user, and instead wants their HTCondor 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 HTCondor 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 HTCondor.

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 HTCondor as Non-Root

While we strongly recommend starting up the HTCondor daemons as `root`, we understand that it is not always possible to do so. The main problems of not running HTCondor daemons as `root` appear when one HTCondor installation is shared by many users on a single machine, or if machines are set up to only execute HTCondor jobs. With a submit-only installation for a single user, there is no need for or benefit from running as `root`.

The effects of HTCondor of running both with and without root access are classified for each daemon:

condor_startd An HTCondor machine set up to execute jobs where the *condor_startd* is not started as `root` relies on the good will of the HTCondor users to agree to the policy configured for the *condor_startd* to enforce for starting, suspending, vacating, and killing HTCondor jobs. When the *condor_startd* is started as `root`, however, these policies may be enforced regardless of malicious users. By running as `root`, the HTCondor daemons run with a different UID than the HTCondor job. 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 HTCondor job cannot do anything to the HTCondor daemons. Without starting the daemons as `root`, all processes started by HTCondor, including the user's job, run with the same UID. Only `root` can switch UIDs. Therefore, a user's job could kill the *condor_startd* and *condor_starter*. By doing so, the user's job avoids getting suspended or vacated. This is nice for the job, as it obtains unlimited access to the machine, but it is awful for the machine owner or administrator. If there is trust of the users submitting jobs to HTCondor, this might not be a concern. However, to ensure that

the policy chosen is enforced by HTCondor, the *condor_startd* should be started as `root`.

In addition, some system information cannot be obtained without `root` access on some platforms. As a result, when running without `root` access, the *condor_startd* must call other programs such as *uptime*, to get this information. This is much less efficient than getting the information directly from the kernel, as is done when running as `root`. On Linux, this information is available without root access, so it is not a concern on those platforms.

If all of HTCondor cannot be run as `root`, at least consider installing the *condor_startd* as setuid root. That would solve both problems. Barring that, install it as a setgid sys or kmem program, depending on whatever group has read access to `/dev/kmem` on the system. That would solve the system information problem.

condor_schedd The biggest problem with running the *condor_schedd* without `root` access is that the *condor_shadow* processes which it spawns are stuck with the same UID that the *condor_schedd* has. This requires users to go out of their way to grant write access to user or group that the *condor_schedd* is run 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 job event log 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* spawns both the *condor_startd* and the *condor_schedd*. To have both running as `root`, have the *condor_master* run as `root`. This happens automatically if the *condor_master* is started from 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*, the *condor_kbdd* must run as `root`. If it is started as any other user, it will not work. Consider installing this program as a setuid root binary if the *condor_master* will not be run as `root`. Without the *condor_kbdd*, the *condor_startd* has no way to monitor USB mouse or keyboard activity, although it will notice keyboard activity on ttys such as xterms and remote logins.

If HTCondor is not run as root, then choose almost any user name. A common choice is to set up and use the `condor` user; this simplifies the setup, because HTCondor will look for its configuration files in the `condor` user's directory. If `condor` is not selected, then the configuration must be placed properly such that HTCondor can find its configuration files.

If users will be submitting jobs as a user different than the user HTCondor 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 HTCondor is using. In practice, this means creating world-writable directories for output from HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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 HTCondor submit installation.

When a machine is an execution site for an HTCondor job, the HTCondor 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 `condor` that exists only to run HTCondor jobs.

Who Jobs Run As

Under Unix, HTCondor runs jobs as one of

- the user called `nobody`

  Running jobs as the `nobody` user is the least preferable. HTCondor uses user `nobody` if the value of the `UID_DOMAIN` configuration variable of the submitting and executing machines are different, or if configuration variable `STARTER_ALLOW_RUNAS_OWNER` is `False`, or if the job ClassAd contains `RunAsOwner=False`.

  When HTCondor 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 HTCondor 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 HTCondor, and leave processes behind after HTCondor 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.

  HTCondor 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-HTCondor system processes. It may also be used by other HTCondor jobs running on the same machine, if it is a multi-processor machine.

- dedicated accounts called slot users set up for the purpose of running HTCondor jobs

  Better than the `nobody` user will be to create user accounts for HTCondor to use. These can be low-privilege accounts, just as the `nobody` user is. Create one of these accounts for each job execution slot per computer, so that distinct user names can be used for concurrently running jobs. This prevents malicious or naive behavior from one slot to affect another slot. For a sample machine with two compute slots, create two users that are intended only to be used by HTCondor. As an example, call them `cndrusr1` and `cndrusr2`. Configuration identifies these users with the `SLOT<N>_USER` configuration variable, where `<N>` is replaced with the slot number. Here is configuration for this example:

  ```
  SLOT1_USER = cndrusr1
  SLOT2_USER = cndrusr2
  ```

  Also tell HTCondor that these accounts are intended only to be used by HTCondor, so HTCondor can kill all the processes belonging to these users upon job completion. The configuration variable

DEDICATED_EXECUTE_ACCOUNT_REGEXP is introduced and set to a regular expression that matches the account names just created:

```
DEDICATED_EXECUTE_ACCOUNT_REGEXP = cndrusr[0-9]+
```

Finally, tell HTCondor not to run jobs as the job owner:

```
STARTER_ALLOW_RUNAS_OWNER = False
```

- the user that submitted the jobs

  Four conditions must be set correctly to run jobs as the user that submitted the job.

  1. In the configuration, the value of variable STARTER_ALLOW_RUNAS_OWNER must be True on the machine that will run the job. Its default value is True on Unix platforms and False on Windows platforms.
  2. The job's ClassAd must have attribute RunAsOwner set to True. This can be set up for all users by adding an attribute to configuration variable SUBMIT_ATTRS. If this were the only attribute to be added to all job ClassAds, it would be set up with

     ```
     SUBMIT_ATTRS = RunAsOwner
     RunAsOwner = True
     ```

  3. The value of configuration variable UID_DOMAIN must be the same for both the *condor_startd* and *condor_schedd* daemons.
  4. The UID_DOMAIN must be trusted. For example, if the *condor_starter* daemon does a reverse DNS lookup on the *condor_schedd* daemon, and finds that the result is *not* the same as defined for configuration variable UID_DOMAIN, then it is not trusted. To correct this, set in the configuration for the *condor_starter*

     ```
     TRUST_UID_DOMAIN = True
     ```

Notes:

1. Currently, none of these configuration settings apply to standard universe jobs. Normally, standard universe jobs do not create additional processes.
2. Under Windows, HTCondor 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.

   SLOT<N>_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. 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. See section 7.2.4, 7.2.5, and the *condor_store_cred* manual page at section 11 for details.
3. The *condor_starter* logs a line similar to

   ```
   Tracking process family by login "cndrusr1"
   ```

   when it treats the account as a dedicated account.

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 HTCondor 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 HTCondor.

There is also an execute-side current working directory. For standard universe jobs, it is set to the `execute` subdirectory of HTCondor's home directory. This directory is world-writable, since an HTCondor 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.

3.9 Networking (includes sections on Port Usage and CCB)

This section on network communication in HTCondor discusses which network ports are used, how HTCondor 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 3.8 as well.

Firewalls, private networks, and network address translation (NAT) pose special problems for HTCondor. There are currently two main mechanisms for dealing with firewalls within HTCondor:

1. Restrict HTCondor to use a specific range of port numbers, and allow connections through the firewall that use any port within the range.
2. Use *HTCondor Connection Brokering* (CCB).

Each method has its own advantages and disadvantages, as described below.

3.9.1 Port Usage in HTCondor

IPv4 Port Specification

The general form for IPv4 port specification is

```
<IP:port?param1name=value1&param2name=value2&param3name=value3&...>
```

These parameters and values are URL-encoded. This means any special character is encoded with %, followed by two hexadecimal digits specifying the ASCII value. Special characters are any non-alphanumeric character.

HTCondor currently recognizes the following parameters with an IPv4 port specification:

CCBID Provides contact information for forming a CCB connection to a daemon, or a space separated list, if the daemon is registered with more than one CCB server. Each contact information is specified in the form of `IP:port#ID`. Note that spaces between list items will be URL encoded by `%20`.

PrivNet Provides the name of the daemon's private network. This value is specified in the configuration with `PRIVATE_NETWORK_NAME`.

sock Provides the name of *condor_shared_port* daemon named socket.

PrivAddr Provides the daemon's private address in form of `IP:port`.

Default Port Usage

Every HTCondor daemon listens on a network port for incoming commands. (Using *condor_shared_port*, this port may be shared between multiple daemons.) Most daemons listen on a dynamically assigned port. In order to send a message, HTCondor 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 HTCondor 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, HTCondor 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 HTCondor daemon can be configured to write its IP address and port number into a well-known file. The file names are controlled using the `<SUBSYS>_ADDRESS_FILE` configuration variables, as described in section 3.5.3 on page 225.

NOTE: In the 6.6 stable series, and HTCondor 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 HTCondor daemons, and publishes its own ClassAd to the *condor_collector* which includes the dynamically assigned port the *condor_negotiator* is listening on. All HTCondor 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, HTCondor uses port 9618 for the *condor_collector* daemon. To use a different port number for this daemon, the configuration variables that tell HTCondor 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 HTCondor 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 HTCondor 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 HTCondor pools, and only if the COLLECTOR_ADDRESS_FILE configuration variable has also been defined. This mechanism allows all of the HTCondor 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 HTCondor clients know to look at the information stored in this file. For example:

```
COLLECTOR_HOST = $(CONDOR_HOST):0
COLLECTOR_ADDRESS_FILE = $(LOG)/.collector_address
```

Configuration definition of COLLECTOR_ADDRESS_FILE is in section 3.5.3 on page 225, and COLLECTOR_HOST is in section 3.5.1 on page 207.

Restricting Port Usage to Operate with Firewalls

If an HTCondor 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 an HTCondor pool, then configuration variables may be set to force the usage of specific ports, and to utilize a specific range of ports.

By default, HTCondor uses port 9618 for the *condor_collector* daemon, and dynamic (apparently random) ports for everything else. See section 3.9.1, if a dynamically assigned port is desired for the *condor_collector* daemon.

All of the HTCondor daemons on a machine may be configured to share a single port. See section 3.5.31 for more information.

The configuration variables HIGHPORT and LOWPORT facilitate setting a restricted range of ports that HTCondor 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 3.5.4. 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 HTCondor 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 HTCondor 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 3.5.4 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. This assumes that *condor_shared_port* is not being used. If it *is* being used, then all daemons can share a single incoming port.

The central manager of the pool needs `5 + 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 * 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 * 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 HTCondor system, and they do not impact port selection used by jobs submitted to HTCondor. Thus, jobs submitted to HTCondor 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 MPI applications to be executed under the parallel 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

3.9.2 Reducing Port Usage with the *condor_shared_port* Daemon

The *condor_shared_port* is an optional daemon responsible for creating a TCP listener port shared by all of the HTCondor daemons.

The main purpose of the *condor_shared_port* daemon is to reduce the number of ports that must be opened. This is desirable when HTCondor daemons need to be accessible through a firewall. This has a greater security benefit than simply reducing the number of open ports. Without the *condor_shared_port* daemon, HTCondor can use a range of ports, but since some HTCondor daemons are created dynamically, this full range of ports will not be in use by HTCondor at all times. This implies that other non-HTCondor processes not intended to be exposed to the outside network could unintentionally bind to ports in the range intended for HTCondor, unless additional steps are taken to

control access to those ports. While the *condor_shared_port* daemon is running, it is exclusively bound to its port, which means that other non-HTCondor processes cannot accidentally bind to that port.

A second benefit of the *condor_shared_port* daemon is that it helps address the scalability issues of a submit machine. Without the *condor_shared_port* daemon, more than 2 ephemeral ports per running job are often required, depending on the rate of job completion. There are only 64K ports in total, and most standard Unix installations only allocate a subset of these as ephemeral ports. Therefore, with long running jobs, and with between 11K and 14K simultaneously running jobs, port exhaustion has been observed in typical Linux installations. After increasing the ephemeral port range to its maximum, port exhaustion occurred between 20K and 25K running jobs. Using the *condor_shared_port* daemon dramatically reduces the required number of ephemeral ports on the submit node where the submit node connects directly to the execute node. If the submit node connects via CCB to the execute node, *no* ports are required per running job; only the one port allocated to the *condor_shared_port* daemon is used.

When CCB is enabled, the *condor_shared_port* daemon registers with the CCB server on behalf of all daemons sharing the port. This means that it is not possible to individually enable or disable CCB connectivity to daemons that are using the shared port; they all effectively share the same setting, and the *condor_shared_port* daemon handles all CCB connection requests on their behalf.

HTCondor's authentication and authorization steps are unchanged by the use of a shared port. Each HTCondor daemon continues to operate according to its configured policy. Requests for connections to the shared port are not authenticated or restricted by the *condor_shared_port* daemon. They are simply passed to the requested daemon, which is then responsible for enforcing the security policy.

When the *condor_master* is configured to use the shared port by setting the configuration variable

```
USE_SHARED_PORT = True
```

the *condor_shared_port* daemon is treated specially. `SHARED_PORT` is automatically added to `DAEMON_LIST`. A command such as *condor_off*, which shuts down all daemons except for the *condor_master*, will also leave the *condor_shared_port* running. This prevents the *condor_master* from getting into a state where it can no longer receive commands.

Also when `USE_SHARED_PORT = True`, the *condor_collector* needs to be configured to use a shared port, so that connections to the shared port that are destined for the *condor_collector* can be forwarded. As an example, the shared port socket name of the *condor_collector* with shared port number 11000 is

```
COLLECTOR_HOST = cm.host.name:11000?sock=collector
```

This example assumes that the socket name used by the *condor_collector* is `collector`, and it runs on `cm.host.name`. This configuration causes the *condor_collector* to automatically choose this socket name. If multiple *condor_collector* daemons are started on the same machine, the socket name can be explicitly set in the daemon's invocation arguments, as in the example:

```
COLLECTOR_ARGS = -sock collector
```

When the *condor_collector* address is a shared port, TCP updates will be automatically used instead of UDP, because the *condor_shared_port* daemon does not work with UDP messages. Under Unix, this means that the *condor_collector*

daemon should be configured to have enough file descriptors. See section 3.9.5 for more information on using TCP within HTCondor.

SOAP commands cannot be sent through the *condor_shared_port* daemon. However, a daemon may be configured to open a fixed, non-shared port, in addition to using a shared port. This is done both by setting `USE_SHARED_PORT = True` and by specifying a fixed port for the daemon using `<SUBSYS>_ARGS = -p <portnum>`.

The TCP connections required to manage standard universe jobs do not make use of shared ports. Therefore, if the firewall is configured to only allow connections through the shared port, standard universe jobs will not be able to run.

3.9.3 Configuring HTCondor for Machines With Multiple Network Interfaces

HTCondor can run on machines with multiple network interfaces. Starting with HTCondor version 6.7.13 (and therefore all HTCondor 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 HTCondor 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 HTCondor 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 HTCondor'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 an HTCondor 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 HTCondor tools directly. In this case, remote clients do not need to use HTCondor tools running on machines in the public network to access the HTCondor daemons on the multi-homed machine. Therefore, there is no reason to have HTCondor daemons listening on ports on the public Internet, causing a potential security threat.

Up to two IP addresses will be advertised. At present, even though a given HTCondor 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, HTCondor clients (other HTCondor daemons or tools) will not necessarily able to locate and communicate with a given daemon running on a multi-homed machine where `BIND_ALL_INTERFACES` has been enabled.

Currently, HTCondor daemons can only advertise two IP addresses in the ClassAd they send to their *condor_collector*. One is the public IP address and the other is the private IP address. HTCondor tools and other daemons that wish to connect to the daemon will use the private IP address if they are configured with the same private network name, and they will use the public IP address otherwise. So, even if the daemon is listening on 3 or more different interfaces, each with a separate IP, the daemon must choose which two IP addresses to advertise so that other daemons and tools can connect to it.

By default, HTCondor advertises the IP address of the network interface used to contact the *condor_collector* as its public address, since this is the most likely to be accessible to other processes that query the same *condor_collector*. The `NETWORK_INTERFACE` configuration variable can be used to specify the public IP address HTCondor should advertise, and `PRIVATE_NETWORK_INTERFACE`, along with `PRIVATE_NETWORK_NAME` can be used to specify the private IP address to advertise.

Sites that make heavy use of private networks and multi-homed machines should consider if using the HTCondor Connection Broker, CCB, is right for them. More information about CCB and HTCondor can be found in section 3.9.4 on page 440.

Central Manager with Two or More NICs

Often users of HTCondor 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 HTCondor:

In this example, `farm-server.farm.org` maps to the private interface. In the central manager's global (to the cluster) configuration file:

```
CONDOR_HOST = farm-server.farm.org
```

In the 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 the central manager and farm machines are all NT, then only vanilla universe will work now. However, if this is set up for Unix, then at this point, standard universe jobs should be able to function in the pool. But, if `UID_DOMAIN` is not configured 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 HTCondor 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, and it does not really matter to HTCondor. 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. Inform HTCondor that an NFS or AFS file system 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 the cluster is set up 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`), configure `DEFAULT_DOMAIN_NAME` to be the domain that is to be added on to the end of the host name.

A Client Machine with Multiple Interfaces

If client machine has two or more NICs, then there might be a specific network interface on which the client machine desires to communicate with the rest of the HTCondor pool. In this case, the local configuration file for the client should have

```
NETWORK_INTERFACE = <IP address of desired interface>
```

A Checkpoint Server on a Machine with Multiple NICs

If a checkpoint server is on a machine with multiple interfaces, then 2 items must be correct to get things to work:

1. The different interfaces have different host names associated with them.
2. In the global configuration file, set configuration variable `CKPT_SERVER_HOST` to the host name that corresponds with the IP address desired for the pool. Configuration variable `NETWORK_INTERFACE` must still be specified in the local configuration file for the checkpoint server.

3.9.4 HTCondor Connection Brokering (CCB)

HTCondor Connection Brokering, or CCB, is a way of allowing HTCondor components to communicate with each other when one side is in a private network or behind a firewall. Specifically, CCB allows communication across a private network boundary in the following scenario: an HTCondor tool or daemon (process A) needs to connect to an HTCondor daemon (process B), but the network does not allow a TCP connection to be created from A to B; it only allows connections from B to A. In this case, B may be configured to register itself with a CCB server that both A and B can connect to. Then when A needs to connect to B, it can send a request to the CCB server, which will instruct B to connect to A so that the two can communicate.

As an example, consider an HTCondor execute node that is within a private network. This execute node's *condor_startd* is process B. This execute node cannot normally run jobs submitted from a machine that is outside of that private network, because bi-directional connectivity between the submit node and the execute node is normally required. However, if both execute and submit machine can connect to the CCB server, if both are authorized by the CCB server, and if it is possible for the execute node within the private network to connect to the submit node, then it is possible for the submit node to run jobs on the execute node.

To effect this CCB solution, the execute node's *condor_startd* within the private network registers itself with the CCB server by setting the configuration variable `CCB_ADDRESS`. The submit node's *condor_schedd* communicates with the CCB server, requesting that the execute node's *condor_startd* open the TCP connection. The CCB server forwards this request to the execute node's *condor_startd*, which opens the TCP connection. Once the connection is open, bi-directional communication is enabled.

If the location of the execute and submit nodes is reversed with respect to the private network, the same idea applies: the submit node within the private network registers itself with a CCB server, such that when a job is running and the execute node needs to connect back to the submit node (for example, to transfer output files), the execute node can connect by going through CCB to request a connection.

If both A and B are in separate private networks, then CCB alone cannot provide connectivity. However, if an incoming port or port range can be opened in one of the private networks, then the situation becomes equivalent to one of the scenarios described above and CCB can provide bi-directional communication given only one-directional connectivity. See section 3.9.1 for information on opening port ranges. Also note that CCB works nicely with *condor_shared_port*.

Unfortunately at this time, CCB does not support standard universe jobs.

Any *condor_collector* may be used as a CCB server. There is no requirement that the *condor_collector* acting as the

CCB server be the same *condor_collector* that a daemon advertises itself to (as with COLLECTOR_HOST). However, this is often a convenient choice.

Example Configuration

This example assumes that there is a pool of machines in a private network that need to be made accessible from the outside, and that the *condor_collector* (and therefore CCB server) used by these machines is accessible from the outside. Accessibility might be achieved by a special firewall rule for the *condor_collector* port, or by being on a dual-homed machine in both networks.

The configuration of variable CCB_ADDRESS on machines in the private network causes registration with the CCB server as in the example:

```
CCB_ADDRESS = $(COLLECTOR_HOST)
PRIVATE_NETWORK_NAME = cs.wisc.edu
```

The definition of PRIVATE_NETWORK_NAME ensures that all communication between nodes within the private network continues to happen as normal, and without going through the CCB server. The name chosen for PRIVATE_NETWORK_NAME should be different from the private network name chosen for any HTCondor installations that will be communicating with this pool.

Under Unix, and with large HTCondor pools, it is also necessary to give the *condor_collector* acting as the CCB server a large enough limit of file descriptors. This may be accomplished with the configuration variable MAX_FILE_DESCRIPTORS or an equivalent. Each HTCondor process configured to use CCB with CCB_ADDRESS requires one persistent TCP connection to the CCB server. A typical execute node requires one connection for the *condor_master*, one for the *condor_startd*, and one for each running job, as represented by a *condor_starter*. A typical submit machine requires one connection for the *condor_master*, one for the *condor_schedd*, and one for each running job, as represented by a *condor_shadow*. If there will be no administrative commands required to be sent to the *condor_master* from outside of the private network, then CCB may be disabled in the *condor_master* by assigning MASTER.CCB_ADDRESS to nothing:

```
MASTER.CCB_ADDRESS =
```

Completing the count of TCP connections in this example: suppose the pool consists of 500 8-slot execute nodes and CCB is not disabled in the configuration of the *condor_master* processes. In this case, the count of needed file descriptors plus some extra for other transient connections to the collector is 500*(1+1+8)=5000. Be generous, and give it twice as many descriptors as needed by CCB alone:

```
COLLECTOR.MAX_FILE_DESCRIPTORS = 10000
```

Security and CCB

The CCB server authorizes all daemons that register themselves with it (using CCB_ADDRESS) at the DAEMON authorization level (these are playing the role of process A in the above description). It authorizes all connection

requests (from process B) at the READ authorization level. As usual, whether process B authorizes process A to do whatever it is trying to do is up to the security policy for process B; from the HTCondor security model's point of view, it is as if process A connected to process B, even though at the network layer, the reverse is true.

Troubleshooting CCB

Errors registering with CCB or requesting connections via CCB are logged at level `D_ALWAYS` in the debugging log. These errors may be identified by searching for "CCB" in the log message. Command-line tools require the argument **-debug** for this information to be visible. To see details of the CCB protocol add `D_FULLDEBUG` to the debugging options for the particular HTCondor subsystem of interest. Or, add `D_FULLDEBUG` to `ALL_DEBUG` to get extra debugging from all HTCondor components.

A daemon that has successfully registered itself with CCB will advertise this fact in its address in its ClassAd. The ClassAd attribute `MyAddress` will contain information about its `"CCBID"`.

Scalability and CCB

Any number of CCB servers may be used to serve a pool of HTCondor daemons. For example, half of the pool could use one CCB server and half could use another. Or for redundancy, all daemons could use both CCB servers and then CCB connection requests will load-balance across them. Typically, the limit of how many daemons may be registered with a single CCB server depends on the authentication method used by the *condor_collector* for DAEMON-level and READ-level access, and on the amount of memory available to the CCB server. We are not able to provide specific recommendations at this time, but to give a very rough idea, a server class machine should be able to handle CCB service plus normal *condor_collector* service for a pool containing a few thousand slots without much trouble.

3.9.5 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. The lack of guaranteed delivery of UDP will negatively affect some pools, particularly ones comprised of machines across a wide area network (WAN) or highly-congested network links, where UDP packets are frequently dropped.

By default, HTCondor daemons will use TCP to send updates to the *condor_collector*, with the exception of the *condor_collector* forwarding updates to any *condor_collector* daemons specified in `CONDOR_VIEW_HOST`, where UDP is used. These configuration variables control the protocol used:

UPDATE_COLLECTOR_WITH_TCP When set to `False`, the HTCondor daemons will use UDP to update the *condor_collector*, instead of the default TCP. Defaults to `True`.

UPDATE_VIEW_COLLECTOR_WITH_TCP When set to `True`, the HTCondor collector will use TCP to forward

updates to *condor_collector* daemons specified by CONDOR_VIEW_HOST, instead of the default UDP. Defaults to False.

TCP_UPDATE_COLLECTORS A list of *condor_collector* daemons which will be updated with TCP instead of UDP, when UPDATE_COLLECTOR_WITH_TCP or UPDATE_VIEW_COLLECTOR_WITH_TCP is set to False.

When there are sufficient file descriptors, the *condor_collector* leaves established TCP sockets open, facilitating better performance. Subsequent updates can reuse an already open socket.

Each HTCondor daemon that sends updates to the *condor_collector* will have 1 socket open to it. So, in a pool with N machines, each of them running a *condor_master*, *condor_schedd*, and *condor_startd*, the *condor_collector* would need at least 3*N file descriptors. If the *condor_collector* is also acting as a CCB server, it will require an additional file descriptor for each registered daemon. In the default configuration, the number of file descriptors available to the *condor_collector* is 10240. For very large pools, the number of descriptor can be modified with the configuration:

```
COLLECTOR_MAX_FILE_DESCRIPTORS = 40960
```

If there are insufficient file descriptors for all of the daemons sending updates to the *condor_collector*, a warning will be printed in the *condor_collector* log file. The string "file descriptor safety level exceeded" identifies this warning.

3.9.6 Running HTCondor on an IPv6 Network Stack

HTCondor supports using IPv4, IPv6, or both. By default, HTCondor will look at a machine's interfaces, and on that machine, enable each protocol for which it finds at least one interface with an address of that procol.

To require IPv4, you may set ENABLE_IPV4 to true; if the machine does not have an interface with an IPv4 address, HTCondor will not start. Likewise, to require IPv6, you may set ENABLE_IPV6 to true.

If you set ENABLE_IPV4 to false, HTCondor will not use IPv4, even if it is available; likewise for ENABLE_IPV6 and IPv6.

The default setting for ENABLE_IPV4 and ENABLE_IPV6 is auto, which uses the corresponding protocol if and only HTCondor finds an interface with an address of that protocol.

If both IPv4 and IPv6 networking are enabled, HTCondor runs in mixed mode. In mixed mode, HTCondor daemons have at least one IPv4 address and at least one IPv6 address. Other daemons and the command-line tools choose between these addresses based on which protocols are enabled for them; if both are, they will prefer the first address listed by that daemon.

A daemon may be listening on one, some, or all of its machine's addresses. (See NETWORK_INTERFACE.) Daemons may presently list at most two addresses, one IPv6 and one IPv4. Each address is the "most public" address of its protocol; by default, the IPv6 address is listed first. HTCondor selects the "most public" address heuristically.

Nonetheless, there are two cases in which HTCondor may not use an IPv6 address when one is available:

- When given a literal IP address, HTCondor will use that IP address.

- When looking up a host name using DNS, HTCondor will use the first address whose protocol is enabled for the tool or daemon doing the look up.

You may force HTCondor to prefer IPv4 in all three of these situations by setting the macro `PREFER_IPV4` to true; this is the default. With `PREFER_IPV4` set, HTCondor daemons will list their "most public" IPv4 address first; prefer the IPv4 address when choosing from another's daemon list; and prefer the IPv4 address when looking up a host name in DNS.

In practice, both an HTCondor pool's central manager and any submit machines within a mixed mode pool must have both IPv4 and IPv6 addresses for both IPv4-only and IPv6-only *condor_startd* daemons to function properly.

IPv6 and Host-Based Security

You may freely intermix IPv6 and IPv4 address literals. You may also specify IPv6 netmasks as a legal IPv6 address followed by a slash followed by the number of bits in the mask; or as the prefix of a legal IPv6 address followed by two colons followed by an asterisk. The latter is entirely equivalent to the former, except that it only allows you to (implicitly) specify mask bits in groups of sixteen. For example, `fe8f:1234::/60` and `fe8f:1234::*` specify the same network mask.

The HTCondor security subsystem resolves names in the ALLOW and DENY lists and uses all of the resulting IP addresses. Thus, to allow or deny IPv6 addresses, the names must have IPv6 DNS entries (AAAA records), or `NO_DNS` must be enabled.

IPv6 Address Literals

When you specify an IPv6 address and a port number simultaneously, you must separate the IPv6 address from the port number by placing square brackets around the address. For instance:

```
COLLECTOR_HOST = [2607:f388:1086:0:21e:68ff:fe0f:6462]:5332
```

If you do not (or may not) specify a port, do not use the square brackets. For instance:

```
NETWORK_INTERFACE = 1234:5678::90ab
```

IPv6 without DNS

When using the configuration variable `NO_DNS`, IPv6 addresses are turned into host names by taking the IPv6 address, changing colons to dashes, and appending `$(DEFAULT_DOMAIN_NAME)`. So,

```
2607:f388:1086:0:21b:24ff:fedf:b520
```

becomes

```
2607-f388-1086-0-21b-24ff-fedf-b520.example.com
```

assuming

```
DEFAULT_DOMAIN_NAME=example.com
```

3.10 The Checkpoint Server

A Checkpoint Server maintains a repository for checkpoint files. Within HTCondor, checkpoints may be produced only for standard universe jobs. 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 HTCondor pool.

If the 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. A checkpoint server configured to use the server's local disk means that the checkpoint file will travel only once over the network, between the execution machine and the checkpoint server. The pool may also obtain checkpointing network performance benefits by using multiple checkpoint servers, as discussed below.

Note that it is a good idea to pick very stable machines for the checkpoint servers. If individual checkpoint servers crash, the HTCondor system will continue to operate, although poorly. While the HTCondor 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 checkpoint 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 on line. This behavior is controlled with the `MAX_DISCARDED_RUN_TIME` configuration variable. This variable represents the maximum amount of CPU time the job is willing to discard, by starting a job over from its beginning if the checkpoint server is not responding to requests.

3.10.1 Preparing to Install a Checkpoint Server

The location of checkpoint files changes upon the installation of a checkpoint server. A configuration change will cause currently queued jobs with checkpoints to not be able to find their checkpoints. This results in the jobs with checkpoints remaining indefinitely queued, 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 the pool down before doing any maintenance on the checkpoint server. See section 3.2.5 for details on shutting down the pool.

A graduated installation of the checkpoint server may be accomplished by configuring submit machines as their queues empty.

3.10.2 Installing the Checkpoint Server Module

The files relevant to a checkpoint server are

```
sbin/condor_ckpt_server
etc/examples/condor_config.local.ckpt.server
```

`condor_ckpt_server` is the checkpoint server binary. `condor_condor_config.local.ckpt.server` is an example configuration for a checkpoint server. The settings embodied in this file must be customized with site-specific information.

There are three steps necessary towards running a checkpoint server:

1. Configure the checkpoint server.
2. Start the checkpoint server.
3. Configure the 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 a template for the needed configuration. Insert these into the local configuration file of the checkpoint server machine.

The value of `CKPT_SERVER_DIR` must be customized. This variable defines the location of checkpoint files. It is better if this location is within a very fast local file system, and preferably a RAID. The speed of this file system will have a direct impact on the speed at which checkpoint files can be retrieved from the remote machines.

The other optional variables are:

DAEMON_LIST Described in section 3.5.7. To have the checkpoint server managed by the *condor_master*, the `DAEMON_LIST` variable's value must list both `MASTER` and `CKPT_SERVER`. Also add `STARTD` to allow jobs to run on the checkpoint server machine. Similarly, add `SCHEDD` to permit the submission of jobs from the checkpoint server machine.

The remainder of these variables are the checkpoint server-specific versions of the HTCondor logging entries, as described in section 3.5.2 on page 218.

CKPT_SERVER_LOG The location of the checkpoint server log.

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 HTCondor 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 the pool. See section 3.8.9 on page 421 for full details about IP/host-based security in HTCondor.

Note that when the *condor_ckpt_server* starts up, it will immediately inspect any checkpoint files in the location described by the CKPT_SERVER_DIR variable, and determine if any of them are stale. Stale checkpoint files will be removed.

Configure the Pool to Use the Checkpoint Server After the checkpoint server is running, modify a few configuration variables to let the other machines in the pool know about the new server:

USE_CKPT_SERVER A boolean value that should be set to True to enable the use of the checkpoint server.

CKPT_SERVER_HOST Provides the full host name of the machine that is now running the checkpoint server.

It is most convenient to set these variables in the pool's global configuration file, so that they affect all submission machines. However, it is permitted to configure each submission machine separately (using local configuration files), for example if it is desired that not all submission machines begin using the checkpoint server at one time. If the variable USE_CKPT_SERVER is set to False, the submission machine will not use a checkpoint server.

Once these variables are in place, send the command *condor_reconfig* to all machines in the pool, so the changes take effect. This is described in section 3.2.6 on page 183.

3.10.3 Configuring the Pool to Use Multiple Checkpoint Servers

An HTCondor pool may use multiple checkpoint servers. The deployment of checkpoint servers across the network improves the performance of checkpoint production. In this case, HTCondor machines are configured to send checkpoints 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.
- Better performance implies that jobs spend less time dealing with checkpoints, and more time doing useful work, leading to jobs having a higher success rate before returning a machine to its owner, and workstation owners see HTCondor jobs leave their machines quicker.

With multiple checkpoint servers running in the pool, the following configuration changes are required to make them active.

Set USE_CKPT_SERVER to True (the default) on all submitting machines where HTCondor jobs should use a checkpoint server. Additionally, variable STARTER_CHOOSES_CKPT_SERVER should be set to True (the default) on these submitting machines. When True, this variable 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 3.5.6 on page 237 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. This identifies the full host name of the checkpoint server machine, and should be the host name of the nearest server to the machine. In the case of multiple checkpoint servers, set this in the local configuration file.

Third, send a *condor_reconfig* command to all machines in the pool, so that the changes take effect. This is described in section 3.2.6 on page 183.

After completing these three steps, the jobs in the pool will send their checkpoints to the nearest checkpoint server. On restart, a job will remember where its checkpoint was stored and retrieve 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 that if the configured checkpoint server is unavailable, the job will keep trying to contact that server. It will not use alternate checkpoint servers. This may change in future versions of HTCondor.

3.10.4 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 HTCondor to localize checkpoint read transfers, which occur when the job restarts from its last checkpoint on a new machine. To localize these transfers, it is desired to schedule the job on a machine which is near the checkpoint server on which the job's checkpoint is stored.

In terminology, all of the machines configured to use checkpoint server *A* are in *checkpoint server domain A*. To localize checkpoint transfers, jobs which run on machines in a given checkpoint server domain should continue running on machines in that domain, thereby 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, set these variables in the global configuration file. Note that this example assumes that `STARTD_ATTRS` is previously defined in the configuration. If not, then use the following configuration instead:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_ATTRS = CkptServer
```

With this configuration, 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, 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 yet written a checkpoint, 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.

To instead allow jobs to transfer to other checkpoint server domains when there are no available machines in the current checkpoint server domain, 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.

The checkpoint server domain `Requirements` or `Rank` expressions can be automatically appended to all standard universe jobs submitted in the pool using the configuration variables `APPEND_REQ_STANDARD` or `APPEND_RANK_STANDARD`. See section 3.5.12 on page 283 for more details.

3.11 DaemonCore

This section is a brief description of *DaemonCore*. DaemonCore is a library that is shared among most of the HTCondor daemons which provides common functionality. Currently, the following daemons use DaemonCore:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*
- *condor_gridmanager*
- *condor_credd*
- *condor_had*

- *condor_replication*
- *condor_transferer*
- *condor_job_router*
- *condor_lease_manager*
- *condor_rooster*
- *condor_shared_port*
- *condor_defrag*
- *condor_c-gahp*
- *condor_c-gahp_worker_thread*
- *condor_dagman*
- *condor_ft-gahp*
- *condor_rooster*
- *condor_shadow*
- *condor_shared_port*
- *condor_transferd*
- *condor_vm-gahp*
- *condor_vm-gahp-vmware*

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.

3.11.1 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 3.5 on page 206 for full details on what settings are in the configuration files and what they do.

3.11.2 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 HTCondor 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 HTCondor checks for its configuration file.

-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 HTCondor 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 HTCondor 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 HTCondor'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 3.3.3 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 HTCondor 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 a full path (one that begins with a slash character, `/`) is not specified, 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.

3.12 Monitoring

Information that the *condor_collector* collects can be used to monitor a pool. The *condor_status* command can be used to display snapshot of the current state of the pool. Monitoring systems can be set up to track the state over time, and they might go further, to alert the system administrator about exceptional conditions.

3.12.1 Ganglia

Support for the Ganglia monitoring system (http://ganglia.info/) is integral to HTCondor. Nagios (http://www.nagios.org/) is often used to provide alerts based on data from the Ganglia monitoring system. The *condor_gangliad* daemon provides an efficient way to take information from an HTCondor pool and supply it to the Ganglia monitoring system.

The *condor_gangliad* gathers up data as specified by its configuration, and it streamlines getting that data to the Ganglia monitoring system. Updates sent to Ganglia are done using the Ganglia shared libraries for efficiency.

If Ganglia is already deployed in the pool, the monitoring of HTCondor is enabled by running the *condor_gangliad* daemon on a single machine within the pool. If the machine chosen is the one running Ganglia's *gmetad*, then the HTCondor configuration consists of adding `GANGLIAD` to the definition of configuration variable `DAEMON_LIST` on that machine. It may be advantageous to run the *condor_gangliad* daemon on the same machine as is running the *condor_collector* daemon, because on a large pool with many ClassAds, there is likely to be less network traffic. If the *condor_gangliad* daemon is to run on a different machine than the one running Ganglia's *gmetad*, modify configuration variable `GANGLIA_GSTAT_COMMAND` to get the list of monitored hosts from the master *gmond* program.

If the pool does not use Ganglia, the pool can still be monitored by a separate server running Ganglia.

By default, the *condor_gangliad* will only propagate metrics to hosts that are already monitored by Ganglia. Set configuration variable `GANGLIA_SEND_DATA_FOR_ALL_HOSTS` to `True` to set up a Ganglia host to monitor a pool not monitored by Ganglia or have a heterogeneous pool where some hosts are not monitored. In this case, default graphs that Ganglia provides will not be present. However, the HTCondor metrics will appear.

On large pools, setting configuration variable `GANGLIAD_PER_EXECUTE_NODE_METRICS` to `False` will reduce the amount of data sent to Ganglia. The execute node data is the least important to monitor. One can also limit the amount of data by setting configuration variable `GANGLIAD_REQUIREMENTS`. Be aware that aggregate sums over the entire pool will not be accurate if this variable limits the ClassAds queried.

Metrics to be sent to Ganglia are specified in all files within the directory specified by configuration variable `GANGLIAD_METRICS_CONFIG_DIR`. Each file in the directory is read, and the format within each file is that of New ClassAds. Here is an example of a single metric definition given as a New ClassAd:

```
[
  Name   = "JobsSubmitted";
  Desc   = "Number of jobs submitted";
  Units  = "jobs";
  TargetType = "Scheduler";
]
```

A nice set of default metrics is in file: `$(GANGLIAD_METRICS_CONFIG_DIR)/00_default_metrics`.

Recognized metric attribute names and their use:

Name The name of this metric, which corresponds to the ClassAd attribute name. Metrics published for the same machine must have unique names.

Value A ClassAd expression that produces the value when evaluated. The default value is the value in the daemon ClassAd of the attribute with the same name as this metric.

Desc A brief description of the metric. This string is displayed when the user holds the mouse over the Ganglia graph for the metric.

Verbosity The integer verbosity level of this metric. Metrics with a higher verbosity level than that specified by configuration variable `GANGLIA_VERBOSITY` will not be published.

TargetType A string containing a comma-separated list of daemon ClassAd types that this metric monitors. The specified values should match the value of `MyType` of the daemon ClassAd. In addition, there are special values that may be included. `"Machine_slot1"` may be specified to monitor the machine ClassAd for slot 1 only. This is useful when monitoring machine-wide attributes. The special value `"ANY"` matches any type of ClassAd.

Requirements A boolean expression that may restrict how this metric is incorporated. It defaults to `True`, which places no restrictions on the collection of this ClassAd metric.

Title The graph title used for this metric. The default is the metric name.

Group A string specifying the name of this metric's group. Metrics are arranged by group within a Ganglia web page. The default is determined by the daemon type. Metrics in different groups must have unique names.

Cluster A string specifying the cluster name for this metric. The default cluster name is taken from the configuration variable `GANGLIAD_DEFAULT_CLUSTER`.

Units A string describing the units of this metric.

Scale A scaling factor that is multiplied by the value of the `Value` attribute. The scale factor is used when the value is not in the basic unit or a human-interpretable unit. For example, duty cycle is commonly expressed as a percent, but the HTCondor value ranges from 0 to 1. So, duty cycle is scaled by 100. Some metrics are reported in KiB. Scaling by 1024 allows Ganglia to pick the appropriate units, such as number of bytes rather than number of KiB. When scaling by large values, converting to the `"float"` type is recommended.

Derivative A boolean value that specifies if Ganglia should graph the derivative of this metric. Ganglia versions prior to 3.4 do not support this.

Type A string specifying the type of the metric. Possible values are `"double"`, `"float"`, `"int32"`, `"uint32"`, `"int16"`, `"uint16"`, `"int8"`, `"uint8"`, and `"string"`. The default is `"string"` for string values, the default is `"int32"` for integer values, the default is `"float"` for real values, and the default is `"int8"` for boolean values. Integer values can be coerced to `"float"` or `"double"`. This is especially important for values stored internally as 64-bit values.

Regex This string value specifies a regular expression that matches attributes to be monitored by this metric. This is useful for dynamic attributes that cannot be enumerated in advance, because their names depend on dynamic information such as the users who are currently running jobs. When this is specified, one metric per matching attribute is created. The default metric name is the name of the matched attribute, and the default value is the value of that attribute. As usual, the `Value` expression may be used when the raw attribute value needs to be manipulated before publication. However, since the name of the attribute is not known in advance, a special ClassAd attribute in the daemon ClassAd is provided to allow the `Value` expression to refer to it. This special attribute is named `Regex`. Another special feature is the ability to refer to text matched by regular expression

groups defined by parentheses within the regular expression. These may be substituted into the values of other string attributes such as `Name` and `Desc`. This is done by putting macros in the string values. `"\\1"` is replaced by the first group, `"\\2"` by the second group, and so on.

Aggregate This string value specifies an aggregation function to apply, instead of publishing individual metrics for each daemon ClassAd. Possible values are `"sum"`, `"avg"`, `"max"`, and `"min"`.

AggregateGroup When an aggregate function has been specified, this string value specifies which aggregation group the current daemon ClassAd belongs to. The default is the metric `Name`. This feature works like GROUP BY in SQL. The aggregation function produces one result per value of `AggregateGroup`. A single aggregate group would therefore be appropriate for a pool-wide metric. As an example, to publish the sum of an attribute across different types of slot ClassAds, make the metric name an expression that is unique to each type. The default `AggregateGroup` would be set accordingly. Note that the assumption is still that the result is a pool-wide metric, so by default it is associated with the *condor_collector* daemon's host. To group by machine and publish the result into the Ganglia page associated with each machine, make the `AggregateGroup` contain the machine name and override the default `Machine` attribute to be the daemon's machine name, rather than the *condor_collector* daemon's machine name.

Machine The name of the host associated with this metric. If configuration variable `GANGLIAD_DEFAULT_MACHINE` is not specified, the default is taken from the `Machine` attribute of the daemon ClassAd. If the daemon name is of the form `name@hostname`, this may indicate that there are multiple instances of HTCondor running on the same machine. To avoid the metrics from these instances overwriting each other, the default machine name is set to the daemon name in this case. For aggregate metrics, the default value of `Machine` will be the name of the *condor_collector* host.

IP A string containing the IP address of the host associated with this metric. If `GANGLIAD_DEFAULT_IP` is not specified, the default is extracted from the `MyAddress` attribute of the daemon ClassAd. This value must be unique for each machine published to Ganglia. It need not be a valid IP address. If the value of `Machine` contains an `"@"` sign, the default IP value will be set to the same value as `Machine` in order to make the IP value unique to each instance of HTCondor running on the same host.

3.12.2 Absent ClassAds

By default, HTCondor assumes that resources are transient: the *condor_collector* will discard ClassAds older than `CLASSAD_LIFETIME` seconds. Its default configuration value is 15 minutes, and as such, the default value for `UPDATE_INTERVAL` will pass three times before HTCondor forgets about a resource. In some pools, especially those with dedicated resources, this approach may make it unnecessarily difficult to determine what the composition of the pool ought to be, in the sense of knowing which machines would be in the pool, if HTCondor were properly functioning on all of them.

This assumption of transient machines can be modified by the use of absent ClassAds. When a machine ClassAd would otherwise expire, the *condor_collector* evaluates the configuration variable `ABSENT_REQUIREMENTS` against the machine ClassAd. If `True`, the machine ClassAd will be saved in a persistent manner and be marked as absent; this causes the machine to appear in the output of `condor_status -absent`. When the machine returns to the pool, its first update to the *condor_collector* will invalidate the absent machine ClassAd.

Absent ClassAds, like offline ClassAds, are stored to disk to ensure that they are remembered, even across *condor_collector* crashes. The configuration variable COLLECTOR_PERSISTENT_AD_LOG defines the file in which the ClassAds are stored, and replaces the no longer used variable OFFLINE_LOG. Absent ClassAds are retained on disk as maintained by the *condor_collector* for a length of time in seconds defined by the configuration variable ABSENT_EXPIRE_ADS_AFTER. A value of 0 for this variable means that the ClassAds are never discarded, and the default value is thirty days.

Absent ClassAds are only returned by the *condor_collector* and displayed when the **-absent** option to *condor_status* is specified, or when the absent machine ClassAd attribute is mentioned on the *condor_status* command line. This renders absent ClassAds invisible to the rest of the HTCondor infrastructure.

A daemon may inform the *condor_collector* that the daemon's ClassAd should not expire, but should be removed right away; the daemon asks for its ClassAd to be invalidated. It may be useful to place an invalidated ClassAd in the absent state, instead of having it removed as an invalidated ClassAd. An example of a ClassAd that could benefit from being absent is a system with an uninterruptible power supply that shuts down cleanly but unexpectedly as a result of a power outage. To cause all invalidated ClassAds to become absent instead of invalidated, set EXPIRE_INVALIDATED_ADS to True. Invalidated ClassAds will instead be treated as if they expired, including when evaluating ABSENT_REQUIREMENTS.

3.13 The High Availability of Daemons

In the case that a key machine no longer functions, HTCondor 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.

3.13.1 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 HTCondor 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).

HTCondor 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 = $(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 file `SCHEDD.lock` to `VALID_SPOOL_FILES`.

As HTCondor 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 3.5.7, 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 HTCondor will not modify the value set for this variable. See the description of MASTER_NAME in section 3.5.7 on page 242 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
```

3.13.2 High Availability of the Central Manager

Interaction with Flocking

The HTCondor high availability mechanisms discussed in this section currently do not work well in configurations involving flocking. The individual problems listed listed below 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 HTCondor matchmaking system. The availability of these daemons is critical to an HTCondor 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 HTCondor 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 are 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 chose 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. To properly do its job, the *condor_replication* daemon must transfer state files. When it needs to transfer a file, the *condor_replication* daemons at both the sending and receiving ends of the transfer invoke the *condor_transferer* daemon. These short lived daemons do the task of file transfer and then exit. Do not place `TRANSFERER` into `DAEMON_LIST`, as it is not a daemon that the *condor_master* should invoke or watch over.

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 3.5.26, for definitions of these configuration variables.

The *condor_had* and *condor_replication* daemons use the *condor_shared_port* daemon by default. If you want to use more than one *condor_had* or *condor_replication* daemon with the *condor_shared_port* daemon under the same master, you must configure those additional daemons to use nondefault socket names. (Set the `-sock` option in `<NAME>_ARGS`.) Because the *condor_had* daemon must know the *condor_replication* daemon's address a priori, you will also need to set `<NAME>.REPLICATION_SOCKET_NAME` appropriately.

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:

```
stabilization period = 12 * (number of central managers) *
                           $(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 too 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.

We begin with a sample configuration using shared port, and then include a sample configuration for not using shared port. Both samples relate to the high availability of central managers.

Each sample is split into two parts: the configuration for the central manager machines, and the configuration for the machines that will *not* be central managers.

The following shared-port configuration is for the central manager machines.

```
## THE FOLLOWING MUST BE IDENTICAL ON ALL CENTRAL MANAGERS

CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
CONDOR_HOST = $(CENTRAL_MANAGER1), $(CENTRAL_MANAGER2)

# Since we're using shared port, we set the port number to the shared
# port daemon's port number.  NOTE: this assumes that each machine in
# the list is using the same port number for shared port.  While this
# will be true by default, if you've changed it in configuration any-
# where, you need to reflect that change here.

HAD_USE_SHARED_PORT = TRUE
HAD_LIST = \
$(CENTRAL_MANAGER1):$(SHARED_PORT_PORT), \
$(CENTRAL_MANAGER2):$(SHARED_PORT_PORT)

REPLICATION_USE_SHARED_PORT = TRUE
REPLICATION_LIST = \
$(CENTRAL_MANAGER1):$(SHARED_PORT_PORT), \
$(CENTRAL_MANAGER2):$(SHARED_PORT_PORT)

# The recommended setting.
HAD_USE_PRIMARY = TRUE

# If you change which daemon(s) you're making highly-available, you must
# change both of these values.
HAD_CONTROLLEE = NEGOTIATOR
MASTER_NEGOTIATOR_CONTROLLER = HAD

## THE FOLLOWING MAY DIFFER BETWEEN CENTRAL MANAGERS

# The daemon list may contain additional entries.
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION

# Using replication is optional.
HAD_USE_REPLICATION = TRUE

# This is the default location for the state file.
STATE_FILE = $(SPOOL)/Accountantnew.log

# See note above the length of the negotiation cycle.
MASTER_HAD_BACKOFF_CONSTANT = 360
```

The following shared-port configuration is for the machines which that will *not* be central managers.

```
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
CONDOR_HOST = $(CENTRAL_MANAGER1), $(CENTRAL_MANAGER2)
```

The following configuration sets fixed port numbers for the central manager machines.

```
##########################################################################
# A sample configuration file for central managers, to enable the        #
# the high availability  mechanism.                                      #
##########################################################################

##########################################################################
## 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.
CONDOR_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 = -f -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 in the same order
## as CONDOR_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)

## 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 is the name of the daemon that the HAD controls.
## This must match the name of a daemon in the master's DAEMON_LIST.
## The default is NEGOTIATOR, but can be any daemon that the master
## controls.
HAD_CONTROLLEE = NEGOTIATOR

## 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


######################################################################
## THE PARAMETERS BELOW ARE ALLOWED TO BE DIFFERENT ON EACH       #
## CENTRAL MANAGER                                                #
## THESE ARE MASTER SPECIFIC PARAMETERS
######################################################################


## the master should start at least these four daemons
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION


## Enables/disables the replication feature of HAD daemon
## Default: false
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_TRANSFER_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 configuration for machines that will *not* be central managers is identical for the fixed- and shared- port cases.

```
##########################################################################
# Sample configuration relating to high availability for machines        #
# that DO NOT run the condor_had daemon.                                 #
##########################################################################

## For simplicity define a variable for each potential central manager
## in the pool.
CENTRAL_MANAGER1 = cm1.domain.name
CENTRAL_MANAGER2 = cm2.domain.name
## List of all potential central managers in the pool
CONDOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)
```

3.14 Setting Up for Special Environments

The following sections describe how to set up HTCondor for use in special environments or configurations.

3.14.1 Using HTCondor with AFS

Configuration variables that allow machines to interact with and use a shared file system are given at section 3.5.5.

Limitations with AFS occur because HTCondor does not currently have a way to authenticate itself to AFS. This is true of the HTCondor daemons that would like to authenticate as the AFS user `condor`, and of 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 special things to do when interacting with AFS. Some of this must be done by the administrator(s) installing HTCondor. Other things must be done by HTCondor users who submit jobs.

AFS and HTCondor for Administrators

The largest result from the lack of authentication with AFS is that the directory defined by the configuration variable `LOCAL_DIR` and its subdirectories `log` and `spool` on each machine must be either writable to unauthenticated users, or must not be on AFS. Making these directories writable a *very* bad security hole, so it is *not* a viable solution. Placing `LOCAL_DIR` onto NFS is acceptable. To avoid AFS, place the directory defined for `LOCAL_DIR` on a local partition on each machine in the pool. This implies running *condor_configure* to install the release directory and configure the pool, setting the `LOCAL_DIR` variable to a local partition. When that is complete, log into each machine in the pool, and run *condor_init* to set up the local HTCondor directory.

The directory defined by `RELEASE_DIR`, which holds all the HTCondor binaries, libraries, and scripts, can be on AFS. None of the HTCondor daemons need to write to these files. They only need to read them. So, the directory defined by `RELEASE_DIR` only needs to be world readable in order to let HTCondor function. This makes it easier to upgrade the binaries to a newer version at a later date, and means that users can find the HTCondor tools in a consistent location on all the machines in the pool. Also, the HTCondor configuration files may be placed in a centralized location. This is what we do for the UW-Madison's CS department HTCondor pool, and it works quite well.

Finally, consider setting up some targeted AFS groups to help users deal with HTCondor and AFS better. This is discussed in the following manual subsection. In short, create an AFS group that contains all users, authenticated or

not, but which is restricted to a given host or subnet. These should be made as host-based ACLs with AFS, but here at UW-Madison, we have had some trouble getting that working. Instead, we have a special group for all machines in our department. The users here are required 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 HTCondor for Users

The *condor_shadow* daemon runs on the machine where jobs are submitted. It performs all file system access on behalf of the jobs. Because the *condor_shadow* daemon is not authenticated to AFS as the user who submitted the job, the *condor_shadow* daemon will not normally be able to write any output. Therefore the directories in which the job will be creating output files will need to be world writable; they need to be writable by non-authenticated AFS users. In addition, the program's `stdout`, `stderr`, log file, and any file the program explicitly opens will need to be in a directory that is world-writable.

An administrator may be able to set up special AFS groups that can make unauthenticated access to the program's files less scary. For example, there is supposed to be a way for AFS to grant access to any unauthenticated process on a given host. If set up, write access need only be granted to unauthenticated processes on the submit machine, as opposed to any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running on the submit machine.

A solution to this problem is to not use AFS for output files. If disk space on the submit machine is available in a partition not on AFS, submit the jobs from there. While the *condor_shadow* daemon 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* daemon will be able to access the files, and no special permissions need be granted to anyone other than the job submitter. If the HTCondor daemons are not invoked as root however, the *condor_shadow* daemon will not be able to run with the submitter's effective UID, leading to a similar problem as with files on AFS.

3.14.2 Enabling the Transfer of Files Specified by a URL

Because staging data on the submit machine is not always efficient, HTCondor permits input files to be transferred from a location specified by a URL; likewise, output files may be transferred to a location specified by a URL. All transfers (both input and output) are accomplished by invoking a *plug-in*, an executable or shell script that handles the task of file transfer.

For transferring input files, URL specification is limited to jobs running under the vanilla universe and to a vm universe VM image file. The execute machine retrieves the files. This differs from the normal file transfer mechanism, in which transfers are from the machine where the job is submitted to the machine where the job is executed. Each file to be transferred by specifying a URL, causing a plug-in to be invoked, is specified separately in the job submit description file with the command **transfer_input_files**; see section 2.5.9 for details.

For transferring output files, either the entire output sandbox, which are all files produced or modified by the job as it executes, or a subset of these files, as specified by the submit description file command **transfer_output_files** are transferred to the directory specified by the URL. The URL itself is specified in the separate submit description file command **output_destination**; see section 2.5.9 for details. The plug-in is invoked once for each output file to be transferred.

Configuration identifies the availability of the one or more plug-in(s). The plug-ins must be installed and available on every execute machine that may run a job which might specify a URL, either for input or for output.

URL transfers are enabled by default in the configuration of execute machines. Disabling URL transfers is accomplished by setting

```
ENABLE_URL_TRANSFERS = FALSE
```

A comma separated list giving the absolute path and name of all available plug-ins is specified as in the example:

```
FILETRANSFER_PLUGINS = /opt/condor/plugins/wget-plugin, \
                       /opt/condor/plugins/hdfs-plugin, \
                       /opt/condor/plugins/custom-plugin
```

The *condor_starter* invokes all listed plug-ins to determine their capabilities. Each may handle one or more protocols (scheme names). The plug-in's response to invocation identifies which protocols it can handle. When a URL transfer is specified by a job, the *condor_starter* invokes the proper one to do the transfer. If more than one plugin is capable of handling a particular protocol, then the last one within the list given by `FILETRANSFER_PLUGINS` is used.

HTCondor assumes that all plug-ins will respond in specific ways. To determine the capabilities of the plug-ins as to which protocols they handle, the *condor_starter* daemon invokes each plug-in giving it the command line argument **-classad**. In response to invocation with this command line argument, the plug-in must respond with an output of three ClassAd attributes. The first two are fixed:

```
PluginVersion = "0.1"
PluginType = "FileTransfer"
```

The third ClassAd attribute is `SupportedMethods`. This attribute is a string containing a comma separated list of the protocols that the plug-in handles. So, for example

```
SupportedMethods = "http,ftp,file"
```

would identify that the three protocols described by `http`, `ftp`, and `file` are supported. These strings will match the protocol specification as given within a URL in a **transfer_input_files** command or within a URL in an **output_destination** command in a submit description file for a job.

When a job specifies a URL transfer, the plug-in is invoked, without the command line argument **-classad**. It will instead be given two other command line arguments. For the transfer of input file(s), the first will be the URL of the file to retrieve and the second will be the absolute path identifying where to place the transferred file. For the transfer of output file(s), the first will be the absolute path on the local machine of the file to transfer, and the second will be the URL of the directory and file name at the destination.

The plug-in is expected to do the transfer, exiting with status 0 if the transfer was successful, and a non-zero status if the transfer was *not* successful. When *not* successful, the job is placed on hold, and the job ClassAd attribute `HoldReason` will be set as appropriate for the job. The job ClassAd attribute `HoldReasonSubCode` will be set to the exit status of the plug-in.

As an example of the transfer of a subset of output files, assume that the submit description file contains

```
output_destination = url://server/some/directory/
transfer_output_files = foo, bar, qux
```

HTCondor invokes the plug-in that handles the `url` protocol three times. The directory delimiter (`/` on Unix, and `\` on Windows) is appended to the destination URL, such that the three (Unix) invocations of the plug-in will appear similar to

```
url_plugin /path/to/local/copy/of/foo url://server/some/directory//foo
url_plugin /path/to/local/copy/of/bar url://server/some/directory//bar
url_plugin /path/to/local/copy/of/qux url://server/some/directory//qux
```

Note that this functionality is not limited to a predefined set of protocols. New ones can be invented. As an invented example, the `zkm` transfer type writes random bytes to a file. The plug-in that handles `zkm` transfers would respond to invocation with the **-classad** command line argument with:

```
PluginVersion = "0.1"
PluginType = "FileTransfer"
SupportedMethods = "zkm"
```

And, then when a job requested that this plug-in be invoked, for the invented example:

```
transfer_input_files = zkm://128/r-data
```

the plug-in will be invoked with a first command line argument of `zkm://128/r-data` and a second command line argument giving the full path along with the file name `r-data` as the location for the plug-in to write 128 bytes of random data.

The transfer of output files in this manner was introduced in HTCondor version 7.6.0. Incompatibility and inability to function will result if the executables for the *condor_starter* and *condor_shadow* are versions earlier than HTCondor version 7.6.0. Here is the expected behavior for these cases that cannot be backward compatible.

- If the *condor_starter* version is earlier than 7.6.0, then regardless of the *condor_shadow* version, transfer of output files, as identified in the submit description file with the command **output_destination** is ignored. The files are transferred back to the submit machine.
- If the *condor_starter* version is 7.6.0 or later, but the *condor_shadow* version is earlier than 7.6.0, then the *condor_starter* will attempt to send the command to the *condor_shadow*, but the *condor_shadow* will ignore the command. No files will be transferred, and the job will be placed on hold.

3.14.3 Configuring HTCondor for Multiple Platforms

A single, initial configuration file may be used for all platforms in an HTCondor pool, with platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing specification 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 single initial configuration file, which will be shared by all platforms. 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.

The name of platform-specific configuration files may be specified by using $(ARCH) and $(OPSYS), as defined automatically by HTCondor. For example, for 32-bit Intel Windows 7 machines and 64-bit Intel Linux machines, the files ought to be named:

```
condor_config.INTEL.WINDOWS
condor_config.X86_64.LINUX
```

Then, assuming these files are in the directory defined by the ETC configuration variable, and machine-specific configuration files are in the same directory, named by each machine's host name, LOCAL_CONFIG_FILE becomes:

```
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, which lets AFS resolve this link based on platform name. 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.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:

RELEASE_DIR Full path to to the installed HTCondor 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.

MAIL The full path to the mail program.

CONSOLE_DEVICES Which devices in /dev should be treated as console devices.

DAEMON_LIST Which daemons the *condor_master* should start up. The reason this setting is platform-specific is to distinguish the *condor_kbdd*. It is needed on many Linux and Windows machines, and it is not needed on other platforms.

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 HTCondor 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 Linux or Windows machines, the DAEMON_LIST 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 HTCondor 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 attempting to change settings that the user should not be permitted to change, the settings would 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.

3.14.4 Full Installation of condor_compile

In order to take advantage of two major HTCondor features: checkpointing and remote system calls, users need to relink their binaries. Programs that are not relinked for HTCondor can run under HTCondor's vanilla universe. However, these jobs cannot take checkpoints and migrate.

To relink programs with HTCondor, we provide the *condor_compile* tool. As installed by default, *condor_compile* works with the following commands: *gcc*, *g++*, *g77*, *cc*, *acc*, *c89*, *CC*, *f77*, *fort77*, *ld*. See the *condor_compile*(1) man page for details on using *condor_compile*.

condor_compile can work transparently with all commands on the system, including *make*. The basic idea here is to replace the system linker (*ld*) with the HTCondor linker. Then, when a program is to be linked, the HTCondor linker

figures out whether this binary will be for HTCondor, 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 HTCondor, the script performs the necessary operations in order to prepare a binary that can be used with HTCondor. In order to differentiate between normal builds and HTCondor builds, the user simply places *condor_compile* before their build command, which sets the appropriate environment variable that lets the HTCondor 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 HTCondor 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 to execute depend upon the platform. The location of the system linker (*ld*), is as follows:

Operating System	Location of ld (ld-path)
Linux	/usr/bin

On these platforms, issue the following commands (as `root`), where *ld-path* is replaced by the path to the 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 HTCondor from your system later on, linking will continue to work, since the HTCondor 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 its former position as *ld*, overwriting the HTCondor 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*.

3.14.5 The *condor_kbdd*

The HTCondor 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. On Linux platforms, it is needed to detect USB keyboard activity. Otherwise, it is not needed. On Windows platforms, the *condor_kbdd* is the primary way of monitoring the idle time of both the keyboard and mouse.

The *condor_kbdd* on Windows Platforms

Windows platforms need to use the *condor_kbdd* to monitor the idle time of both the keyboard and mouse. By adding `KBDD` to configuration variable `DAEMON_LIST`, the *condor_master* daemon invokes the *condor_kbdd*, which then does the right thing to monitor activity given the version of Windows running.

With Windows Vista and more recent version of Windows, user sessions are moved out of session 0. Therefore, the *condor_startd* service is no longer able to listen to keyboard and mouse events. The *condor_kbdd* will run in an invisible window and should not be noticeable by the user, except for a listing in the task manager. When the user logs out, the program is terminated by Windows. This implementation also appears in versions of Windows that predate Vista, because it adds the capability of monitoring keyboard activity from multiple users.

To achieve the auto-start with user login, the HTCondor installer adds a *condor_kbdd* entry to the registry key at `HKLM\Software\Microsoft\Windows\CurrentVersion\Run`. On 64-bit versions of Vista and more recent Windows versions, the entry is actually placed in `HKLM\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run`.

In instances where the *condor_kbdd* is unable to connect to the *condor_startd*, it is likely because an exception was not properly added to the Windows firewall.

The *condor_kbdd* on Linux Platforms

On Linux platforms, great measures have been taken to make the *condor_kbdd* as robust as possible, but the X window system was not designed to facilitate such a need, and thus is not as efficient on machines where many users frequently log in and out on the console.

In order to work with X authority, which is the system by which X authorizes processes to connect to X servers, the *condor_kbdd* needs to run with super user privileges. Currently, the *condor_kbdd* assumes that X uses the `HOME` environment variable in order to locate a file named `.Xauthority`. This file contains keys necessary to connect to an X server. The keyboard daemon attempts to set `HOME` to various users' home directories in order to gain a connection to the X server and monitor events. This may fail to work if the keyboard daemon is not allowed to attach to the X server, and 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 files on AFS, because 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 logged in user. If running X11R6.3, the files to edit will probably be in `/usr/X11R6/lib/X11/xdm`. The `.xsession` file should start up the *condor_kbdd* at the end, and the `.Xreset` file should shut down the *condor_kbdd*. 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. The file's recommended location will be similar to `$HOME/.kbdd.log`, since this is a place where every user can write, and the file will not get in the way. The **-pidfile** and **-k** options allow for easy shut down of the *condor_kbdd* by storing the process ID in a file. It will be necessary to

add lines to the XDM configuration similar to

```
condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the *condor_kbdd* as the user who is currently logged in and write the log to a file in the directory `$HOME/.kbdd.log/`. This will also 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 file `~/.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.

3.14.6 Configuring The HTCondorView Server

The HTCondorView 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 HTCondorView 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 HTCondorView collector is slightly more complicated, while it offers the advantage that the same HTCondorView 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 HTCondorView server and to configure a pool to send updates to it.

Configuring a Machine to be a HTCondorView Server

To configure the HTCondorView collector, a few configuration variables are added or modified for the *condor_collector* chosen to act as the HTCondorView collector. These configuration variables are described in section 3.5.14 on page 286. 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 HTCondorView 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 HTCondorView server called (`POOL_HISTORY_MAX_STORAGE`).

NOTE: This directory should be separate and different from the `spool` or `log` directories already set up for HTCondor. There are a few problems putting these files into either of those directories.

KEEP_POOL_HISTORY A boolean value that determines if the HTCondorView 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 HTCondorView server host, create the directory specified in `POOL_HISTORY_DIR` and make it writable by the user the HTCondorView 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 HTCondorView collector, no further configuration is needed. To run a different *condor_collector* to act as the HTCondorView collector, configure HTCondor to automatically start it.

If using a separate host for the HTCondorView collector, to start it, add the value `COLLECTOR` to `DAEMON_LIST`, and restart HTCondor on that host. To run the HTCondorView 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 HTCondorView collector are started up by the same *condor_master* daemon on the same machine. In this example, the HTCondorView 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 HTCondorView Server

For the HTCondorView server to function, configure the existing collector to forward ClassAd updates to it. This configuration is only necessary if the HTCondorView collector is a different collector from the existing *condor_collector* for the pool. All the HTCondor daemons in the pool send their ClassAd updates to the regular *condor_collector*, which in turn will forward them on to the HTCondorView 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 HTCondorView collector. The full host name is optionally followed by a colon and port number. This is only necessary if the HTCondorView 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 HTCondorView collector will ignore this setting (as it should) as it notices that it is being asked to forward ClassAds to itself.

Once the HTCondorView 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 HTCondorView collector will verify that it is working. A query example:

```
condor_status -pool condor.view.host[:portnumber]
```

A *condor_collector* may also be configured to report to multiple HTCondorView servers. The configuration variable CONDOR_VIEW_HOST can be given as a list of HTCondorView servers separated by commas and/or spaces.

The following demonstrates an example configuration for two HTCondorView servers, where both HTCondorView servers (and the *condor_collector*) are running on the same machine, localhost.localdomain:

```
VIEWSERV01 = $(COLLECTOR)
VIEWSERV01_ARGS = -f -p 12345 -local-name VIEWSERV01
VIEWSERV01_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog01"
VIEWSERV01.POOL_HISTORY_DIR = $(LOCAL_DIR)/poolhist01
VIEWSERV01.KEEP_POOL_HISTORY = TRUE
VIEWSERV01.CONDOR_VIEW_HOST =

VIEWSERV02 = $(COLLECTOR)
VIEWSERV02_ARGS = -f -p 24680 -local-name VIEWSERV02
VIEWSERV02_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/ViewServerLog02"
VIEWSERV02.POOL_HISTORY_DIR = $(LOCAL_DIR)/poolhist02
VIEWSERV02.KEEP_POOL_HISTORY = TRUE
VIEWSERV02.CONDOR_VIEW_HOST =

CONDOR_VIEW_HOST = localhost.localdomain:12345 localhost.localdomain:24680
DAEMON_LIST = $(DAEMON_LIST) VIEWSERV01 VIEWSERV02
```

Note that the value of CONDOR_VIEW_HOST for VIEWSERV01 and VIEWSERV02 is unset, to prevent them from inheriting the global value of CONDOR_VIEW_HOST and attempting to report to themselves or each other. If the HTCondorView servers are running on different machines where there is no global value for CONDOR_VIEW_HOST, this precaution is not required.

3.14.7 Running HTCondor Jobs within a Virtual Machine

HTCondor jobs are formed from executables that are compiled to execute on specific platforms. This in turn restricts the machines within an HTCondor pool where a job may be executed. An HTCondor job may now be executed on a virtual machine running VMware, Xen, or KVM. This allows Windows executables to run on a Linux machine, and Linux executables to run on a Windows machine.

In older versions of HTCondor, 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 HTCondor, a *slot* (formerly known as *virtual machine*) refers to the multiple cores of a multi-core 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. Targeting an HTCondor job to run on an inner virtual machine is also different than using the **vm** universe. The **vm** universe lands and starts up a virtual machine instance, which is the HTCondor job, on an execute machine.

HTCondor 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, HTCondor understands the illusion, allowing an HTCondor job to be executed on only one at a time.

Installation and Configuration

HTCondor 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_VM_LIST`. 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 HTCondor daemons running on the host and on the inner virtual machine, the policy for when jobs may execute is set by HTCondor. While the host is executing an HTCondor job, the `START` policy on the inner virtual machine is overridden with `False`, so no HTCondor jobs will be started on the inner virtual machine. Conversely, while the inner virtual machine is executing an HTCondor job, the `START` policy on the host is overridden with `False`, so no HTCondor 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 ) )
```

3.14.8 HTCondor's Dedicated Scheduling

The dedicated scheduler is a part of the *condor_schedd* that handles the scheduling of parallel jobs that require more than one machine concurrently running per job. MPI applications are a common use for the dedicated scheduler, but parallel applications which do not require MPI can also be run with the dedicated scheduler. All jobs which use the parallel universe are routed to the dedicated scheduler within the *condor_schedd* they were submitted to. A default HTCondor installation does not configure a dedicated scheduler; the administrator must designate one or more *condor_schedd* daemons to perform as dedicated scheduler.

Selecting and Setting Up a Dedicated Scheduler

We recommend that you select a single machine within an HTCondor 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 HTCondor'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 162 for details on these issues.

Configuration Examples for Dedicated Resources

Each execute machine may have its own policy for the execution of jobs, as 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/examples/condor_config.local.dedicated.resource`.

Each execute machine defines the configuration variable `DedicatedScheduler`, which identifies the dedicated scheduler it is managed by. The local configuration file 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 HTCondor, 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 execute machines 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 HTCondor 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, configure 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 any other of the *condor_startd* daemon's pool-wide policies, such as those 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 Desktop Resources To The Mix A third policy example allows all jobs. These desktop 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 may preempt other jobs as defined by 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 a floating point value, so there is no harm in assigning a very large value.

Preemption with Dedicated Jobs

The dedicated scheduler can be configured to preempt running parallel universe jobs in favor of higher priority parallel universe jobs. Note that this is different from preemption in other universes, and parallel universe 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 parallel universe jobs. Two configuration variables control preemption of these dedicated resources: `SCHEDD_PREEMPTION_REQUIREMENTS` and `SCHEDD_PREEMPTION_RANK`. These variables 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 parallel universe job requires killing the entire job on all of its nodes. So, when preemption occurs, it may end up freeing more machines than are needed for the new job. Also, as HTCondor does not produce checkpoints for parallel universe jobs, preempted jobs will be re-run, starting again from the beginning. Thus, the administrator should be careful when enabling preemption of these dedicated resources. Enable dedicated preemption with the configuration:

```
STARTD_JOB_EXPRS = JobPrio
SCHEDD_PREEMPTION_REQUIREMENTS = (My.JobPrio < Target.JobPrio)
SCHEDD_PREEMPTION_RANK = 0.0
```

In this example, preemption is enabled by user-defined 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 newly acquired 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 within the same group. The most common example is a pool of machine using InfiniBand switches. For example, 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* feature supports this operation.

Each *condor_startd* must define which group it belongs to by setting the `ParallelSchedulingGroup` variable in the configuration file, and advertising it into the machine ClassAd. The value of this variable is a string, which should be the same for all *condor_startd* daemons within a given group. The property must be advertised in the *condor_startd* ClassAd by appending `ParallelSchedulingGroup` to the `STARTD_ATTRS` configuration variable.

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.

3.14.9 Configuring HTCondor for Running Backfill Jobs

HTCondor 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, HTCondor 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 HTCondor, details for configuring the policy for when backfill jobs are started or killed, and details on how to configure HTCondor to spawn the BOINC client to perform the work.

Overview of Backfill jobs in HTCondor

Whenever a resource controlled by HTCondor is in the Unclaimed/Idle state, it is totally idle; neither the interactive user nor an HTCondor 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 HTCondor 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 an HTCondor resource can enter and all of the possible transitions between them are described in section 3.7 beginning on page 352, especially sections 3.7.1, 3.7.1, and 3.7.1.

At this point, the only backfill system supported by HTCondor 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 HTCondor might provide additional functionality to make it easier to manage BOINC computations from within HTCondor. For now, the BOINC client must be manually installed and configured outside of HTCondor 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 3.5.8 on page 244.

ENABLE_BACKFILL A boolean value to determine if any backfill functionality should be used. The default value is `False`.

BACKFILL_SYSTEM 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 an HTCondor 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 an HTCondor 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 HTCondor 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, called the *boinc_client*, 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 instead of always working on a hard coded computation. The BOINC terminology for a computation is a *project*. 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 *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 HTCondor. 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 an HTCondor 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

In HTCondor Version 8.6.10, the *boinc_client* must be manually downloaded, installed and configured outside of HTCondor. Download the *boinc_client* executables at http://boinc.berkeley.edu/download.php.

Once the BOINC client software has been downloaded, the *boinc_client* binary should be placed in a location where the HTCondor daemons can use it. The path will be specified with the HTCondor configuration variable `BOINC_Executable`.

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. This is the same restriction as placed on the `spool` or `execute` directories used by HTCondor. The location of this directory is defined by `BOINC_InitialDir`. The directory must be writable by whatever user the *boinc_client* will run as. This user is either the same as the user the HTCondor daemons are running as, if HTCondor is not running as root, or a user defined via the `BOINC_Owner` configuration variable.

Finally, HTCondor 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_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), HTCondor 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 HTCondor to use BOINC for backfill computations.

Configuring the BOINC client under HTCondor

After the *boinc_client* has been installed on a given machine, the BOINC projects to join have been selected, and a unique project account key has been created for each project, the HTCondor configuration needs to be modified.

Whenever the *condor_startd* decides to spawn the *boinc_client* to perform backfill computations, 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

invoke any other HTCondor jobs. In fact, the argv[0] of the *boinc_client* will be renamed to *condor_exec*, as described in section 2.15.1 on page 156.

This *condor_starter* reads values out of the HTCondor configuration files to define the job it should run, as opposed to getting these values from a job ClassAd in the case of a normal HTCondor job. All of the configuration variables names for variables to control things such as the path to the *boinc_client* binary to use, the command-line arguments, and the initial working directory, are prefixed with the string `"BOINC_"`. Each of these variables is described as either a required or an optional configuration variable.

Required configuration variables:

BOINC_Executable The full path and executable name of the *boinc_client* binary to use.

BOINC_InitialDir The full path to the local directory where BOINC should run.

BOINC_Universe The HTCondor universe used for running the *boinc_client* program. This *must* be set to `vanilla` for BOINC to work under HTCondor.

BOINC_Owner What user the *boinc_client* program should be run as. This variable is only used if the HTCondor daemons are running as root. In this case, the *condor_starter* must be told what user identity to switch to before invoking the *boinc_client*. This can be any valid user on the local system, but it must have write permission in whatever directory is specified by `BOINC_InitialDir`.

Optional configuration variables:

BOINC_Arguments 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:

```
BOINC_Arguments = --attach_project http://einstein.phys.uwm.edu [account_key]
```

BOINC_Environment Environment variables that should be set for the *boinc_client*.

BOINC_Output Full path to the file where `stdout` from the *boinc_client* should be written. If this variable is not defined, `stdout` will be discarded.

BOINC_Error 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)

# Where to place stdout and stderr
BOINC_Output = $(BOINC_HOME)/boinc.out
BOINC_Error = $(BOINC_HOME)/boinc.err
```

If the HTCondor daemons reading this configuration are running as root, an additional variable must be defined:

```
# Specify the user that the boinc_client should run as:
BOINC_Owner = nobody
```

In this case, HTCondor 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 an HTCondor/BOINC installation to a given BOINC project:

- Use the **–attach_project** argument to the *boinc_client* program, defined via the `BOINC_Arguments` variable. 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 HTCondor resource has entered the Backfill state and has spawned the *boinc_client*.
- Manually create account files in the local BOINC directory. Upon start up, 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 an HTCondor pool, so administrators can either distribute them manually, or use symbolic links to point to a shared file system.

In the two cases of 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).

BOINC on Windows

The Windows version of BOINC has multiple installation methods. The preferred method of installation for use with HTCondor 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 HTCondor 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 7.2.4. For more information on the the STARTER_ALLOW_RUNAS_OWNER configuration variable, see section 3.5.5.

3.14.10 Per Job PID Namespaces

Per job PID namespaces provide enhanced isolation of one process tree from another through kernel level process ID namespaces. HTCondor may enable the use of per job PID namespaces for Linux RHEL 6, Debian 6, and more recent kernels.

Read about per job PID namespaces http://lwn.net/Articles/531419/.

The needed isolation of jobs from the same user that execute on the same machine as each other is already provided by the implementation of slot users as described in section 3.8.13. This is the recommended way to implement the prevention of interference between more than one job submitted by a single user. However, the use of a shared file system by slot users presents issues in the ownership of files written by the jobs.

The per job PID namespace provides a way to handle the ownership of files produced by jobs within a shared file system. It also isolates the processes of a job within its PID namespace. As a side effect and benefit, the clean up of processes for a job within a PID namespace is enhanced. When the process with PID = 1 is killed, the operating system takes care of killing all child processes.

To enable the use of per job PID namespaces, set the configuration to include

```
USE_PID_NAMESPACES = True
```

This configuration variable defaults to False, thus the use of per job PID namespaces is disabled by default.

3.14.11 Group ID-Based Process Tracking

One function that HTCondor often must perform is keeping track of all processes created by a job. This is done so that HTCondor can provide resource usage statistics about jobs, and also so that HTCondor 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 HTCondor uses a combination of process parent-child relationships, process groups, and information that HTCondor 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 HTCondor's use can be reliably tracked, since all HTCondor needs to do is look for all processes running using the given account. Administrators must specify in HTCondor's configuration what accounts can be considered dedicated via the `DEDICATED_EXECUTE_ACCOUNT_REGEXP` setting. See Section 3.8.13 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 HTCondor'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 HTCondor daemons run as `root`.

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 HTCondor.

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
```

If the defined range is too small, such that there is not a GID available when starting a job, then the *condor_starter* will fail as it tries to start the job. An error message will be logged stating that there are no more tracking GIDs.

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. Changes to `MIN_TRACKING_GID` and `MAX_TRACKING_GID` require a full restart of HTCondor.

3.14.12 Cgroup-Based Process Tracking

A new feature in Linux version 2.6.24 allows HTCondor to more accurately and safely manage jobs composed of sets of processes. This Linux feature is called Control Groups, or cgroups for short, and it is available starting with RHEL 6, Debian 6, and related distributions. Documentation about Linux kernel support for cgroups can be found in the Documentation directory in the kernel source code distribution. Another good reference is http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html Even if cgroup support is built into the kernel, many distributions do not install the cgroup tools by default.

The interface between the kernel cgroup functionality is via a (virtual) file system. When the condor_master starts on a Linux system with cgroup support in the kernel, it checks to see if cgroups are mounted, and if not, it will try to mount the cgroup virtual filesystem onto the directory /cgroup.

If your Linux distribution uses *systemd*, it will mount the cgroup file system, and the only remaining item is to set configuration variable `BASE_CGROUP`, as described below.

On Debian based systems, the memory cgroup controller is often not on by default, and needs to be enabled with a boot time option.

This setting needs to be inherited down to the per-job cgroup with the following commands in `rc.local`:

```
/usr/sbin/cgconfigparser -l /etc/cgconfig.conf
/bin/echo 1 > /sys/fs/cgroup/htcondor/cgroup.clone_children
```

When cgroups are correctly configured and running, the virtual file system mounted on `/cgroup` should have several subdirectories under it, and there should an `htcondor` subdirectory under the directory `/cgroup/cpu`.

The *condor_starter* daemon uses cgroups by default on Linux systems to accurately track all the processes started by a job, even when quickly-exiting parent processes spawn many child processes. As with the GID-based tracking, this is only implemented when a *condor_procd* daemon is running.

Kernel cgroups are named in a virtual file system hierarchy. HTCondor will put each running job on the execute node in a distinct cgroup. The name of this cgroup is the name of the execute directory for that *condor_starter*, with slashes replaced by underscores, followed by the name and number of the slot. So, for the memory controller, a job running on slot1 would have its cgroup located at `/cgroup/memory/htcondor/condor_var_lib_condor_execute_slot1/`. The `tasks` file in this directory will contain a list of all the processes in this cgroup, and many other files in this directory have useful information about resource usage of this cgroup. See the kernel documentation for full details.

Once cgroup-based tracking is configured, usage should be invisible to the user and administrator. The *condor_procd* log, as defined by configuration variable `PROCD_LOG`, will mention that it is using this method, but no user visible changes should occur, other than the impossibility of a quickly-forking process escaping from the control of the *condor_starter*, and the more accurate reporting of memory usage.

3.14.13 Limiting Resource Usage with a User Job Wrapper

An administrator can strictly limit the usage of system resources by jobs for any job that may be wrapped using the script defined by the configuration variable `USER_JOB_WRAPPER`. These are jobs within universes that are controlled by the *condor_starter* daemon, and they include the **vanilla**, **standard**, **java**, **local**, and **parallel** universes.

The job's ClassAd is written by the *condor_starter* daemon. It will need to contain attributes that the script defined by `USER_JOB_WRAPPER` can use to implement platform specific resource limiting actions. Examples of resources that may be referred to for limiting purposes are RAM, swap space, file descriptors, stack size, and core file size.

An initial sample of a `USER_JOB_WRAPPER` script is provided in the installation at `$(LIBEXEC)/condor_limits_wrapper.sh`. Here is the contents of that file:

```
#!/bin/bash
# Copyright 2008 Red Hat, Inc.
```

```
#
# 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.

if [[ $_CONDOR_MACHINE_AD != "" ]]; then
   mem_limit=$((`egrep '^Memory' $_CONDOR_MACHINE_AD | cut -d ' ' -f 3` * 1024))
   disk_limit=`egrep '^Disk' $_CONDOR_MACHINE_AD | cut -d ' ' -f 3`

   ulimit -d $mem_limit
   if [[ $? != 0 ]] || [[ $mem_limit = "" ]]; then
      echo "Failed to set Memory Resource Limit" > $_CONDOR_WRAPPER_ERROR_FILE
      exit 1
   fi
   ulimit -f $disk_limit
   if [[ $? != 0 ]] || [[ $disk_limit = "" ]]; then
      echo "Failed to set Disk Resource Limit" > $_CONDOR_WRAPPER_ERROR_FILE
      exit 1
   fi
fi

exec "$@"
error=$?
echo "Failed to exec($error): $@" > $_CONDOR_WRAPPER_ERROR_FILE
exit 1
```

If used in an unmodified form, this script sets the job's limits on a per slot basis for memory and disk usage, with the limits defined by the values in the machine ClassAd. This example file will need to be modified and merged for use with a preexisting USER_JOB_WRAPPER script.

If additional functionality is added to the script, an administrator is likely to use the USER_JOB_WRAPPER script in conjunction with SUBMIT_ATTRS or SUBMIT_EXPRS to force the job ClassAd to contain attributes that the USER_JOB_WRAPPER script expects to have defined.

The following variables are set in the environment of the the USER_JOB_WRAPPER script by the *condor_starter* daemon, when the USER_JOB_WRAPPER is defined.

_CONDOR_MACHINE_AD The full path and file name of the file containing the machine ClassAd.

_CONDOR_JOB_AD The full path and file name of the file containing the job ClassAd.

_CONDOR_WRAPPER_ERROR_FILE The full path and file name of the file that the USER_JOB_WRAPPER script should create, if there is an error. The text in this file will be included in any HTCondor failure messages.

3.14.14 Limiting Resource Usage Using Cgroups

While the method described to limit a job's resource usage is portable, and it should run on any Linux or BSD or Unix system, it suffers from one large flaw. The flaw is that resource limits imposed are per process, not per job. An HTCondor job is often composed of many Unix processes. If the method of limiting resource usage with a user job wrapper is used to impose a 2 Gigabyte memory limit, that limit applies to each process in the job individually. If a job created 100 processes, each using just under 2 Gigabytes, the job would continue without the resource limits kicking in. Clearly, this is not what the machine owner intends. Moreover, the memory limit only applies to the virtual memory size, not the physical memory size, or the resident set size. This can be a problem for jobs that use the `mmap` system call to map in a large chunk of virtual memory, but only need a small amount of memory at one time. Typically, the resource the administrator would like to control is physical memory, because when that is in short supply, the machine starts paging, and can become unresponsive very quickly.

The *condor_starter* can, using the Linux cgroup capability, apply resource limits collectively to sets of jobs, and apply limits to the physical memory used by a set of processes. The main downside of this technique is that it is only available on relatively new Unix distributions such as RHEL 6 and Debian 6. This technique also may require editing of system configuration files.

To enable cgroup-based limits, first ensure that cgroup-based tracking is enabled, as it is by default on supported systems, as described in section 3.14.12. Once set, the *condor_starter* will create a cgroup for each job, and set two attributes in that cgroup which control resource usage therein. These two attributes are the cpu.shares attribute in the cpu controller, and one of two attributes in the memory controller, either memory.limit_in_bytes, or memory.soft_limit_in_bytes. The configuration variable `CGROUP_MEMORY_LIMIT_POLICY` controls whether the hard limit (the former) or the soft limit will be used. If `CGROUP_MEMORY_LIMIT_POLICY` is set to the string `hard`, the hard limit will be used. If set to `soft`, the soft limit will be used. Otherwise, no limit will be set if the value is `none`. The default is `none`. If the hard limit is in force, then the total amount of physical memory used by the sum of all processes in this job will not be allowed to exceed the limit. If the processes try to allocate more memory, the allocation will succeed, and virtual memory will be allocated, but no additional physical memory will be allocated. The system will keep the amount of physical memory constant by swapping some page from that job out of memory. However, if the soft limit is in place, the job will be allowed to go over the limit if there is free memory available on the system. Only when there is contention between other processes for physical memory will the system force physical memory into swap and push the physical memory used towards the assigned limit. The memory size used in both cases is the machine ClassAd attribute `Memory`. Note that `Memory` is a static amount when using static slots, but it is dynamic when partitionable slots are used. That is, the limit is whatever the "Mem" column of condor_status reports for that slot. If the job exceeds both the physical memory and swap space, the job will be killed by the Linux Out-of-Memory killer, and HTCondor will put the job on hold with an appropriate message.

If `CGROUP_MEMORY_LIMIT_POLICY` is set, HTCondor will also also use cgroups to limit the amount of swap space used by each job. By default, the maximum amount of swap space used by each slot is the total amount of Virtual Memory in the slot, minus the amount of physical memory. Note that HTCondor measures virtual memory in kbytes, and physical memory in megabytes. To prevent jobs with high memory usage from thrashing and excessive paging, and force HTCondor to put them on hold instead, you can set a lower limit on the amount of swap space they are allowed to use. With partitionable slots, this is done in the per slot definition, and must be a percentage of the total swap space on the system. For example,

```
NUM_SLOTS_TYPE_1 = 1
```

```
SLOT_TYPE_1_PARTITIONABLE = true
SLOT_TYPE_1 = cpus=100%,swap=10%
```

Optionally, if the administrator sets the config file setting PROPORTIONAL_SWAP_ASSSIGNMENT = true, the maximum amount of swap space per slot will be set to the same proportion of the total swap as as the proportion of physical memory. That is, if a slot (static or dyanmic) has half of the physical memory of the machine, it will be given half of the swap space.

In addition to memory, the *condor_starter* can also control the total amount of CPU used by all processes within a job. To do this, it writes a value to the cpu.shares attribute of the cgroup cpu controller. The value it writes is copied from the Cpus attribute of the machine slot ClassAd multiplied by 100. Again, like the Memory attribute, this value is fixed for static slots, but dynamic under partitionable slots. This tells the operating system to assign cpu usage proportionally to the number of cpus in the slot. Unlike memory, there is no concept of soft or hard, so this limit only applies when there is contention for the cpu. That is, on an eight core machine, with only a single, one-core slot running, and otherwise idle, the job running in the one slot could consume all eight cpus concurrently with this limit in play, if it is the only thing running. If, however, all eight slots where running jobs, with each configured for one cpu, the cpu usage would be assigned equally to each job, regardless of the number of processes or threads in each job.

3.14.15 Concurrency Limits

Concurrency limits allow an administrator to limit the number of concurrently running jobs that declare that they use some pool-wide resource. This limit is applied globally to all jobs submitted from all schedulers across one HTCondor pool; the limits are *not* applied to scheduler, local, or grid universe jobs. This is useful in the case of a shared resource, such as an NFS or database server that some jobs use, where the administrator needs to limit the number of jobs accessing the server.

The administrator must predefine the names and capacities of the resources to be limited in the negotiator's configuration file. The job submitter must declare in the submit description file which resources the job consumes.

The administrator chooses a name for the limit. Concurrency limit names are case-insensitive. The names are formed from the alphabet letters 'A' to 'Z' and 'a' to 'z', the numerical digits 0 to 9, the underscore character '_' , and at most one period character. The names cannot start with a numerical digit.

For example, assume that there are 3 licenses for the X software, so HTCondor should constrain the number of running jobs which need the X software to 3. The administrator picks XSW as the name of the resource and sets the configuration

```
XSW_LIMIT = 3
```

where XSW is the invented name of this resource, and this name 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
```

will enforce a limit of at most 1 running job that declares a usage of an unnamed resource. If CONCURRENCY_LIMIT_DEFAULT is omitted from the configuration, then no limits are placed on the number of concurrently executing jobs 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"
```

Jobs may declare that they need more than one type of resource. In this case, specify a comma-separated list of resources:

```
concurrency_limits = XSW, DATABASE, FILESERVER
```

The units of these limits are arbitrary. This job consumes one unit of each resource. Jobs can declare that they use more than one unit with syntax that follows the resource name by a colon character and the integer number of resources. For example, if the above job uses three units of the file server resource, it is declared with

```
concurrency_limits = XSW, DATABASE, FILESERVER:3
```

If there are sets of resources which have the same capacity for each member of the set, the configuration may become tedious, as it defines each member of the set individually. A shortcut defines a name for a set. For example, define the sets called LARGE and SMALL:

```
CONCURRENCY_LIMIT_DEFAULT = 5
CONCURRENCY_LIMIT_DEFAULT_LARGE = 100
CONCURRENCY_LIMIT_DEFAULT_SMALL = 25
```

To use the set name in a concurrency limit, the syntax follows the set name with a period and then the set member's name. Continuing this example, there may be a concurrency limit named LARGE.SWLICENSE, which gets the capacity of the default defined for the LARGE set, which is 100. A concurrency limit named LARGE.DBSESSION will also have a limit of 100. A concurrency limit named OTHER.LICENSE will receive the default limit of 5, as there is no set named OTHER.

A concurrency limit may be evaluated against the attributes of a matched machine. This allows a job to vary what concurrency limits it requires based on the machine to which it is matched. To implement this, the job uses submit command **concurrency_limits_expr** instead of **concurrency_limits**. Consider an example in which execute machines

are located on one of two local networks. The administrator sets a concurrency limit to limit the number of network intensive jobs on each network to 10. Configuration of each execute machine advertises which local network it is on. A machine on "NETWORK_A" configures

```
NETWORK = "NETWORK_A"
STARTD_ATTRS = $(STARTD_ATTRS) NETWORK
```

and a machine on "NETWORK_B" configures

```
NETWORK = "NETWORK_B"
STARTD_ATTRS = $(STARTD_ATTRS) NETWORK
```

The configuration for the negotiator sets the concurrency limits:

```
NETWORK_A_LIMIT = 10
NETWORK_B_LIMIT = 10
```

Each network intensive job identifies itself by specifying the limit within the submit description file:

```
concurrency_limits_expr = TARGET.NETWORK
```

The concurrency limit is applied based on the network of the matched machine.

An extension of this example applies two concurrency limits. One limit is the same as in the example, such that it is based on an attribute of the matched machine. The other limit is of a specialized application called "SWX" in this example. The negotiator configuration is extended to also include

```
SWX_LIMIT = 15
```

The network intensive job that also uses two units of the SWX application identifies the needed resources in the single submit command:

```
concurrency_limits_expr = strcat("SWX:2 ", TARGET.NETWORK)
```

Submit command **concurrency_limits_expr** may not be used together with submit command **concurrency_limits**.

Note that it is possible, under unusual circumstances, for more jobs to be started than should be allowed by the concurrency limits feature. 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. If the limits are exceeded, HTCondor will not kill any job to reduce the number of running jobs to meet the limit.

3.15 Java Support Installation

Compiled Java programs may be executed (under HTCondor) on any execution site with a Java Virtual Machine (JVM). To do this, HTCondor must be informed of some details of the JVM installation.

Begin by installing a Java distribution according to the vendor's instructions. Your machine may have been delivered with a JVM already installed – installed code is frequently found in `/usr/bin/java`.

HTCondor'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.

Many implementations of the JVM set a value of the Java maximum heap size that is too small for particular applications. HTCondor uses this value. The administrator can change this value through configuration by setting a different value for `JAVA_EXTRA_ARGUMENTS`.

```
JAVA_EXTRA_ARGUMENTS = -Xmx1024m
```

Note that if a specific job sets the value in the submit description file, using the submit command **java_vm_args**, the job's value takes precedence over a configured value.

3.16 Setting Up the VM and Docker Universes

3.16.1 The VM Universe

vm universe jobs may be executed on any execution site with VMware, Xen (via *libvirt*), or KVM. To do this, HTCondor must be informed of some details of the virtual machine installation, and the execution machines must be configured correctly.

What follows is not a comprehensive list of the options that help set up to use the **vm** universe; rather, it is intended to serve as a starting point for those users interested in getting **vm** universe jobs up and running quickly. Details of configuration variables are in section 3.5.25.

Begin by installing the virtualization package on all execute machines, according to the vendor's instructions. We have successfully used VMware, Xen, and KVM. If considering running on a Windows system, a *Perl* distribution will also need to be installed; we have successfully used *ActivePerl*.

For VMware, *VMware Server 1* must be installed and running on the execute machine. HTCondor also supports using *VMware Workstation* and *VMware Player*, version 5. Earlier versions of these products may also work. HTCondor will attempt to automatically discern which VMware product is installed. If using *Player*, also install the *VIX API*, which is freely available from VMware.

For Xen, there are three things that must exist on an execute machine to fully support **vm** universe jobs.

1. A Xen-enabled kernel must be running. This running Xen kernel acts as Dom0, in Xen terminology, under which all VMs are started, called DomUs Xen terminology.
2. The *libvirtd* daemon must be available, and *Xend* services must be running.
3. The *pygrub* program must be available, for execution of VMs whose disks contain the kernel they will run.

For KVM, there are two things that must exist on an execute machine to fully support **vm** universe jobs.

1. The machine must have the KVM kernel module installed and running.
2. The *libvirtd* daemon must be installed and running.

Configuration is required to enable the execution of **vm** universe jobs. The type of virtual machine that is installed on the execute machine must be specified with the `VM_TYPE` variable. For now, only one type can be utilized per machine. For instance, the following tells HTCondor to use VMware:

```
VM_TYPE = vmware
```

The location of the *condor_vm-gahp* and its log file must also be specified on the execute machine. On a Windows installation, these options would look like this:

```
VM_GAHP_SERVER = $(SBIN)/condor_vm-gahp.exe
VM_GAHP_LOG = $(LOG)/VMGahpLog
```

VMware-Specific Configuration

To use VMware, identify the location of the *Perl* executable on the execute machine. In most cases, the default value should suffice:

```
VMWARE_PERL = perl
```

This, of course, assumes the *Perl* executable is in the path of the *condor_master* daemon. If this is not the case, then a full path to the *Perl* executable will be required.

If using *VMware Player*, which does not support snapshots, configure the `START` expression to reject jobs which require snapshots. These are jobs that do not have **vmware_snapshot_disk** set to `False`. Here is an example modification to the `START` expression.

```
START = ($(START)) && (!(TARGET.VMPARAM_VMware_SnapshotDisk =?= TRUE))
```

The final required configuration is the location of the VMware control script used by the *condor_vm-gahp* on the execute machine to talk to the virtual machine hypervisor. It is located in HTCondor's `sbin` directory:

```
VMWARE_SCRIPT = $(SBIN)/condor_vm_vmware
```

Note that an execute machine's `EXECUTE` variable should not contain any symbolic links in its path, if the machine is configured to run VMware **vm** universe jobs. Strange behavior has been noted when HTCondor 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 event log:

```
Error from starter on master_vmuniverse_strtd@nostos.cs.wisc
.edu: register(/scratch/gquinn/condor/git/CONDOR_SRC/src/con
dor_tests/31426/31426vmuniverse/execute/dir_31534/vmN3hylp_c
ondor.vmx) = 1/Error: Command failed: A file was not found/(
ERROR) Can't create snapshot for vm(/scratch/gquinn/condor/g
it/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.

Xen-Specific and KVM-Specific Configuration

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 reveal that 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 and KVM, the **vm** universe is only available when `root` starts HTCondor. This is a restriction currently imposed because root privileges are required to create a virtual machine on top of a Xen-enabled kernel. Specifically, root is needed to properly use the *libvirt* utility that controls creation and management of Xen and KVM guest virtual machines. This restriction may be lifted in future versions, depending on features provided by the underlying tool *libvirt*.

When a vm Universe Job Fails to Start

If a vm universe job should fail to launch, HTCondor will attempt to distinguish between a problem with the user's job description, and a problem with the virtual machine infrastructure of the matched machine. If the problem is with the job, the job will go on hold with a reason explaining the problem. If the problem is with the virtual machine infrastructure, HTCondor will reschedule the job, and it will modify the machine ClassAd to prevent any other vm universe job from matching. vm universe configuration is not slot-specific, so this change is applied to all slots.

When the problem is with the virtual machine infrastructure, these machine ClassAd attributes are changed:

- `HasVM` will be set to `False`
- `VMOfflineReason` will be set to a somewhat explanatory string
- `VMOfflineTime` will be set to the time of the failure
- `OfflineUniverses` will be adjusted to include `"VM"` and `13`

Since *condor_submit* adds `HasVM == True` to a vm universe job's requirements, no further vm universe jobs will match.

Once any problems with the infrastructure are fixed, to change the machine ClassAd attributes such that the machine will once again match to vm universe jobs, an administrator has three options. All have the same effect of setting the machine ClassAd attributes to the correct values such that the machine will not reject matches for vm universe jobs.

1. Restart the *condor_startd* daemon.
2. Submit a vm universe job that explicitly matches the machine. When the job runs, the code detects the running job and causes the attributes related to the vm universe to be set indicating that vm universe jobs can match with this machine.
3. Run the command line tool *condor_update_machine_ad* to set machine ClassAd attribute `HasVM` to `True`, and this will cause the other attributes related to the vm universe to be set indicating that vm universe jobs can match with this machine. See the *condor_update_machine_ad* manual page for examples and details.

3.16.2 The Docker Universe

The execution of a docker universe job causes the instantiation of a Docker container on an execute host.

The docker universe job is mapped to a vanilla universe job, and the submit description file must specify the submit command **docker_image** to identify the Docker image. The job's `requirement` ClassAd attribute is automatically appended, such that the job will only match with an execute machine that has Docker installed.

The Docker service must be pre-installed on each execute machine that can execute a docker universe job. Upon start up of the *condor_startd* daemon, the capability of the execute machine to run docker universe jobs is probed, and the machine ClassAd attribute `HasDocker` is advertised for a machine that is capable of running Docker universe jobs.

When a docker universe job is matched with a Docker-capable execute machine, HTCondor invokes the Docker CLI to instantiate the image-specific container. The job's scratch directory tree is mounted as a Docker volume. When the job completes, is put on hold, or is evicted, the container is removed.

An administrator of a machine can optionally make additional directories on the host machine readable and writable by a running container. To do this, the admin must first give an HTCondor name to each directory with the DOCKER_VOLUMES parameter. Then, each volume must be configured with the path on the host OS with the DOCKER_VOLUME_DIR_XXX parameter. Finally, the parameter DOCKER_MOUNT_VOLUMES tells HTCondor which of these directories to always mount onto containers running on this machine.

For example,

```
DOCKER_VOLUMES = SOME_DIR, ANOTHER_DIR
DOCKER_VOLUME_DIR_SOME_DIR = /path1
DOCKER_VOLUME_DIR_ANOTHER_DIR = /path/to/no2
DOCKER_MOUNT_VOLUMES = SOME_DIR, ANOTHER_DIR
```

The *condor_startd* will advertise which docker volumes it has available for mounting with the machine attributes HasDockerVolumeSOME_NAME = true so that jobs can match to machines with volumes they need.

Optionally, if the directory name is two directories, separated by a colon, the first directory is the name on the host machine, and the second is the value inside the container. If a ":ro" is specified after the second directory name, the volume will be mounted read-only inside the container.

These directories will be bind-mounted unconditionally inside the container. If an administrator wants to bind mount a directory only for some jobs, perhaps only those submitted by some trusted user, the setting `DOCKER_VOLUME_DIR_xxx_MOUNT_IF` may be used. This is a class ad expression, evaluated in the context of the job ad and the machine ad. Only when it evaluted to TRUE, is the volume mounted. Extending the above example,

```
DOCKER_VOLUMES = SOME_DIR, ANOTHER_DIR
DOCKER_VOLUME_DIR_SOME_DIR = /path1
DOCKER_VOLUME_DIR_SOME_DIR_MOUNT_IF = WantSomeDirMounted && Owner == "smith"
DOCKER_VOLUME_DIR_ANOTHER_DIR = /path/to/no2
DOCKER_MOUNT_VOLUMES = SOME_DIR, ANOTHER_DIR
```

In this case, the directory /path1 will get mounted inside the container only for jobs owned by user "smith", and who set +WantSomeDirMounted = true in their submit file.

In addition to installing the Docker service, the single configuration variable DOCKER must be set. It defines the location of the Docker CLI and can also specify that the *condor_starter* daemon has been given a password-less sudo permission to start the container as root. Details of the DOCKER configuration variable are in section 3.5.8.

Docker may be installed as root on a RedHat Linux machine these ordered steps.

1. Acquire and install the docker software:

   ```
   yum install docker-io
   ```

   Note that the *docker* package, which manages the window manager's dock, may need to be uninstalled, if it conflicts with this *docker-io* package.

2. Set up the groups:

   ```
   useradd -G docker condor
   ```

3. Invoke the docker software:

   ```
   service docker start
   ```

4. Reconfigure the execute machine, such that it can set the machine ClassAd attribute HasDocker:

   ```
   condor_reconfig
   ```

5. Check that the execute machine properly advertises that it is docker-capable with:

   ```
   condor_status -l | grep -i docker
   ```

   The output of this command line for a correctly-installed and docker-capable execute host will be similar to

   ```
   HasDocker = true
   DockerVersion = "Docker Version 1.6.0, build xxxxx/1.6.0"
   ```

By default, HTCondor will keep the 20 most recently used Docker images on the local machine. This number may be controlled with the configuration variable DOCKER_IMAGE_CACHE_SIZE, to increase or decrease the number of images, and the corresponding disk space, used by Docker.

By default, Docker containers will be run with all rootly capabilties dropped, and with setuid and setgid binaries disabled, for security reasons. If you need to run containers with root privilige, you may set the configuration parameter DOCKER_DROP_ALL_CAPABILITIES to an expression that evalutes to false. This expression is evaluted in the context of the machine ad (my) and the job ad (target).

3.17 Singularity Support

Note: This documentation is very basic and needs improvement!

Here's an example configuration file:

```
# Only set if singularity is not in $PATH.
#SINGULARITY = /opt/singularity/bin/singularity

# Forces _all_ jobs to run inside singularity.
SINGULARITY_JOB = true

# Forces all jobs to use the CernVM-based image.
SINGULARITY_IMAGE_EXPR = "/cvmfs/cernvm-prod.cern.ch/cvm3"

# Maps $_CONDOR_SCRATCH_DIR on the host to /srv inside the image.
SINGULARITY_TARGET_DIR = /srv

# Writable scratch directories inside the image.  Auto-deleted after the job exits.
MOUNT_UNDER_SCRATCH = /tmp, /var/tmp
```

This provides the user with no opportunity to select a specific image. Here are some changes to the above example to allow the user to specify an image path:

```
SINGULARITY_JOB = !isUndefined(TARGET.SingularityImage)
SINGULARITY_IMAGE_EXPR = TARGET.SingularityImage
```

Then, users could add the following to their submit file (note the quoting):

```
+SingularityImage = "/cvmfs/cernvm-prod.cern.ch/cvm3"
```

Finally, let's pick an image based on the OS – not the filename:

```
SINGULARITY_JOB = (TARGET.DESIRED_OS isnt MY.OpSysAndVer) && ((TARGET.DESIRED_OS is "C
SINGULARITY_IMAGE_EXPR = (TARGET.DESIRED_OS is "CentOS6") ? "/cvmfs/cernvm-prod.cern.c
```

Then, the user adds to their submit file:

```
+DESIRED_OS="CentOS6"
```

That would cause the job to run on the native host for CentOS6 hosts and inside a CentOS6 Singularity container on CentOS7 hosts.

3.18 Power Management

HTCondor supports placing machines in low power states. A machine in the low power state is identified as being offline. Power setting decisions are based upon HTCondor configuration.

Power conservation is relevant when machines are not in heavy use, or when there are known periods of low activity within the pool.

3.18.1 Entering a Low Power State

By default, HTCondor 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 (see section 3.5.8 on page 258) 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 extra time not at a low power state 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 reference 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.

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`.

```
TimeToWait  = (2 * $(HOUR))
ShouldHibernate = ( (KeyboardIdle > $(StartIdleTime)) \
                    && $(CPUIdle) \
                    && ($(StateTimer) > $(TimeToWait)) )
```

This macro evaluates to `True` if the following are all `True`:

- The keyboard has been idle long enough.
- The CPU is idle.
- The slot has been Unclaimed for more than 2 hours.

The sample HIBERNATE expression that enters the power state called "RAM", if ShouldHibernate evaluates to True, and remains in its current state otherwise is

```
HibernateState  = "RAM"
HIBERNATE = ifThenElse($(ShouldHibernate), $(HibernateState), "NONE" )
```

If any slot returns "NONE", that slot 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.

3.18.2 Returning From a Low Power State

The HTCondor 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 820.

To automatically call *condor_power* under specific conditions, *condor_rooster* may be used. The configuration options for *condor_rooster* are described in section 3.5.30.

3.18.3 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. This is required by *condor_rooster* so that it can evaluate the UNHIBERNATE expression of the offline machines.

To do this, define a log file using the OFFLINE_LOG configuration variable. See section 3.5.8 on page 260 for the definition. 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. See section 3.5.8 on page 260 for the definition.

3.18.4 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 HTCondor, 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 HTCondor, 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 HTCondor, this is defined by the string `"/proc"`.

By default, the HTCondor 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 3.5.8 on page 259. 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.

3.18.5 Windows Platform Details

If after a suitable amount of time, a Windows machine has not entered the expected power state, then HTCondor 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 HTCondor can offer little guidance. For operating system difficulties, 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 Sn 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.

Chapter 4

Miscellaneous Concepts

This chapter contains sections describing a variety of key HTCondor 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 HTCondor are presented.

The various hooks that HTCondor implements are described.

The many varieties of logs used by HTCondor are listed and described.

4.1 HTCondor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the HTCondor system. ClassAds are used extensively in the HTCondor system to represent jobs, resources, submitters and other HTCondor daemons. An understanding of this mechanism is required to harness the full flexibility of the HTCondor system.

A ClassAd is is a set of uniquely named expressions. Each named expression is called an *attribute*. Figure 4.1 shows ten attributes, a portion of an example ClassAd.

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 job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be *total* operators, which means that they

```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "LINUX"
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

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 ClassAds: Old and New

ClassAds have existed for quite some time in two forms: Old and New. Old ClassAds were the original form and were used in HTCondor until HTCondor version 7.5.0. They were heavily tied to the HTCondor development libraries. New ClassAds added new features and were designed as a stand-alone library that could be used apart from HTCondor.

In HTCondor version 7.5.1, HTCondor switched the internal usage of ClassAds from Old to New. All user interaction with tools (such as *condor_q*) as well as output of tools is still done as Old ClassAds. Before HTCondor version 7.5.1, New ClassAds were used in just a few places within HTCondor, for example, in the Job Router. There are some syntax and behavior differences between Old and New ClassAds, all of which will remain invisible to users of HTCondor for this version. A complete description of New ClassAds can be found at http://htcondor.org/classad/classad.html, and in the ClassAd Language Reference Manual found on that web page.

Some of the features of New ClassAds that are *not* in Old ClassAds are lists, nested ClassAds, time values, and matching groups of ClassAds. HTCondor has avoided using these features, as using them makes it difficult to interact with older versions of HTCondor. But, users can start using them if they do not need to interact with versions of HTCondor older than 7.5.1.

The syntax varies slightly between Old and New ClassAds. Here is an example ClassAd presented in both forms. The Old form:

```
Foo = 3
Bar = "ab\"cd\ef"
Moo = Foo =!= Undefined
```

The New form:

```
[
Foo = 3;
Bar = "ab\"cd\\ef";
Moo = Foo isnt Undefined;
]
```

HTCondor will convert to and from Old ClassAd syntax as needed.

New ClassAd Attribute References

Expressions often refer to ClassAd attributes. These attribute references work differently in Old ClassAds as compared with New ClassAds. In New ClassAds, an unscoped reference is looked for only in the local ClassAd. An *unscoped reference* is an attribute that does not have a `MY.` or `TARGET.` prefix. The *local ClassAd* may be described by an example. Matchmaking uses two ClassAds: the job ClassAd and the machine ClassAd. The job ClassAd is evaluated to see if it is a match for the machine ClassAd. The job ClassAd is the local ClassAd. Therefore, in the `Requirements` attribute of the job ClassAd, any attribute without the prefix `TARGET.` is looked up only in the job ClassAd. With New ClassAd evaluation, the use of the prefix `MY.` is eliminated, as an unscoped reference can only refer to the local ClassAd.

The `MY.` and `TARGET.` scoping prefixes only apply when evaluating an expression within the context of two ClassAds. Two examples that exemplify this are matchmaking and machine policy evaluation. When evaluating an expression within the context of a single ClassAd, `MY.` and `TARGET.` are not defined. Using them within the context of a single ClassAd will result in a value of `Undefined`. Two examples that exemplify evaluating an expression within the context of a single ClassAd are during user job policy evaluation, and with the **-constraint** option to command-line tools.

New ClassAds have no `CurrentTime` attribute. If needed, use the `time()` function instead. In order to mimic Old ClassAd semantics in this HTCondor version 7.5.1 release, all ClassAds have an explicit `CurrentTime` attribute, with a value of `time()`.

In current versions of HTCondor, New ClassAds will mimic the evaluation behavior of Old ClassAds. No configuration variables or submit description file contents should need to be changed. To eliminate this behavior and use only the semantics of New ClassAds, set the configuration variable `STRICT_CLASSAD_EVALUATION` to `True`. This permits testing expressions to see if any adjustment is required, before a future version of HTCondor potentially makes New ClassAds evaluation behavior the default or the only option.

4.1.2 Old ClassAd 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 attribute 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 4.1.3.

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 precedence is the unary minus operator. The only operators which are unfamiliar are the `=?=` and `=!=` operators, which are discussed in Section 4.1.3.

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 Section 4.1.2. Where the type may be any of these literal types, it is called out as `AnyType`. Where the type is

```
- (unary negation)    (high precedence)
*    /
+    - (addition, subtraction)
<    <=    >=    >
==   !=   =?=   =!=
&&
||                      (low precedence)
```

Figure 4.2: Relative precedence of ClassAd expression operators

`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 eval(AnyType Expr) Evaluates `Expr` as a string and then returns the result of evaluating the *contents* of the string as a ClassAd expression. This is useful when referring to an attribute such as `slotX_State` where `X`, the desired slot number is an expression, such as `SlotID+10`. In such a case, if attribute `SlotID` is 5, the value of the attribute `slot15_State` can be referenced using the expression `eval(strcat("slot", SlotID+10,"_State"))`. Function `strcat()` calls function `string()` on the second parameter, which evaluates the expression, and then converts the integer result 15 to the string `"15"`. The concatenated string returned by `strcat()` is `"slot15_State"`, and this string is then evaluated.

Note that referring to attributes of a job from within the string passed to `eval()` in the `Requirements` or `Rank` expressions could cause inaccuracies in HTCondor's automatic auto-clustering of jobs into equivalent groups for matchmaking purposes. This is because HTCondor needs to determine which ClassAd attributes are significant for matchmaking purposes, and indirect references from within the string passed to `eval()` will not be counted.

String unparse(Attribute attr) This function looks up the value of the provided attribute and returns the unparsed version as a string. The attribute's value is not evaluated. If the attribute's value is `x + 3`, then the function would return the string `"x + 3"`. If the provided attribute cannot be found, an empty string is returned.

This function returns `ERROR` if other than exactly 1 argument is given or the argument is not an attribute reference.

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 pow(Integer base, Integer exponent)

OR Real pow(Integer base, Integer exponent)

OR Real pow(Real base, Real exponent) Calculates `base` raised to the power of `exponent`. If `exponent` is an integer value greater than or equal to 0, and `base` is an integer, then an integer value is returned. If `exponent` is an integer value less than 0, or if either `base` or `exponent` is a real, then a real value is returned. An invocation with `exponent=0` or `exponent=0.0`, for any value of `base`, including 0 or 0.0, returns the value 1 or 1.0, type appropriate.

Integer quantize(AnyType a, Integer b)

OR Real quantize(AnyType a, Real b)

OR AnyType quantize(AnyType a, AnyType list b) `quantize()` computes the quotient of `a/b`, in order to further compute `ceiling(quotient) * b`. This computes and returns an integral multiple of `b` that is at least as large as `a`. So, when `b >= a`, the return value will be `b`. The return type is the same as that of `b`, where `b` is an Integer or Real.

When `b` is a list, `quantize()` returns the first value in the list that is greater than or equal to `a`. When no value in the list is greater than or equal to `a`, this computes and returns an integral multiple of the last member in the list that is at least as large as `a`.

This function returns `ERROR` if `a` or `b`, or a member of the list that must be considered is not an Integer or Real.

Here are examples:

```
8     = quantize(3, 8)
4     = quantize(3, 2)
0     = quantize(0, 4)
6.8   = quantize(1.5, 6.8)
7.2   = quantize(6.8, 1.2)
10.2  = quantize(10, 5.1)

4     = quantize(0, {4})
2     = quantize(2, {1, 2, "A"})
3.0   = quantize(3, {1, 2, 0.5})
3.0   = quantize(2.7, {1, 2, 0.5})
ERROR = quantize(3, {1, 2, "A"})
```

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 <= 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 join(String sep, AnyType Expr1 [, AnyType Expr2 ...])

OR String join(String sep, List list

OR String join(List list Returns the string which is the concatenation of all arguments after the first one. The first argument is the separator, and it is inserted between each of the other arguments during concatenation. All arguments are converted to type `String` by function `string(Expr)` before concatenation. When there are exactly two arguments, If the second argument is a List, all members of the list are converted to strings and then joined using the separator. When there is only one argument, and the argument is a List, all members of the list are converted to strings and then concatenated.

Returns `ERROR` if any argument evaluates to `UNDEFINED` or `ERROR`.

For example:

```
"a, b, c" = join(", ", "a", "b", "c")
"abc"   = join(split("a b c"))
"a;b;c" = join(";", split("a b c"))
```

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 other than exactly 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.

List split(String s [, String tokens]) Returns a list of the substrings of `s` that have been split up by using any of the characters within string `tokens`. If `tokens` is not specified, then all white space characters are used to delimit the string.

List splitUserName(String Name) Returns a list of two strings. Where `Name` includes an `@` character, the first string in the list will be the substring that comes before the `@` character, and the second string in the list will be the substring that comes after. Thus, if `Name` is `"user@domain"`, then the returned list will be `{"user", "domain"}`. If there is no `@` character in `Name`, then the first string in the list will be `Name`, and the second string in the list will be the empty string. Thus, if `Name` is `"username"`, then the returned list will be `{"username", ""}`.

List splitSlotName(String Name) Returns a list of two strings. Where `Name` includes an `@` character, the first string in the list will be the substring that comes before the `@` character, and the second string in the list will be the substring that comes after. Thus, if `Name` is `"slot1@machine"`, then the returned list will be `{"slot1", "machine"}`. If there is no `@` character in `Name`, then the first string in the list will be the empty string, and the second string in the list will be `Name`, Thus, if `Name` is `"machinename"`, then the returned list will be `{"", "machinename"}`.

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 universal 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".

AnyType debug(AnyType expression) This function evaluates its argument, and it returns the result. Thus, it is a no-operation. However, a side-effect of the function is that information about the evaluation is logged to the evaluating program's log file, at the D_FULLDEBUG debug level. This is useful for determining why a given ClassAd expression is evaluating the way it does. For example, if a *condor_startd* START expression is unexpectedly evaluating to UNDEFINED, then wrapping the expression in this debug() function will log information about each component of the expression to the log file, making it easier to understand the expression.

String envV1ToV2(String old_env) This function converts a set of environment variables from the old HTCondor syntax to the new syntax. The single argument should evaluate to a string that represents a set of environment variables using the old HTCondor syntax (usually stored in the job ClassAd attribute Env). The result is the same set of environment variables using the new HTCondor syntax (usually stored in the job ClassAd attribute Environment). If the argument evaluates to UNDEFINED, then the result is also UNDEFINED.

String mergeEnvironment(String env1 [, String env2, ...]) This function merges multiple sets of environment variables into a single set. If multiple arguments include the same variable, the one that appears last in the argument list is used. Each argument should evaluate to a string which represents a set of environment variables using the new HTCondor syntax or UNDEFINED, which is treated like an empty string. The result is a string that represents the merged set of environment variables using the new HTCondor syntax (suitable for use as the value of the job ClassAd attribute Environment).

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 stringListAvg(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.

Integer stringListsIntersect(String list1, String list2 [, String delimiter]) Returns `TRUE` if the lists contain any matching elements, and returns `FALSE` if the lists do not contain any matching elements. Returns `ERROR` if either argument is not a string or if an incorrect number of arguments are given.

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 caret (`^`) and dollar sign (`$`) characters. The caret 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.

Boolean regexp(String pattern, String target [, String options]) Uses the description of a regular expression given by string `pattern` to scan through the string `target`. Returns `TRUE` when `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]) Uses the description of a regular expression given by string `pattern` to scan through the string `target`. When `target` is a regular expression as described by `pattern`, the string `substitute` is returned, with backslash expansion performed. If any argument is not a string, returns `ERROR`.

Boolean stringList_regexpMember (String pattern, String list [, String delimiter] [, String options]) Uses the description of a regular expression given by string `pattern` to scan through the list of strings in `list`. Returns `TRUE` when one of the strings in `list` is a

regular expression as described by `pattern`. The optional `delimiter` describes how the list is delimited, and string `options` modifies how the match is performed. Returns `FALSE` if `pattern` does not match any entries in `list`. The return value is `ERROR`, if any of the arguments are not strings, or if `pattern` is not a valid regular expression.

String userHome(String userName [, String default]) Returns the home directory of the given user as configured on the current system (determined using the getpwdnam() call). (Returns `default` if the `default` argument is passed and the home directory of the user is not defined.)

List userMap(String mapSetName, String userName) Map an input string using the given mapping set. Returns a list of groups to which the user belongs.

String userMap(String mapSetName, String userName, String preferredGroup) Map an input string using the given mapping set. Returns a string, which is the preferred group if the user is in that group; otherwise it is the first group to which the user belongs, or undefined if the user belongs to no groups.

String userMap(String mapSetName, String userName, String preferredGroup, String default
Map an input string using the given mapping set. Returns a string, which is the preferred group if the user is in that group; the first group to which the user belongs, if any; and the default group if the user belongs to no groups.

The maps for the `userMap()` function are defined by the following configuration macros: `<SUBSYS>_CLASSAD_USER_MAP_NAMES` (see 3.5.1), `CLASSAD_USER_MAPFILE_<name>` (see 3.5.1) and `CLASSAD_USER_MAPDATA_<name>` (see 3.5.1).

4.1.3 Old ClassAd 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 ClassAd A that is being evaluated in a context with another ClassAd 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 $\star$, /, + 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 tests 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.

- **The Ternary operator:**

1. The Ternary operator (expr1 ? expr2 : expr3) operate with expressions. If all three expressions are given, the operation is strict.
2. However, if the middle expression is missing, eg. expr1 ?: expr3, then, when expr1 is defined, that defined value is returned. Otherwise, when expr1 evaluated to UNDEFINED, the value of expr3 is evaluated and returned. This can be a convenient shortcut for writing what would otherwise be a much longer classad expression.

Expression Examples

The =?= operator is similar to the == operator. It checks if the left hand side operand is identical in both type and value to the the right hand side operand, returning TRUE when they are identical. **For strings, the comparison is case-insensitive with the == operator and case-sensitive with the =?= operator.** A key point in understanding is that the =?= operator only produces evaluation results of TRUE and FALSE, where the == operator may produce evaluation results TRUE, FALSE, UNDEFINED, or ERROR. Table 4.1 presents examples that define the outcome of the == operator. Table 4.2 presents examples that define the outcome of the =?= operator.

expression	evaluated result
(10 == 10)	TRUE
(10 == 5)	FALSE
(10 == "ABC")	ERROR
"ABC" == "abc"	TRUE
(10 == UNDEFINED)	UNDEFINED
(UNDEFINED == UNDEFINED)	UNDEFINED

Table 4.1: Evaluation examples for the == operator

expression	evaluated result
(10 =?= 10)	TRUE
(10 =?= 5)	FALSE
(10 =?= "ABC")	FALSE
"ABC" =?= "abc"	FALSE
(10 =?= UNDEFINED)	FALSE
(UNDEFINED =?= UNDEFINED)	TRUE

Table 4.2: Evaluation examples for the =?= operator

The =!= operator is similar to the != operator. It checks if the left hand side operand is *not* identical in both type and value to the the right hand side operand, returning FALSE when they are identical. **For strings, the comparison is case-insensitive with the != operator and case-sensitive with the =!= operator.** A key point in understanding is that the =!= operator only produces evaluation results of TRUE and FALSE, where the != operator may produce evaluation results TRUE, FALSE, UNDEFINED, or ERROR. Table 4.3 presents examples that define the outcome of the != operator. Table 4.4 presents examples that define the outcome of the =!= operator.

expression	evaluated result
(10 != 10)	FALSE
(10 != 5)	TRUE
(10 != "ABC")	ERROR
"ABC" != "abc"	FALSE
(10 != UNDEFINED)	UNDEFINED
(UNDEFINED != UNDEFINED)	UNDEFINED

Table 4.3: Evaluation examples for the != operator

expression	evaluated result
(10 =!= 10)	FALSE
(10 =!= 5)	TRUE
(10 =!= "ABC")	TRUE
"ABC" =!= "abc"	TRUE
(10 =!= UNDEFINED)	TRUE
(UNDEFINED =!= UNDEFINED)	FALSE

Table 4.4: Evaluation examples for the =!= operator

4.1.4 Old ClassAds in the HTCondor System

The simplicity and flexibility of ClassAds is heavily exploited in the HTCondor system. ClassAds are not only used to represent machines and jobs in the HTCondor 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 ClassAds, users have a uniform and powerful mechanism to specify constraints over these ClassAds. These constraints can take the form of `Requirements` expressions in resource and job ClassAds, or queries over other ClassAds.

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 ClassAd may contain the following expressions:

```
Requirements = (Arch == "INTEL") && (OpSys == "LINUX")
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the job requires a 32-bit Intel processor running a Linux 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.

Needed syntax is different on Unix and Windows platforms, due to the interpretation of characters in forming command-line arguments. The expression must be a single command-line argument, and the resulting examples differ for the platforms. For Unix shells, single quote marks are used to delimit a single argument. For a Windows command window, double quote marks are used to delimit a single argument. Within the argument, Unix escapes the double quote mark by prepending a backslash to the double quote mark. Windows escapes the double quote mark by prepending another double quote mark. There may not be spaces in between.

Here are several examples. To find all computers which have had their keyboards idle for more than 60 minutes and have more than 4000 MB of memory, the desired ClassAd expression is

```
KeyboardIdle > 60*60 && Memory > 4000
```

On a Unix platform, the command appears as

```
% condor_status -const 'KeyboardIdle > 60*60 && Memory > 4000'

Name               OpSys   Arch   State     Activity LoadAv Mem  ActvtyTime
100
slot1@altair.cs.wi LINUX   X86_64 Owner     Idle     0.000 8018 13+00:31:46
slot2@altair.cs.wi LINUX   X86_64 Owner     Idle     0.000 8018 13+00:31:47
...
...
slot1@athena.stat. LINUX   X86_64 Unclaimed Idle     0.000 7946  0+00:25:04
slot2@athena.stat. LINUX   X86_64 Unclaimed Idle     0.000 7946  0+00:25:05
...
...
```

The Windows equivalent command is

```
>condor_status -const "KeyboardIdle > 60*60 && Memory > 4000"
```

Here is an example for a Unix platform 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
```

The ClassAd expression describing a machine that advertises a Windows operating system:

```
OpSys == "WINDOWS"
```

Here are three equivalent ways on a Unix platform to list all machines advertising a Windows operating system. Spaces appear in these examples to show where they are permitted.

```
% condor_status -constraint ' OpSys == "WINDOWS"  '


% condor_status -constraint OpSys==\"WINDOWS\"


% condor_status -constraint "OpSys==\"WINDOWS\""
```

The equivalent command on a Windows platform to list all machines advertising a Windows operating system must delimit the single argument with double quote marks, and then escape the needed double quote marks that identify the string within the expression. Spaces appear in this example where they are permitted.

```
>condor_status -constraint " OpSys == ""WINDOWS"" "
```

4.1.5 Extending ClassAds with User-written Functions

The ClassAd language provides a rich set of functions. It is possible to add new functions to the ClassAd language without recompiling the HTCondor system or the ClassAd library. This requires implementing the new function in the C++ programming language, compiling the code into a shared library, and telling HTCondor where in the file system the shared library lives.

While the details of the ClassAd implementation are beyond the scope of this document, the ClassAd source distribution ships with an example source file that extends ClassAds by adding two new functions, named `todays_date()` and `double()`. This can be used as a model for users to implement their own functions. To deploy this example extension, follow the following steps on Linux:

- Download the ClassAd source distribution from http://www.cs.wisc.edu/condor/classad.
- Unpack the tarball.
- Inspect the source file `shared.cpp`. This one file contains the whole extension.
- Build `shared.cpp` into a shared library. On Linux, the command line to do so is

```
$ g++ -DWANT_CLASSAD_NAMESPACE -I. -shared -o shared.so \
  -Wl,-soname,shared.so -o shared.so -fPIC shared.cpp
```

- Copy the file `shared.so` to a location that all of the HTCondor tools and daemons can read.

  ```
  $ cp shared.so `condor_config_val LIBEXEC`
  ```

- Tell HTCondor to load the shared library into all tools and daemons, by setting the `CLASSAD_USER_LIBS` configuration variable to the full name of the shared library. In this case,

  ```
  CLASSAD_USER_LIBS = $(LIBEXEC)/shared.so
  ```

- Restart HTCondor.

- Test the new functions by running

  ```
  $ condor_status -format "%s\n" todays_date()
  ```

4.2 HTCondor's Checkpoint Mechanism

A checkpoint is a snapshot of the current state of a program, taken in such a way that the program can be restarted from that state at a later time. Taking checkpoints gives the HTCondor 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 take a checkpoint of the job and preempt the job 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 checkpoints provides fault tolerance in HTCondor. Snapshots are taken periodically, and after an interruption in service the program can continue from the most recent snapshot.

HTCondor provides checkpoint services to single process jobs on some Unix platforms. To enable the taking of checkpoints, the user must link the program with the HTCondor 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 HTCondor checkpoints. However, the checkpoint services provided by HTCondor are strictly optional. So, while there are some classes of jobs for which HTCondor does not provide checkpoint services, these jobs may still be submitted to HTCondor to take advantage of HTCondor's resource management functionality. See section 2.4.1 on page 13 for a description of the classes of jobs for which HTCondor does not provide checkpoint services.

The taking of process checkpoints is implemented in the HTCondor system call library as a signal handler. When HTCondor 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.

HTCondor processes for which the taking of checkpoints is enabled take 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 using the checkpoint, and computation resumes from where it left off. Jobs that can not take checkpoints are preempted and restarted from the beginning.

HTCondor's taking of periodic checkpoints provides fault tolerance. Pools may be configured with the PERIODIC_CHECKPOINT variable, which controls when and how often jobs which can take and use checkpoints do so periodically. Examples of when are never, and every three hours. When the time to take a periodic checkpoint occurs, the job suspends processing, takes the checkpoint, and immediately continues from where it left off. There is also a *condor_ckpt* command which allows the user to request that an HTCondor job immediately take a periodic checkpoint.

In all cases, HTCondor jobs continue execution from the most recent complete checkpoint. If service is interrupted while a checkpoint is being taken, causing that checkpoint to fail, the process will restart from the previous checkpoint. HTCondor 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, taking a checkpoint may be delayed until a more appropriate time. For example, an HTCondor 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 be taken.

The HTCondor checkpoint feature can also be used for any Unix process outside of the HTCondor batch environment. Standalone checkpoints are described in section 4.2.1.

HTCondor can produce and use compressed checkpoints. Configuration variables (detailed in section 3.5.10 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. An HTCondor pool can also be configured with a checkpoint server or servers that serve as a repository for checkpoints, as described in section 3.10 on page 445. 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 for the purpose of storing checkpoints).

4.2.1 Standalone Checkpoint Mechanism

Using the HTCondor checkpoint library without the remote system call functionality and outside of the HTCondor system is known as the standalone mode checkpoint mechanism.

To prepare a program for taking standalone checkpoints, use the *condor_compile* utility as for a standard HTCondor job, but do not use *condor_submit*. Run the program from the command line. The checkpoint library will print a message to let you know that taking checkpoints is enabled and to inform you of the default name for the checkpoint image. The message is of the form:

```
HTCondor: Notice: Will checkpoint to program_name.ckpt
HTCondor: Notice: Remote system calls disabled.
```

Platforms that use address space randomization will need a modified invocation of the program, as described in section 7.1.1 on page 642. The invocation disables the address space randomization.

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, invoke the program with the command line 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
```

Again, platforms that implement address space randomization will need a modified invocation, as described in section 7.1.1.

By default, the program will restart in the same directory in which it originally ran, and the program will fail if it can not change to that absolute path. To suppress this behavior, also pass the *-_condor_relocatable* argument to the program. Not all programs will continue to work. Doing this may simplify moving standalone checkpoints between machines. Continuing the example given above, the command would be

```
P1 -_condor_restart P1.ckpt -_condor_relocatable
```

4.2.2 Checkpoint Safety

Some programs have fundamental limitations that make them unsafe for taking checkpoints. For example, a program that both reads and writes a single file may enter an unexpected state. Here is an example of the ordered events that exhibit this issue.

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, HTCondor displays a warning if a file is used for both reading and writing. You can ignore or disable these warnings if you choose as described in section 4.2.3, but please understand that your program may compute incorrect results.

4.2.3 Checkpoint Warnings

HTCondor displays warning messages upon encountering unexpected behaviors in the program. For example, if file `x` is opened for reading and writing, this message will be displayed:

```
HTCondor: Warning: READWRITE: File '/tmp/x' used for both reading and writing.
```

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, use `-_condor_warning` with no arguments.

For example, the additional command line argument to limit read/write warnings to one instance is

```
-_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()`.

4.2.4 Checkpoint Library Interface

A program need not be rewritten to take advantage of checkpoints. However, the checkpoint library provides several C entry points that allow for a program to control its own checkpoint behavior. These functions are provided.

- `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 the taking of checkpoints. This can be handy if the 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 the taking of checkpoints to be disabled.

- `void _condor_ckpt_enable()`
  This function re-enables the taking of checkpoints after a call to `_condor_ckpt_disable()`. If a checkpoint signal arrived while the taking of checkpoints was disabled, the checkpoint will be taken when this function is called. Disabling and enabling the taking of checkpoints 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 HTCondor. 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 1 (one) causes checkpoint images to be compressed. Setting it to 0 (zero) disables compression.

4.3 Computing On Demand (COD)

Computing On Demand (COD) extends HTCondor's high throughput computing abilities to include a method for running short-term jobs on instantly-available resources.

The motivation for COD extends HTCondor'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 HTCondor jobs to working on the COD jobs. These COD jobs (applications) cannot use the batch scheduling functionality of HTCondor, since the COD jobs require interactive response-time. Many of the applications that are well-suited to HTCondor'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 HTCondor pool. When the user presses a `recalculate` button, the predefined HTCondor 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 HTCondor helps these kinds of applications is to provide an infrastructure to use HTCondor batch resources for the types of compute nodes described above. HTCondor 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 HTCondor batch jobs. The user provides both the compute node applications and the interactive master application that controls them. HTCondor only provides a mechanism to allow these interactive (and often parallelized) applications to seamlessly interact with the HTCondor batch system.

4.3.1 Overview of How COD Works

The resources of an HTCondor 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 HTCondor administrator places a given user name in the `VALID_COD_USERS` list in the HTCondor 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 3.8.1 on HTCondor's Security Model. 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 in which to run the application, 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 HTCondor 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.

Independent 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 HTCondor's support. There is no mechanism to transfer files for a COD application, and all I/O must be handled locally or put 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 an HTCondor 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 HTCondor 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 HTCondor configuration files, and does not define any attributes with the *condor_cod activate* command-line tool, 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 (`"`).

RequestCpus Required when running on a *condor_startd* that uses partitionable slots. It specifies the number of CPU cores from the partitionable slot allocated for this job.

RequestDisk Required when running on a *condor_startd* that uses partitionable slots. It specifies the disk space, in Megabytes, from the partitionable slot allocated for this job.

RequestMemory Required when running on a *condor_startd* that uses partitionable slots. It specifies the memory, in Megabytes, from the partitionable slot allocated for this job.

The following list of attributes are *optional*:

JobUniverse This attribute defines what HTCondor job universe to use for the given COD application. The only tested universes are vanilla and java. This attribute must be an integer, with vanilla using the value 5, and java using the value 10.

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 file names. 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`. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

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 semicolon. 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 (`"`).

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 white space characters, and therefore, file names can not contain spaces.

KillSig This attribute specifies what signal should be sent whenever the HTCondor 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 job event log file specified for regular HTCondor jobs with the **Log** command in a submit description file. However, certain attributes that are placed in a job event log 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, HTCondor can write out this log in an XML representation, instead. To enable the XML format for this job event log, the StarterUserLogUseXML boolean is set to TRUE. The default if not specified is FALSE.

If any attribute that specifies a path (Cmd, In, Out,Err, StarterUserLog) is not a full path name, HTCondor 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. This job ID is similar to the job ID that is created whenever a regular HTCondor batch job is submitted. For regular HTCondor 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 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 HTCondor debugging logs described in section 3.5.2 on page 218. 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 job IDs are optional. The job ID is useful to define where it helps a user with the 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 that the ClusterId and ProcId identifiers can also be specified as command-line arguments to the *condor_cod activate* when spawning a given COD application. See section 4.3.4 below for details on using *condor_cod activate*.

Defining Attributes in the HTCondor Configuration Files

To define COD attributes in the HTCondor 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 variable 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 HTCondor 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, the command line `condor_cod activate` appears with the `-keyword FractGen` option.

NOTE: If a user is defining all attributes of their COD application in the HTCondor 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.

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* command using the **-jobad** option. These attributes are read by the *condor_cod* tool and passed through the system to 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 *condor_cod activate*. 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 HTCondor will automatically insert the correct value based on what user runs the *condor_cod* command and how that user authenticates to the COD resource. If a user defines an attribute that does not match the authenticated identity, HTCondor 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 HTCondor 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 HTCondor 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 glexec 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 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 HTCondor 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 HTCondor 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 HTCondor 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, an HTCondor 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 HTCondor 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 HTCondor 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 glexec will need a copy of this credential before the claim can be activated. Therefore, beginning with HTCondor version 6.9.2, COD users have access to a 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 HTCondor

HTCondor's support for COD has a few limitations:

- 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 and how often it is run.
- There is no accounting done for applications run under COD claims. Therefore, use of a lot of COD resources in a given HTCondor pool does not adversely affect user priority.

- COD claims are not persistent on a given *condor_startd* daemon.
- HTCondor does not provide a mechanism to parallelize a graphic application to take advantage of COD. The HTCondor Team is not in the business of developing applications, we only provide mechanisms to execute them.

4.4 Hooks

A *hook* is an external program or script invoked by HTCondor.

Job hooks that fetch work allow sites to write their own programs or scripts, and allow HTCondor 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, job hooks that fetch work allow HTCondor to more easily and directly interface with external scheduling systems.

Hooks may also behave as a Job Router.

The Daemon ClassAd hooks permit the *condor_startd* and the *condor_schedd* daemons to execute hooks once or on a periodic basis.

Note that standard universe jobs execute different *condor_starter* and *condor_shadow* daemons that do not implement any hook mechanisms.

4.4.1 Job Hooks That Fetch Work

In the past, HTCondor 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 HTCondor version 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 HTCondor 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 CCB 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 CCB.

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, compare it to the value of `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 HTCondor

There are a handful of hooks invoked by HTCondor related to fetching work, some of which are called by the *condor_startd* and others by the *condor_starter*. Each hook is described, including when it is invoked, what task it is supposed to accomplish, what data is passed to the hook, what output is expected, and, when relevant, the exit status expected.

Hook: Fetch Work The hook defined by the configuration variable `<Keyword>_HOOK_FETCH_WORK` is invoked whenever the *condor_startd* wants to see if there is any work to fetch. There is a related configuration variable 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 544. `<Keyword>_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 on COD Application Attributes, page 529.

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.

User id that the hook runs as The `<Keyword>_HOOK_FETCH_WORK` hook runs with the same privileges as the *condor_startd*. When Condor was started as `root`, this is usually the `condor` user, or the user specified in the `CONDOR_IDS` configuration variable.

Exit status of the hook Ignored.

Hook: Reply Fetch The hook defined by the configuration variable `<Keyword>_HOOK_REPLY_FETCH` is invoked whenever `<Keyword>_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 it ignores all output. The hook is simply advisory, and it 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.

User id that the hook runs as The `<Keyword>_HOOK_REPLY_FETCH` hook runs with the same privileges as the *condor_startd*. When Condor was started as `root`, this is usually the `condor` user, or the user specified in the `CONDOR_IDS` configuration variable.

Exit status of the hook Ignored.

Hook: Evict Claim The hook defined by the configuration variable `<Keyword>_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.

User id that the hook runs as The `<Keyword>_HOOK_EVICT_CLAIM` hook runs with the same privileges as the *condor_startd*. When Condor was started as `root`, this is usually the `condor` user, or the user specified in the `CONDOR_IDS` configuration variable.

Exit status of the hook Ignored.

Hook: Prepare Job The hook defined by the configuration variable `<Keyword>_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 set up 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.

User id that the hook runs as The `<Keyword>_HOOK_PREPARE_JOB` hook runs with the same privileges as the job itself. If slot users are defined, the hook runs as the slot user, just as the job does.

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 `<Keyword>_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, which is 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.

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.

Expected standard output from the hook None.

User id that the hook runs as The `<Keyword>_HOOK_UPDATE_JOB_INFO` hook runs with the same privileges as the job itself.

Exit status of the hook Ignored.

Hook: Job Exit The hook defined by the configuration variable `<Keyword>_HOOK_JOB_EXIT` is invoked by the *condor_starter* 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 job event 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 as the result of 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.

The job ClassAd passed to this hook contains all of the extra attributes described above for `<Keyword>_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.

Expected standard output from the hook None.

User id that the hook runs as The `<Keyword>_HOOK_JOB_EXIT` hook runs with the same privileges as the job itself.

Exit status of the hook Ignored.

Keywords to Define Job Fetch Hooks in the HTCondor Configuration files

Hooks are defined in the HTCondor 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 with `SLOT<N>_JOB_HOOK_KEYWORD`. For example, on slot 1, the variable name will be called `SLOT1_JOB_HOOK_KEYWORD`. If the slot-specific keyword is not defined, the *condor_startd* will use a global keyword as defined by `STARTD_JOB_HOOK_KEYWORD`.

Once a job is fetched via `<Keyword>_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:

- the *condor_startd* evaluates its own state
- 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 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 Hook: Specifying the Executable at Execution Time

The availability of multiple versions of an application leads to the need to specify one of the versions. As an example, consider that the java universe utilizes a single, fixed JVM. There may be multiple JVMs available, and the HTCondor job may need to make the choice of JVM version. The use of a job hook solves this problem. The job does not use

the java universe, and instead uses the vanilla universe in combination with a prepare job hook to overwrite the `Cmd` attribute of the job ClassAd. This attribute is the name of the executable the *condor_starter* daemon will invoke, thereby selecting the specific JVM installation.

In the configuration of the execute machine:

```
JAVA5_HOOK_PREPARE_JOB = $(LIBEXEC)/java5_prepare_hook
```

With this configuration, a job that sets the `HookKeyword` attribute with

```
+HookKeyword = "JAVA5"
```

in the submit description file causes the *condor_starter* will run the hook specified by `JAVA5_HOOK_PREPARE_JOB` before running this job. Note that the double quote marks are required to correctly define the attribute. Any output from this hook is an update to the job ClassAd. Therefore, the hook that changes the executable may be

```
#!/bin/sh

# Read and discard the job ClassAd
cat > /dev/null
echo 'Cmd = "/usr/java/java5/bin/java"'
```

If some machines in your pool have this hook and others do not, this fact should be advertised. Add to the configuration of every execute machine that has the hook:

```
HasJava5PrepareHook = True
STARTD_ATTRS = HasJava5PrepareHook $(STARTD_ATTRS)
```

The submit description file for this example job may be

```
universe = vanilla
executable = /usr/bin/java
arguments = Hello
# match with a machine that has the hook
requirements = HasJava5PrepareHook

should_transfer_files = always
when_to_transfer_output = on_exit
transfer_input_files = Hello.class

output = hello.out
error  = hello.err
log    = hello.log

+HookKeyword="JAVA5"
queue
```

Note that the **requirements** command ensures that this job matches with a machine that has JAVA5_HOOK_PREPARE_JOB defined.

4.4.2 Hooks for a Job Router

Job Router Hooks allow for an alternate transformation and/or monitoring than the *condor_job_router* daemon implements. Routing is still managed by the *condor_job_router* daemon, but if the Job Router Hooks are specified, then these hooks will be used to transform and monitor the job instead.

Job Router Hooks are similar in concept to Fetch Work Hooks, but they are limited in their scope. A hook is an external program or script invoked by the *condor_job_router* daemon at various points during the life cycle of a routed job.

The following sections describe how and when these hooks are used, what hooks are invoked at various stages of the job's life, and how to configure HTCondor to use these Hooks.

Hooks Invoked for Job Routing

The Job Router Hooks allow for replacement of the transformation engine used by HTCondor for routing a job. Since the external transformation engine is not controlled by HTCondor, additional hooks provide a means to update the job's status in HTCondor, and to clean up upon exit or failure cases. This allows one job to be transformed to just about any other type of job that HTCondor supports, as well as to use execution nodes not normally available to HTCondor.

It is important to note that if the Job Router Hooks are utilized, then HTCondor will not ignore or work around a failure in any hook execution. If a hook is configured, then HTCondor assumes its invocation is required 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, HTCondor will not fall back on its transformation accomplished without the hook to process the job.

There are 2 ways in which the Job Router Hooks may be enabled. A job's submit description file may cause the hooks to be invoked with

```
+HookKeyword = "HOOKNAME"
```

Adding this attribute to the job's ClassAd causes the *condor_job_router* daemon on the submit machine to invoke hooks prefixed with the defined keyword. HOOKNAME is a string chosen as an example; any string may be used.

The job's ClassAd attribute definition of HookKeyword takes precedence, but if not present, hooks may be enabled by defining on the submit machine the configuration variable

```
JOB_ROUTER_HOOK_KEYWORD = HOOKNAME
```

Like the example attribute above, HOOKNAME represents a chosen name for the hook, replaced as desired or appropriate.

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. All hooks must exit successfully.

Hook: Translate The hook defined by the configuration variable `<Keyword>_HOOK_TRANSLATE_JOB` 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 HTCondor 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 HTCondor'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 `<Keyword>_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 HTCondor's management of job attributes, only attributes that are not managed by HTCondor 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 `<Keyword>_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 `<Keyword>_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.

4.4.3 Daemon ClassAd Hooks

Overview

The *Daemon ClassAd Hook* mechanism is used to run executables (called jobs) directly from the *condor_startd* and *condor_schedd* daemons. The output from these jobs is incorporated into the machine ClassAd generated by the respective daemon. This mechanism and associated jobs have been identified by various names, including the *Startd Cron*, dynamic attributes, and a distribution of executables collectively known as *Hawkeye*.

Pool management tasks can be enhanced by using a daemon's ability to periodically run executables. The executables are expected to generate ClassAd attributes as their output; these ClassAds are then incorporated into the machine ClassAd. Policy expressions can then reference dynamic attributes (created by the ClassAd hook jobs) in the machine ClassAd.

Job output

The output of the job is incorporated into one or more ClassAds when the job exits. When the job outputs the special line:

```
- update:true
```

the output of the job is merged into all proper ClassAds, and an update goes to the *condor_collector* daemon.

As of version 8.3.0, it is possible for a *Startd Cron* job (but not a *Schedd Cron* job) to define multiple ClassAds, using the mechanism defined below:

- An output line starting with `'-'` has always indicated end-of-ClassAd. The `'-'` can now be followed by a uniqueness tag to indicate the name of the ad that should be replaced by the new ad. This name is joined to the name of the *Startd Cron* job to produced a full name for the ad. This allows a single *Startd Cron* job to return multiple ads by giving each a unique name, and to replace multiple ads by using the same unique name as a previous invocation. The optional uniqueness tag can also be followed by the optional keyword `update:<bool>`, which can be used to override the *Startd Cron* configuration and suppress or force immediate updates.

  In other words, the syntax is:

  - [*name*] [**update:** *bool*]

- Each ad can contain one of four possible attributes to control what slot ads the ad is merged into when the *condor_startd* sends updates to the collector. These attributes are, in order of highest to lower priority (in other words, if `SlotMergeConstraint` matches, the other attributes are not considered, and so on):
  - **SlotMergeConstraint** *expression*: the current ad is merged into all slot ads for which this expression is true. The expression is evaluated with the slot ad as the TARGET ad.
  - **SlotName|Name** *string*: the current ad is merged into all slots whose `Name` attributes match the value of `SlotName` up to the length of `SlotName`.

- **SlotTypeId** *integer*: the current ad is merged into all ads that have the same value for their `SlotTypeId` attribute.
- **SlotId** *integer*: the current ad is merged into all ads that have the same value for their `SlotId` attribute.

For example, if the *Startd Cron* job returns:

```
Value=1
SlotId=1
-s1
Value=2
SlotId=2
-s2
Value=10
- update:true
```

it will set `Value=10` for all slots except slot1 and slot2. On those slots it will set `Value=1` and `Value=2` respectively. It will also send updates to the collector immediately.

Configuration

Configuration variables related to Daemon ClassAd Hooks are defined in section 3.5.32.

Here is a complete configuration example. It defines all three of the available types of jobs: ones that use the *condor_startd*, benchmark jobs, and ones that use the *condor_schedd*.

```
#
# Startd Cron Stuff
#
# auxiliary variable to use in identifying locations of files
MODULES = $(ROOT)/modules

STARTD_CRON_CONFIG_VAL = $(RELEASE_DIR)/bin/condor_config_val
STARTD_CRON_MAX_JOB_LOAD = 0.2
STARTD_CRON_JOBLIST =

# Test job
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) test
STARTD_CRON_TEST_MODE = OneShot
STARTD_CRON_TEST_RECONFIG_RERUN = True
STARTD_CRON_TEST_PREFIX = test_
STARTD_CRON_TEST_EXECUTABLE = $(MODULES)/test
STARTD_CRON_TEST_KILL = True
STARTD_CRON_TEST_ARGS = abc 123
STARTD_CRON_TEST_SLOTS = 1
STARTD_CRON_TEST_JOB_LOAD = 0.01

# job 'date'
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) date
STARTD_CRON_DATE_MODE = Periodic
```

```
STARTD_CRON_DATE_EXECUTABLE = $(MODULES)/date
STARTD_CRON_DATE_PERIOD = 15s
STARTD_CRON_DATE_JOB_LOAD = 0.01

# Job 'foo'
STARTD_CRON_JOBLIST = $(STARTD_CRON_JOBLIST) foo
STARTD_CRON_FOO_EXECUTABLE = $(MODULES)/foo
STARTD_CRON_FOO_PREFIX = Foo
STARTD_CRON_FOO_MODE = Periodic
STARTD_CRON_FOO_PERIOD = 10m
STARTD_CRON_FOO_JOB_LOAD = 0.2

#
# Benchmark Stuff
#
BENCHMARKS_JOBLIST = mips kflops

# MIPS benchmark
BENCHMARKS_MIPS_EXECUTABLE = $(LIBEXEC)/condor_mips
BENCHMARKS_MIPS_JOB_LOAD = 1.0

# KFLOPS benchmark
BENCHMARKS_KFLOPS_EXECUTABLE = $(LIBEXEC)/condor_kflops
BENCHMARKS_KFLOPS_JOB_LOAD = 1.0

#
# Schedd Cron Stuff
#
SCHEDD_CRON_CONFIG_VAL = $(RELEASE_DIR)/bin/condor_config_val
SCHEDD_CRON_JOBLIST =

# Test job
SCHEDD_CRON_JOBLIST = $(SCHEDD_CRON_JOBLIST) test
SCHEDD_CRON_TEST_MODE = OneShot
SCHEDD_CRON_TEST_RECONFIG_RERUN = True
SCHEDD_CRON_TEST_PREFIX = test_
SCHEDD_CRON_TEST_EXECUTABLE = $(MODULES)/test
SCHEDD_CRON_TEST_PERIOD = 5m
SCHEDD_CRON_TEST_KILL = True
SCHEDD_CRON_TEST_ARGS = abc 123
```

4.5 Logging in HTCondor

HTCondor records many types of information in a variety of logs. Administration may require locating and using the contents of a log to debug issues. Listed here are details of the logs, to aid in identification.

4.5.1 Job and Daemon Logs

job event log The job event log is an optional, chronological list of events that occur as a job runs. The job event log is written on the submit machine. The submit description file for the job requests a job event log with the

submit command **log**. The log is created and remains on the submit machine. Contents of the log are detailed in section 2.6.7. Examples of events are that the job is running, that the job is placed on hold, or that the job completed.

daemon logs Each daemon configured to have a log writes events relevant to that daemon. Each event written consists of a timestamp and message. The name of the log file is set by the value of configuration variable `<SUBSYS>_LOG`, where `<SUBSYS>` is replaced by the name of the daemon. The log is not permitted to grow without bound; log rotation takes place after a configurable maximum size or length of time is encountered. This maximum is specified by configuration variable `MAX_<SUBSYS>_LOG`.

Which events are logged for a particular daemon are determined by the value of configuration variable `<SUBSYS>_DEBUG`. The possible values for `<SUBSYS>_DEBUG` categorize events, such that it is possible to control the level and quantity of events written to the daemon's log.

Configuration variables that affect daemon logs are

MAX_NUM_<SUBSYS>_LOG

TRUNC_<SUBSYS>_LOG_ON_OPEN

<SUBSYS>_LOG_KEEP_OPEN

<SUBSYS>_LOCK

FILE_LOCK_VIA_MUTEX

TOUCH_LOG_INTERVAL

LOGS_USE_TIMESTAMP

Daemon logs are often investigated to accomplish administrative debugging. *condor_config_val* can be used to determine the location and file name of the daemon log. For example, to display the location of the log for the *condor_collector* daemon, use

```
condor_config_val COLLECTOR_LOG
```

job queue log The job queue log is a transactional representation of the current job queue. If the *condor_schedd* crashes, the job queue can be rebuilt using this log. The file name is set by configuration variable `JOB_QUEUE_LOG`, and defaults to `$(SPOOL)/job_queue.log`.

Within the log, each transaction is identified with an integer value and followed where appropriate with other values relevant to the transaction. To reduce the size of the log and remove any transactions that are no longer relevant, a copy of the log is kept by renaming the log at each time interval defined by configuration variable `QUEUE_CLEAN_INTERVAL`, and then a new log is written with only current and relevant transactions.

Configuration variables that affect the job queue log are

SCHEDD_BACKUP_SPOOL

ROTATE_HISTORY_DAILY

ROTATE_HISTORY_MONTHLY

QUEUE_CLEAN_INTERVAL

MAX_JOB_QUEUE_LOG_ROTATIONS

***condor_schedd* audit log** The optional *condor_schedd* audit log records user-initiated events that modify the job queue, such as invocations of *condor_submit*, *condor_rm*, *condor_hold* and *condor_release*. Each event has a time stamp and a message that describes details of the event.

This log exists to help administrators track the activities of pool users.

The file name is set by configuration variable `SCHEDD_AUDIT_LOG`.

Configuration variables that affect the audit log are

MAX_SCHEDD_AUDIT_LOG

MAX_NUM_SCHEDD_AUDIT_LOG

***condor_shared_port* audit log** The optional *condor_shared_port* audit log records connections made through the `DAEMON_SOCKET_DIR`. Each record includes the source address, the socket file name, and the target process's PID, UID, GID, executable path, and command line.

This log exists to help administrators track the activities of pool users.

The file name is set by configuration variable `SHARED_PORT_AUDIT_LOG`.

Configuration variables that affect the audit log are

MAX_SHARED_PORT_AUDIT_LOG

MAX_NUM_SHARED_PORT_AUDIT_LOG

event log The event log is an optional, chronological list of events that occur for all jobs and all users. The events logged are the same as those that would go into a job event log. The file name is set by configuration variable `EVENT_LOG`. The log is created only if this configuration variable is set.

Configuration variables that affect the event log, setting details such as the maximum size to which this log may grow and details of file rotation and locking are

EVENT_LOG_MAX_SIZE

EVENT_LOG_MAX_ROTATIONS

EVENT_LOG_LOCKING

EVENT_LOG_FSYNC

EVENT_LOG_ROTATION_LOCK

EVENT_LOG_JOB_AD_INFORMATION_ATTRS

EVENT_LOG_USE_XML

accountant log The accountant log is a transactional representation of the *condor_negotiator* daemon's database of accounting information, which are user priorities. The file name of the accountant log is `$(SPOOL)/Accountantnew.log`. Within the log, users are identified by `username@uid_domain`.

To reduce the size and remove information that is no longer relevant, a copy of the log is made when its size hits the number of bytes defined by configuration variable `MAX_ACCOUNTANT_DATABASE_SIZE`, and then a new log is written in a more compact form.

Administrators can change user priorities kept in this log by using the command line tool *condor_userprio*.

negotiator match log The negotiator match log is a second daemon log from the *condor_negotiator* daemon. Events written to this log are those with debug level of `D_MATCH`. The file name is set by configuration variable `NEGOTIATOR_MATCH_LOG`, and defaults to `$(LOG)/MatchLog`.

history log This optional log contains information about all jobs that have been completed. It is written by the *condor_schedd* daemon. The file name is `$(SPOOL)/history`.

Administrators can change view this historical information by using the command line tool *condor_history*.

Configuration variables that affect the history log, setting details such as the maximum size to which this log may grow are

ENABLE_HISTORY_ROTATION

MAX_HISTORY_LOG

MAX_HISTORY_ROTATIONS

4.5.2 DAGMan Logs

default node log A job event log of all node jobs within a single DAG. It is used to enforce the dependencies of the DAG.

The file name is set by configuration variable `DAGMAN_DEFAULT_NODE_LOG`, and the full path name of this file must be unique while any and all submitted DAGs and other jobs from the submit host run. The syntax used in the definition of this configuration variable is different to enable the setting of a unique file name. See section 3.5.23 for the complete definition.

Configuration variables that affect this log are

DAGMAN_ALWAYS_USE_NODE_LOG

the `.dagman.out` file A log created or appended to for each DAG submitted with timestamped events and extra information about the configuration applied to the DAG. The name of this log is formed by appending `.dagman.out` to the name of the DAG input file. The file remains after the DAG completes.

This log may be helpful in debugging what has happened in the execution of a DAG, as well as help to determine the final state of the DAG.

Configuration variables that affect this log are

DAGMAN_VERBOSITY

DAGMAN_PENDING_REPORT_INTERVAL

the `jobstate.log` file This optional, machine-readable log enables automated monitoring of DAG. Section 2.10.14 details this log.

Chapter 5

Grid Computing

5.1 Introduction

A goal of grid computing is to allow the utilization of resources that span many administrative domains. An HTCondor 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 HTCondor pool. HTCondor shines in grid computing, continuing to evolve with the field.

Due to the field's rapid evolution, HTCondor has its own native mechanisms for grid computing as well as developing interactions with other grid systems.

Flocking is a native mechanism that allows HTCondor jobs submitted from within one pool to execute on another, separate HTCondor 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 HTCondor 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 HTCondor job, the job is submitted on a machine in the local HTCondor pool. The location where it is executed is identified as the remote machine or remote resource. These various grid computing mechanisms offered by HTCondor are distinguished by the software running on the remote resource.

When HTCondor 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 HTCondor-C. The job is submitted using the **grid universe**, and the **grid_type** is **condor**. HTCondor-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 HTCondor-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, HTCondor can still submit and manage jobs to be executed on remote resources. A **grid universe** job, with a **grid_type** of **gt2** or **gt5** calls on Globus software

to execute the job on a remote resource. Like HTCondor-C jobs, a network partition does not affect the execution of the job. The remote resource must have Globus software running.

HTCondor permits the temporary addition of a Globus-controlled resource to a local pool. This is called *glidein*. Globus software is utilized to execute HTCondor daemons on the remote resource. The remote resource appears to have joined the local HTCondor pool. A user submitting a job may then explicitly specify the remote resource as the execution site of a job.

Starting with HTCondor 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 HTCondor Pools with Flocking

Flocking is HTCondor'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 HTCondor pool. If a machine within HTCondor pool A can send jobs to be run on HTCondor 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 HTCondor's high availability mechanisms is not advised. See section 3.13.2 for a discussion of the issues.

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)
```

HOSTALLOW_NEGOTIATOR_SCHEDD provides an 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.
ALLOW_NEGOTIATOR_SCHEDD = $(CONDOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS), $(IP_ADDRESS)
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine. See section 3.8.7 on page 415 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 configuration variables 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 authorization:

```
ALLOW_WRITE_COLLECTOR = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_WRITE_STARTD    = $(ALLOW_WRITE), $(FLOCK_FROM)
ALLOW_READ_COLLECTOR  = $(ALLOW_READ), $(FLOCK_FROM)
ALLOW_READ_STARTD     = $(ALLOW_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.

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.

The submission of jobs other than standard universe jobs 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 enable file transfer mechanisms. These mechanisms are discussed in section 2.5.9 on page 32.

5.3 The Grid Universe

5.3.1 HTCondor-C, The condor Grid Type

HTCondor-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 HTCondor software and its configuration.

HTCondor-C is highly resistant to network disconnections and machine failures on both the submission and remote sides. An expected usage sets up Personal HTCondor on a laptop, submits some jobs that are sent to an HTCondor 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.

HTCondor-C scales gracefully when compared with HTCondor'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.

HTCondor-C Configuration

There are two aspects to configuration to enable the submission and execution of HTCondor-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.$(USERNAME)
C_GAHP_WORKER_THREAD_LOCK = /tmp/CGAHPWorkerLock.$(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 HTCondor-C. The configuration variable `C_GAHP_LOG` defines the location of the log that the HTCondor GAHP server writes. The log for the HTCondor 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. HTCondor-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
```

Other working authentication methods are GSI, SSL, KERBEROS, and FS.

HTCondor-C Job Submission

Job submission of HTCondor-C jobs is the same as for any HTCondor job. The **universe** is **grid**. The submit command **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 an HTCondor-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 HTCondor-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 ClassAd attributes

```
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
```

are used to invoke HTCondor's file transfer mechanism.

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 HTCondor's file transfer mechanism (see section 2.5.9 on page 32), the mechanism is applied based on the resulting job universe on the remote machine.

HTCondor-C Jobs Between Differing Platforms

HTCondor-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.

5.3.2 HTCondor-G, the gt2, and gt5 Grid Types

HTCondor-G is the name given to HTCondor 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.

HTCondor provides the same job management capabilities for HTCondor-G jobs as for other jobs. From HTCondor, a user may effectively submit jobs, manage jobs, and have jobs execute on widely distributed machines.

It may appear that HTCondor-G is a simple replacement for the Globus Toolkit's *globusrun* command. However, HTCondor-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, HTCondor-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.

HTCondor (and Globus) utilize the following protocols and terminology. The protocols allow HTCondor 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 HTCondor-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 HTCondor-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 HTCondor 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.

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 HTCondor 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 HTCondor, LSF, and PBS).

In its interaction with Globus software, HTCondor contains a GASS server, used to transfer the executable, `stdin`, `stdout`, and `stderr` to and from the remote job execution site. HTCondor 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. HTCondor detects and intelligently handles cases such as if the remote resource crashes.

There are now two different versions of the GRAM protocol in common usage: **gt2** and **gt5**. HTCondor supports both of them.

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) or GRAM2.

gt5 This latest GRAM protocol is an extension of GRAM2 that is intended to be more scalable and robust. It is usually referred to as GRAM5.

The gt2 Grid Type

HTCondor-G supports submitting jobs to remote resources running the Globus Toolkit's GRAM2 (or pre-WS GRAM) service. This flavor of GRAM is the most common. These HTCondor-G jobs are submitted the same as any other HTCondor 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 HTCondor, 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 HTCondor 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, HTCondor 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.

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 HTCondor 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 an HTCondor 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. HTCondor-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 HTCondor-G. If the location of the GASS server changes (for example, if HTCondor-G restarts), HTCondor-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 HTCondor-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 an HTCondor 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 HTCondor-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 HTCondor-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 gt5 Grid Type

The Globus GRAM5 protocol works the same as the gt2 grid type. Its implementation differs from gt2 in the following 3 items:

- The Grid Monitor is disabled.
- Globus job managers are not stopped and restarted.
- The configuration variable `GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE` is not applied (for gt5 jobs).

Normally, HTCondor will automatically detect whether a service is GRAM2 or GRAM5 and interact with it accordingly. It does not matter whether gt2 or gt5 is specified. Disable this detection by setting the configuration variable `GRAM_VERSION_DETECTION` to `False`. If disabled, each resource must be accurately identified as either gt2 or gt5 in the **grid_resource** submit command.

Credential Management with *MyProxy*

HTCondor-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://grid.ncsa.illinois.edu/myproxy/

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. HTCondor-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 HTCondor-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 an HTCondor-G job, specifying:

   MyProxy server name (host:port)

   MyProxy credential name (optional)

   MyProxy password
4. At the short-lived proxy expiration HTCondor-G talks to the *MyProxy* server to refresh the proxy.

HTCondor-G keeps track of the password to the *MyProxy* server for credential renewal. Although HTCondor-G tries to keep the password encrypted and secure, it is still possible (although highly unlikely) for the password to be intercepted from the HTCondor-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   = gt2 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 description file is not really secure, as it relies upon security provided by the file system. 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 login on that local machine could generate a simple shell script to watch for the password.

Currently, HTCondor-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 HTCondor-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

HTCondor's Grid Monitor is designed to improve the scalability of machines running the Globus Toolkit's GRAM2 gatekeeper. Normally, this service 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.

HTCondor'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, HTCondor 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, HTCondor 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.

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.

The Grid Monitor is enabled by default, and the configuration macro `GRID_MONITOR` identifies the location of the executable.

Limitations of HTCondor-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 are available when using **gt2**.
3. Limited platform availability. Windows support is not 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.

HTCondor 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. *condor_submit* looks in default locations for the proxy. The submit description file command **x509userproxy** may be 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 HTCondor constructs. The format this submit description file command is

```
nordugrid_rsl = (name=value)(name=value)
```

5.3.4 The unicore Grid Type

Unicore is a Java-based grid scheduling system. See http://www.unicore.eu/ for more information about Unicore.

HTCondor jobs may be submitted to Unicore resources using the **grid** universe. The **grid_resource** command specifies the name of the Unicore resource as follows:

```
grid_resource = unicore usite.example.com vsite
```

usite.example.com is the host name of the Unicore gateway machine to which the HTCondor job is to be submitted. **vsite** is the name of the Unicore virtual resource to which the HTCondor job is to be submitted.

Unicore 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 batch Grid Type (for PBS, LSF, SGE, and SLURM)

The **batch** grid type is used to submit to a local PBS, LSF, SGE, or SLURM system using the **grid** universe and the **grid_resource** command by placing a variant of the following into the submit description file.

```
grid_resource = batch pbs
```

The second argument on the right hand side will be one of `pbs`, `lsf`, `sge`, or `slurm`.

Any of these batch grid types requires two variables to be set in the HTCondor configuration file. BATCH_GAHP is the path to the GAHP server binary that is to be used to submit one of these batch jobs. GLITE_LOCATION is the path to the directory containing the GAHP's configuration file and auxiliary binaries. In the HTCondor distribution, these files are located in $(LIB)/glite. The batch GAHP's configuration file is in $(GLITE_LOCATION)/etc/batch_gahp.config. The batch GAHP's auxiliary binaries are to be in the directory $(GLITE_LOCATION)/bin. The HTCondor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
BATCH_GAHP     = $(GLITE_LOCATION)/bin/batch_gahp
```

The batch GAHP's configuration file has 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.

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.

The popular PBS (Portable Batch System) can be found at http://www.pbsworks.com/, and Torque is at (http://www.adaptivecomputing.com/products/open-source/torque/).

As an alternative to the submission details given above, HTCondor jobs may be 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
```

HTCondor jobs may be submitted to the Platform LSF batch system. Find the Platform product from the page http://www.platform.com/Products/ for more information about Platform LSF.

As an alternative to the submission details given above, HTCondor jobs may be 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 popular Grid Engine batch system (formerly known as Sun Grid Engine and abbreviated SGE) is available in two varieties: Oracle Grid Engine (http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html) and Univa Grid Engine (http://www.univa.com/?gclid=CLXg6-OEy6wCFWICQAodl0lm9Q).

As an alternative to the submission details given above, HTCondor jobs may be submitted to a local SGE system using the **grid** universe and adding the **grid_resource** command by placing into the submit description file:

```
grid_resource = sge
```

The *condor_qsub* command line tool will take PBS/SGE style batch files or command line arguments and submit the job to HTCondor instead. See the *condor_qsub* manual page at 11 for details.

5.3.6 The EC2 Grid Type

HTCondor jobs may be submitted to clouds supporting Amazon's Elastic Compute Cloud (EC2) interface. The EC2 interface permits on-line commercial services that provide the rental of computers by the hour to run computational applications. They run virtual machine images that have been uploaded to Amazon's online storage service (S3 or EBS). More information about Amazon's EC2 service is available at http://aws.amazon.com/ec2.

The **ec2** grid type uses the EC2 Query API, also called the EC2 REST API.

EC2 Job Submission

HTCondor jobs are submitted to an EC2 service with the **grid** universe, setting the **grid_resource** command to **ec2**, followed by the service's URL. For example, partial contents of the submit description file may be

```
grid_resource = ec2 https://ec2.amazonaws.com/
```

Since the job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It can be used to identify different jobs in the output of *condor_q*.

The VM image for the job must already reside in one of Amazon's storage service (S3 or EBS) and be registered with EC2. In the submit description file, provide the identifier for the image using **ec2_ami_id**.

This grid type requires access to user authentication information, in the form of path names to files containing the appropriate keys.

The **ec2** grid type has two different authentication methods. The first authentication method uses the EC2 API's built-in authentication. Specify the service with expected `http://` or `https://` URL, and set the EC2 access key and secret access key as follows:

```
ec2_access_key_id = /path/to/access.key
ec2_secret_access_key = /path/to/secret.key
```

The `euca3://` and `euca3s://` protocols must use this authentication method. These protocols exist to work correctly when the resources do not support the `InstanceInitiatedShutdownBehavior` parameter.

The second authentication method for the EC2 grid type is X.509. Specify the service with an `x509://` URL, even if the URL was given in another form. Use **ec2_access_key_id** to specify the path to the X.509 public key (certificate), which is not the same as the built-in authentication's access key. **ec2_secret_access_key** specifies the path to the X.509 private key, which is not the same as the built-in authentication's secret key. The following example illustrates the specification for X.509 authentication:

```
grid_resource = ec2 x509://service.example
ec2_access_key_id = /path/to/x.509/public.key
ec2_secret_access_key = /path/to/x.509/private.key
```

If using an X.509 proxy, specify the proxy in both places.

HTCondor can use the EC2 API to create an SSH key pair that allows secure log in to the virtual machine once it is running. If the command **ec2_keypair_file** is set in the submit description file, HTCondor will write an SSH private key into the indicated file. The key can be used to log into the virtual machine. Note that modification will also be needed of the firewall rules for the job to incoming SSH connections.

An EC2 service uses a firewall to restrict network access to the virtual machine instances it runs. Typically, no incoming connections are allowed. One can define sets of firewall rules and give them names. The EC2 API calls these security groups. If utilized, tell HTCondor what set of security groups should be applied to each VM using the **ec2_security_groups** submit description file command. If not provided, HTCondor uses the security group **default**. This command specifies security group names; to specify IDs, use **ec2_security_ids**. This may be necessary when specifying a Virtual Private Cloud (VPC) instance.

To run an instance in a VPC, set **ec2_vpc_subnet** to the the desired VPC's specification string. The instance's IP address may also be specified by setting **ec2_vpc_id**.

The EC2 API allows the choice of different hardware configurations for instances to run on. Select which configuration to use for the **ec2** grid type with the **ec2_instance_type** submit description file command. HTCondor provides no default.

Certain instance types provide additional block devices whose names must be mapped to kernel device names in order to be used. The **ec2_block_device_mapping** submit description file command allows specification of these maps. A map is a device name followed by a colon, followed by kernel name; maps are separated by a commas, and/or spaces. For example, to specify that the first ephemeral device should be `/dev/sdb` and the second `/dev/sdc`:

```
ec2_block_device_mapping = ephemeral0:/dev/sdb, ephemeral1:/dev/sdc
```

Each virtual machine instance can be given up to 16 KiB 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 HTCondor in one of two ways. First, the data can be provided directly in the submit description file using the **ec2_user_data** command. Second, the data can be stored in a file, and the file name is specified with the **ec2_user_data_file** submit description file command. This second option allows the use of binary data. If both options are used, the two blocks of data are concatenated, with the data from **ec2_user_data** occurring first. HTCondor performs the base64 encoding that EC2 expects on the data.

Amazon also offers an Identity and Access Management (IAM) service. To specify an IAM (instance) profile for an EC2 job, use submit commands **ec2_iam_profile_name** or **ec2_iam_profile_arn**.

Termination of EC2 Jobs

A protocol defines the shutdown procedure for jobs running as EC2 instances. The service is told to shut down the instance, and the service acknowledges. The service then advances the instance to a state in which the termination is imminent, but the job is given time to shut down gracefully.

Once this state is reached, some services other than Amazon cannot be relied upon to actually terminate the job.

Thus, HTCondor must check that the instance has terminated before removing the job from the queue. This avoids the possibility of HTCondor losing track of a job while it is still accumulating charges on the service.

HTCondor checks after a fixed time interval that the job actually has terminated. If the job has not terminated after a total of four checks, the job is placed on hold.

Using Spot Instances

EC2 jobs may also be submitted to clouds that support spot instances. A spot instance differs from a conventional, or dedicated, instance in two primary ways. First, the instance price varies according to demand. Second, the cloud provider may terminate the instance prematurely. To start a spot instance, the submitter specifies a bid, which represents the most the submitter is willing to pay per hour to run the VM. Within HTCondor, the submit command **ec2_spot_price** specifies this floating point value. For example, to bid 1.1 cents per hour on Amazon:

```
ec2_spot_price = 0.011
```

Note that the EC2 API does not specify how the cloud provider should interpret the bid. Empirically, Amazon uses fractional US dollars.

Other submission details for a spot instance are identical to those for a dedicated instance.

A spot instance will not necessarily begin immediately. Instead, it will begin as soon as the price drops below the bid. Thus, spot instance jobs may remain in the idle state for much longer than dedicated instance jobs, as they wait for the price to drop. Furthermore, if the price rises above the bid, the cloud service will terminate the instance.

More information about Amazon's spot instances is available at http://aws.amazon.com/ec2/spot-instances/.

Advanced Usage

Additional control of EC2 instances is available in the form of permitting the direct specification of instance creation parameters. To set an instance creation parameter, first list its name in the submit command **ec2_parameter_names**, a space or comma separated list. The parameter may need to be properly capitalized. Also tell HTCondor the parameter's value, by specifying it as a submit command whose name begins with **ec2_parameter_**; dots within the parameter name must be written as underscores in the submit command name.

For example, the submit description file commands to set parameter `IamInstanceProfile.Name` to value `ExampleProfile` are

```
ec2_parameter_names = IamInstanceProfile.Name
ec2_parameter_IamInstanceProfile_Name = ExampleProfile
```

EC2 Configuration Variables

The configuration variables EC2_GAHP and EC2_GAHP_LOG must be set, and by default are equal to $(SBIN)/ec2_gahp and /tmp/EC2GahpLog.$(USERNAME), respectively.

The configuration variable EC2_GAHP_DEBUG is optional and defaults to D_PID; we recommend you keep D_PID if you change the default, to disambiguate between the logs of different resources specified by the same user.

Communicating with an EC2 Service

The **ec2** grid type does not presently permit the explicit use of an HTTP proxy.

By default, HTCondor assumes that EC2 services are reliably available. If an attempt to contact a service during the normal course of operation fails, HTCondor makes a special attempt to contact the service. If this attempt fails, the service is marked as down, and normal operation for that service is suspended until a subsequent special attempt succeeds. The jobs using that service do not go on hold. To place jobs on hold when their service becomes unavailable, set configuration variable EC2_RESOURCE_TIMEOUT to the number of seconds to delay before placing the job on hold. The default value of -1 for this variable implements an infinite delay, such that the job is never placed on hold. When setting this value, consider the value of configuration variable GRIDMANAGER_RESOURCE_PROBE_INTERVAL, which sets the number of seconds that HTCondor will wait after each special contact attempt before trying again.

By default, the EC2 GAHP enforces a 100 millisecond interval between requests to the same service. This helps ensure reliable service. You may configure this interval with the configuration variable EC2_GAHP_RATE_LIMIT, which must be an integer number of milliseconds. Adjusting the interval may result in higher or lower throughput, depending on the service. Too short of an interval may trigger rate-limiting by the service; while HTCondor will react appropriately (by retrying with an exponential back-off), it may be more efficient to configure a longer interval.

Secure Communication with and EC2 Service

The specification of a service with an https://, an x509://, or an euca3s:// URL validates that service's certificate, checking that a trusted certificate authority (CA) signed it. Commercial EC2 service providers generally use certificates signed by widely-recognized CAs. These CAs will usually work without any additional configuration. For other providers, a specification of trusted CAs may be needed. Without, errors such as the following will be in the EC2 GAHP log:

```
06/13/13 15:16:16 curl_easy_perform() failed (60):
'Peer certificate cannot be authenticated with given CA certificates'.
```

Specify trusted CAs by including their certificates in a group of trusted CAs either in an on disk directory or in a single file. Either of these alternatives may contain multiple certificates. Which is used will vary from system to system, depending on the system's SSL implementation. HTCondor uses *libcurl*; information about the *libcurl* specification of trusted CAs is available at

http://curl.haxx.se/libcurl/c/curl_easy_setopt.html

Versions of HTCondor with standard universe support ship with their own *libcurl*, which will be linked against *OpenSSL*.

The behavior when specifying both a directory and a file is undefined, although the EC2 GAHP allows it.

The EC2 GAHP will set the CA file to whichever variable it finds first, checking these in the following order:

1. The environment variable `X509_CERT_FILE`, set when the *condor_master* starts up.
2. The HTCondor configuration variable `SOAP_SSL_CA_FILE`.

The EC2 GAHP supplies no default value, if it does not find a CA file.

The EC2 GAHP will set the CA directory given whichever of these variables it finds first, checking in the following order:

1. The HTCondor configuration variable `GSI_DAEMON_TRUSTED_CA_DIR`.
2. The environment variable `X509_CERT_DIR`, set when the *condor_master* starts up.
3. The HTCondor configuration variable `SOAP_SSL_CA_DIR`.

The EC2 GAHP supplies no default value, if it does not find a CA directory.

EC2 GAHP Statistics

The EC2 GAHP tracks, and reports in the corresponding grid resource ad, statistics related to resource's rate limit.

NumRequests: The total number of requests made by HTCondor to this resource.

NumDistinctRequests: The number of distinct requests made by HTCondor to this resource. The difference between this and NumRequests is the total number of retries. Retries are not unusual.

NumRequestsExceedingLimit: The number of requests which exceeded the service's rate limit. Each such request will cause a retry, unless the maximum number of retries is exceeded, or if the retries have already taken so long that the signature on the original request has expired.

NumExpiredSignatures: The number of requests which the EC2 GAHP did not even attempt to send to the service because signature expired. Signatures should not, generally, expire; a request's retries will usually – eventually – succeed.

5.3.7 The GCE Grid Type

HTCondor jobs may be submitted to the Google Compute Engine (GCE) cloud service. GCE is an on-line commercial service that provides the rental of computers by the hour to run computational applications. Its runs virtual machine images that have been uploaded to Google's servers. More information about Google Compute Engine is available at http://cloud.google.com/Compute.

GCE Job Submission

HTCondor jobs are submitted to the GCE service with the **grid** universe, setting the **grid_resource** command to **gce**, followed by the service's URL, your GCE project, and the desired GCE zone to be used. The submit description file command will be similar to:

```
grid_resource = gce https://www.googleapis.com/compute/v1 my_proj us-central1-a
```

Since the HTCondor job is a virtual machine image, most of the submit description file commands specifying input or output files are not applicable. The **executable** command is still required, but its value is ignored. It identifies different jobs in the output of *condor_q*.

The VM image for the job must already reside in Google's Cloud Storage service and be registered with GCE. In the submit description file, provide the identifier for the image using the **gce_image** command.

This grid type requires granting HTCondor permission to use your Google account. The easiest way to do this is to use the *gcloud* command-line tool distributed by Google. Find *gcloud* and documentation for it at https://cloud.google.com/compute/docs/gcloud-compute/. After installation of *gcloud*, run *gcloud auth login* and follow its directions. Once done with that step, the tool will write authorization credentials to the file `.config/gcloud/credentials` under your HOME directory.

Given an authorization file, specify its location in the submit description file using the **gce_auth_file** command, as in the example:

```
gce_auth_file = /path/to/auth-file
```

GCE allows the choice of different hardware configurations for instances to run on. Select which configuration to use for the **gce** grid type with the **gce_machine_type** submit description file command. HTCondor provides no default.

Each virtual machine instance can be given a unique set of metadata, which consists of name/value pairs, similar to the environment variables of regular jobs. The instance can query its metadata via 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 HTCondor in one of two ways. First, the data can be provided directly in the submit description file using the **gce_metadata** command. The value should be a comma-separated list of name=value settings, as the example:

```
gce_metadata = setting1=foo,setting2=bar
```

Second, the data can be stored in a file, and the file name is specified with the **gce_metadata_file** submit description file command. This second option allows a wider range of characters to be used in the metadata values. Each name=value pair should be on its own line. No white space is removed from the lines, except for the newline that separates entries.

Both options can be used at the same time, but do not use the same metadata name in both places.

HTCondor sets the following elements when describing the instance to the GCE server: **machineType**, **name**, **scheduling**, **disks**, **metadata**, and **networkInterfaces**. You can provide additional elements to be included in the instance description as a block of JSON. Write the additional elements to a file, and specify the filename in your submit

file with the **gce_json_file** command. The contents of the file are inserted into HTCondor's JSON description of the instance, between a comma and the closing brace.

Here's a sample JSON file that sets two additional elements:

```
"canIpForward": True,
"description": "My first instance"
```

GCE Configuration Variables

The following configuration parameters are specific to the **gce** grid type. The values listed here are the defaults. Different values may be specified in the HTCondor configuration files.

```
GCE_GAHP     = $(SBIN)/gce_gahp
GCE_GAHP_LOG = /tmp/GceGahpLog.$(USERNAME)
```

5.3.8 The cream Grid Type

CREAM is a job submission interface being developed at INFN for the gLite software stack. The CREAM homepage is http://grid.pd.infn.it/cream/. The protocol is based on web services.

The protocol requires an X.509 proxy for the job, so the submit description file command **x509userproxy** will be used.

A CREAM resource specification is of the form:

```
grid_resource = cream <web-services-address> <batch-system> <queue-name>
```

The `<web-services-address>` appears the same for most servers, differing only in the host name, as

```
<machinename[:port]>/ce-cream/services/CREAM2
```

Future versions of HTCondor may require only the host name, filling in other aspects of the web service for the user.

The `<batch-system>` is the name of the batch system that sits behind the CREAM server, into which it submits the jobs. Normal values are `pbs`, `lsf`, and `condor`.

The `<queue-name>` identifies which queue within the batch system should be used. Values for this will vary by site, with no typical values.

A full example for the specification of a CREAM **grid_resource** is

```
grid_resource = cream https://cream-12.pd.infn.it:8443/ce-cream/services/CREAM2
   pbs cream_1
```

This is a single line within the submit description file, although it is shown here on two lines for formatting reasons.

CREAM uses ClassAd syntax to describe jobs, although the attributes used are different than those for HTCondor. The submit description file command **cream_attributes** adds additional attributes to the CREAM-style job ClassAd that HTCondor constructs. The format for this submit description file command is

```
cream_attributes = name=value;name=value
```

5.3.9 The BOINC Grid Type

HTCondor jobs may be submitted to BOINC (Berkeley Open Infrastructure for Network Computing) servers. BOINC is a software system for volunteer computing. More information about BOINC is available at http://boinc.berkeley.edu/.

BOINC Job Submission

HTCondor jobs are submitted to a BOINC service with the **grid** universe, setting the **grid_resource** command to **boinc**, followed by the service's URL.

To use this grid type, you must have an account on the BOINC server that is authorized to submit jobs. Provide the authenticator string for that account for HTCondor to use. Write the authenticator string in a file and specify its location in the submit description file using the **boinc_authenticator_file** command, as in the example:

```
boinc_authenticator_file = /path/to/auth-file
```

Before submitting BOINC jobs, register the application with the BOINC server. This includes describing the application's resource requirements and input and output files, and placing application files on the server. This is a manual process that is done on the BOINC server. See the BOINC documentation for details.

In the submit description file, the **executable** command gives the registered name of the application on the BOINC server. Input and output files can be described as in the vanilla universe, but the file names must match the application description on the BOINC server. If **transfer_output_files** is omitted, then all output files are transferred.

BOINC Configuration Variables

The following configuration variable is specific to the **boinc** grid type. The value listed here is the default. A different value may be specified in the HTCondor configuration files.

```
BOINC_GAHP = $(SBIN)/boinc_gahp
```

5.3.10 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, HTCondor 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 HTCondor'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 HTCondor by advertising a ClassAd. This ClassAd specifies all necessary attributes, such that HTCondor 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 HTCondor 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 HTCondor

Any grid resource that wishes to be matched by HTCondor with a job must advertise itself to HTCondor 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 HTCondor: 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 1005.

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 HTCondor, 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 HTCondor'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
WantAdRevaluate = True
CurMatches     = 1
```

In this example, the two attributes `WantAdRevaluate` and `CurMatches` appear, and the `Requirements` expression has changed.

`WantAdRevaluate` 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 HTCondor 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, HTCondor 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 HTCondor does not need to be installed. The required argument for the *condor_advertise* command is *UPDATE_STARTD_AD*.

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 HTCondor will attempt to match it and re-run it at another site. HTCondor 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 HTCondor 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 control grid universe job resubmissions and rematching. See the definitions of the submit description file commands **globus_resubmit** and **globus_rematch** at page 919 and page 919. 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 987 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))" ) && \
   ((time() - 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 HTCondor's security options, documented in Section 3.8.

5.4 The HTCondor Job Router

The HTCondor Job Router is an add-on to the *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 for the task of sending excess jobs to one or more remote grid sites. The Job Router can transform the jobs such as vanilla universe jobs into grid universe jobs that use any of the grid types supported by HTCondor. 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 HTCondor pool, and any flocked HTCondor 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.

5.4.1 Routing Mechanism

The *condor_job_router* daemon and configuration determine a policy for which jobs may be transformed and sent to grid sites. By default, a job is transformed into a grid universe job by making a copy of the original job ClassAd, and modifying some attributes in this copy 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, because it is tied to the routed copy. The original job 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.

Although the default mode routes vanilla universe jobs to grid universe jobs, the routing rules may be configured to do some other transformation of the job. It is also possible to edit the job in place rather than creating a new transformed version of the job.

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 in the New ClassAd language, as currently used by HTCondor internally.

A good place to learn about the syntax of New ClassAds is the Informal Language Description in the C++ ClassAds tutorial: http://htcondor.org/classad/c++tut.html. 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 an HTCondor 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 HTCondor configuration file.

5.4.2 Job Submission with Job Routing Capability

If Job Routing is set up, then the following items ought to be considered for 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 HTCondor'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 most types of 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 all grid universe jobs, except for HTCondor-C, only transfer back the **output** file, as well as those explicitly listed with **transfer_output_files**. Therefore, when routing jobs to grid universes other than HTCondor-C, it is important to 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 job event 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.ulog
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 HTCondor 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    Runs
                       Completed Aborted
-------------------------------------------------------
Site1              10       2     0
Site2               8       2     1
Site3              40       6     0
-------------------------------------------------------
TOTAL              58      10     1
```

5.4.3 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 3.5.19. One route is an HTCondor site accessed via the Globus gt2 protocol. A second route is a PBS site, also accessed via Globus gt2. The third site is an HTCondor site accessed by HTCondor-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 HTCondor (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 CA, if needed.

Note that, as of version 8.5.6, the configuration language supports multi-line values, as shown in the example below (see section 3.3.5 for more details).

```
# These settings become the default settings for all routes
JOB_ROUTER_DEFAULTS @=jrd
  [
    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 && (time() - QDate) > 3600*6);
    delete_WantJobRouter = true;
    set_requirements = true;
  ]
  @jrd

# 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 @=jre
  [ 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;
  ]
  @jre
```

```
# Reminder: you must restart HTCondor 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
```

5.4.4 Routing Table Entry ClassAd Attributes

The conversion of a job to a routed copy may require the job ClassAd to be modified. The Routing Table specifies attributes of the different possible routes and it may specify specific modifications that should be made to the job when it is sent along a specific route. In addition to this mechanism for transforming the job, external programs may be invoked to transform the job. For more information, see section 4.4.2.

The following attributes and instructions for modifying job attributes may appear in a Routing Table entry.

GridResource Specifies the value for the `GridResource` attribute that will be inserted into the routed copy of the job's 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 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 site 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 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 `target.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 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 `target.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 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 copy's 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. The expression is evaluated, and the resulting value sets the value of the routed copy's job ClassAd attribute `<ATTR>`. Use this attribute to set a custom or local value, especially for modifying an attribute which may have been already specified in a default routing table.

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.

EditJobInPlace A boolean expression that, when `True`, causes the original job to be transformed in place rather than creating a new transformed version (a routed copy) of the job. In this mode, the Job Router Hook `<Keyword>_HOOK_TRANSLATE_JOB` and transformation rules in the routing table are applied during the job transformation. The routing table attribute `GridResource` is ignored, and there is no default transformation of

the job from a vanilla job to a grid universe job as there is otherwise. Once transformed, the job is still a candidate for matching routing rules, so it is up to the routing logic to control whether the job may be transformed multiple times or not. For example, to transform the job only once, an attribute could be set in the job ClassAd to prevent it from matching the same routing rule in the future. To transform the job multiple times with limited frequency, a timestamp could be inserted into the job ClassAd marking the time of the last transformation, and the routing entry could require that this timestamp either be undefined or older than some limit.

5.4.5 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 an HTCondor 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.

Chapter 6

Application Programming Interfaces (APIs)

There are several ways of interacting with the HTCondor system. Depending on your application and resources, the interfaces to HTCondor listed below may be useful for your installation. If you have developed an interface to HTCondor, please consider sharing it with the HTCondor community.

6.1 Web Service

HTCondor's Web Service (WS) API provides a way for application developers to interact with HTCondor, without needing to utilize HTCondor's command-line tools. In keeping with the HTCondor philosophy of reliability and fault-tolerance, this API is designed to provide a simple and powerful way to interact with HTCondor. HTCondor daemons understand and implement the SOAP (Simple Object Access Protocol) XML API to provide a web service interface for HTCondor job submission and management.

To deal with the issues of reliability and fault-tolerance, a two-phase commit mechanism to 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.

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 Implementation Details

HTCondor 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 HTCondor implements is included with the HTCondor 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.

HTCondor must be configured to enable responses to SOAP calls. Please see section 3.5.28 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 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
```

HTCondor's security configuration must be set up such that access is authorized for the SOAP client. See Section 3.8.7 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.

6.1.5 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.

6.1.6 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.

6.1.7 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.

6.1.8 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.

6.1.9 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.

6.1.10 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.

6.1.11 Methods for Version Information

`getVersionString` A prototype is

```
StatusAndString getVersionString();
```

Return Value Returns the HTCondor version as a string.

`getPlatformString` A prototype is

```
StatusAndString getPlatformString();
```

Return Value Returns the platform information HTCondor is running on as string.

6.1.12 Common Data Structures

Many methods return a status. Table 6.1 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 6.1: StatusCode definitions

6.2 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 HTCondor implementation of a portion of the API allows programs (applications) to use the library functions provided to submit, monitor and control HTCondor jobs.

See the DRMAA site (http://www.drmaa.org) to find the API specification for DRMA 1.0 for further details on the API.

6.2.1 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 $(LIB) directory. Its header file is $(INCLUDE)/drmaa.h, and file $(INCLUDE)/README 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 11 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 HTCondor 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);
```

6.3 The HTCondor User and Job Log Reader API

HTCondor has the ability to log an HTCondor job's significant events during its lifetime. This is enabled in the job's submit description file with the **Log** command.

This section describes the API defined by the C++ `ReadUserLog` class, which provides a programming interface for applications to read and parse events, polling for events, and saving and restoring reader state.

6.3.1 Constants and Enumerated Types

The following define enumerated types useful to the API.

- `ULogEventOutcome` (defined in `condor_event.h`):
  - `ULOG_OK`: Event is valid
  - `ULOG_NO_EVENT`: No event occurred (like EOF)
  - `ULOG_RD_ERROR`: Error reading log file
  - `ULOG_MISSED_EVENT`: Missed event
  - `ULOG_UNK_ERROR`: Unknown Error
- `ReadUserLog::FileStatus`
  - `LOG_STATUS_ERROR`: An error was encountered
  - `LOG_STATUS_NOCHANGE`: No change in file size
  - `LOG_STATUS_GROWN`: File has grown
  - `LOG_STATUS_SHRUNK`: File has shrunk

6.3.2 Constructors and Destructors

All `ReadUserLog` constructors invoke one of the `initialize()` methods. Since C++ constructors cannot return errors, an application using any but the default constructor should call `isInitialized()` to verify that the object initialized correctly, and for example, had permissions to open required files.

Note that because the constructors cannot return status information, most of these constructors will be eliminated in the future. All constructors, except for the default constructor with no parameters, will be removed. The application will need to call the appropriate `initialize()` method.

- `ReadUserLog::ReadUserLog(bool isEventLog)`
  Synopsis: Constructor default
  Returns: None
  Constructor parameters:
  - `bool isEventLog` (*Optional with default* = `false`)
    If `true`, the `ReadUserLog` object is initialized to read the schedd-wide event log.
    NOTE: If `isEventLog` is `true`, the initialization may silently fail, so the value of `ReadUserLog::isInitialized` should be checked to verify that the initialization was successful.
    NOTE: The `isEventLog` parameter will be removed in the future.
- `ReadUserLog::ReadUserLog(FILE *fp, bool is_xml, bool enable_close)`
  Synopsis: Constructor of a limited functionality reader: no rotation handling, no locking
  Returns: None
  Constructor parameters:

  - `FILE * fp`
    File pointer to the previously opened log file to read.
  - `bool is_xml`
    If `true`, the file is treated as XML; otherwise, it will be read as an old style file.
  - `bool enable_close` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only.

  NOTE: The `ReadUserLog::isInitialized` method should be invoked to verify that this constructor was initialized successfully.
  NOTE: This constructor will be removed in the future.

- `ReadUserLog::ReadUserLog(const char *filename, bool read_only)`
  Synopsis: Constructor to read a specific log file
  Returns: None
  Constructor parameters:

  - `const char * filename`
    Path to the log file to read
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

  NOTE: This constructor will be removed in the future.

- `ReadUserLog::ReadUserLog(const FileState &state, bool read_only)`
  Synopsis: Constructor to continue from a persisted reader state
  Returns: None
  Constructor parameters:

  - `const FileState & state`
    Reference to the persisted state to restore from
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

  NOTE: The `ReadUserLog::isInitialized` method should be invoked to verify that this constructor was initialized successfully.
  NOTE: This constructor will be removed in the future.

- `ReadUserLog::~ReadUserLog(void)`
  Synopsis: Destructor
  Returns: None
  Destructor parameters:

  - None.

6.3.3 Initializers

These methods are used to perform the initialization of the `ReadUserLog` objects. These initializers are used by all constructors that do real work. Applications should never use those constructors, should use the default constructor, and should instead use one of these initializer methods.

All of these functions will return `false` if there are problems such as being unable to open the log file, or `true` if successful.

- `bool ReadUserLog::initialize(void)`
  Synopsis: Initialize to read the EventLog file.
  NOTE: This method will likely be eliminated in the future, and this functionality will be moved to a new `ReadEventLog` class.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:
  - None.
- `bool ReadUserLog::initialize(const char *filename, bool handle_rotation, bool check_for_rotated, bool read_only)`
  Synopsis: Initialize to read a specific log file.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:
  - `const char * filename`
    Path to the log file to read
  - `bool handle_rotation` (*Optional with default* = `false`)
    If `true`, enable the reader to handle rotating log files, which is only useful for global user logs
  - `bool check_for_rotated` (*Optional with default* = `false`)
    If `true`, try to open the rotated files (with file names appended with `.old` or `.1`, `.2`, ...) first.
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.
- `bool ReadUserLog::initialize(const char *filename, int max_rotation, bool check_for_rotated, bool read_only)`
  Synopsis: Initialize to read a specific log file.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:
  - `const char * filename`
    Path to the log file to read
  - `int max_rotation`
    Limits what previously rotated files will be considered by the number given in the file name suffix. A value of 0 disables looking for rotated files. A value of 1 limits the rotated file to be that with the file name suffix of `.old`. As only event logs are rotated, this parameter is only useful for event logs.

  - `bool check_for_rotated` (*Optional with default* = `false`)
    If `true`, try to open the rotated files (with file names appended with `.old` or `.1`, `.2`, ...) first.
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

- `bool ReadUserLog::initialize(const FileState &state, bool read_only)`
  Synopsis: Initialize to continue from a persisted reader state.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:
  - `const FileState & state`
    Reference to the persisted state to restore from
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

- `bool ReadUserLog::initialize(const FileState &state, int max_rotation, bool read_only)`
  Synopsis: Initialize to continue from a persisted reader state and set the rotation parameters.
  Returns: `bool`; `true`: success, `false`: failed
  Method parameters:
  - `const FileState & state`
    Reference to the persisted state to restore from
  - `int max_rotation`
    Limits what previously rotated files will be considered by the number given in the file name suffix. A value of 0 disables looking for rotated files. A value of 1 limits the rotated file to be that with the file name suffix of `.old`. As only event logs are rotated, this parameter is only useful for event logs.
  - `bool read_only` (*Optional with default* = `false`)
    If `true`, the reader will open the file read-only and disable locking.

6.3.4 Primary Methods

- `ULogEventOutcome ReadUserLog::readEvent(ULogEvent *& event)`
  Synopsis: Read the next event from the log file.
  Returns: `ULogEventOutcome`; Outcome of the log read attempt. `ULogEventOutcome` is an enumerated type.
  Method parameters:
  - `ULogEvent *& event`
    Pointer to an `ULogEvent` that is allocated by this call to `ReadUserLog::readEvent`. If no event is allocated, this pointer is set to `NULL`. Otherwise the event needs to be `delete()ed` by the application.

- `bool ReadUserLog::synchronize(void)`
  Synopsis: Synchronize the log file if the last event read was an error. This safe guard function should be called if there is some error reading an event, but there are events after it in the file. It will skip over the bad event, meaning it will read up to and including the event separator, so that the rest of the events can be read.

Returns: `bool`; `true`: success, `false`: failed
Method parameters:

- None.

6.3.5 Accessors

- `ReadUserLog::FileStatus ReadUserLog::CheckFileStatus(void)`
  Synopsis: Check the status of the file, and whether it has grown, shrunk, etc.
  Returns: `ReadUserLog::FileStatus`; the status of the log file, an enumerated type.
  Method parameters:
  - None.
- `ReadUserLog::FileStatus ReadUserLog::CheckFileStatus(bool &is_empty)`
  Synopsis: Check the status of the file, and whether it has grown, shrunk, etc.
  Returns: `ReadUserLog::FileStatus`; the status of the log file, an enumerated type.
  Method parameters:
  - `bool & is_empty`
    Set to `true` if the file is empty, `false` otherwise.

6.3.6 Methods for saving and restoring persistent reader state

The `ReadUserLog::FileState` structure is used to save and restore the state of the `ReadUserLog` state for persistence. The application should always use `InitFileState()` to initialize this structure.

All of these methods take a reference to a state buffer as their only parameter.

All of these methods return `true` upon success.

6.3.7 Save state to persistent storage

To save the state, do something like this:

```
ReadUserLog               reader;
ReadUserLog::FileState    statebuf;

status = ReadUserLog::InitFileState( statebuf );

status = reader.GetFileState( statebuf );
write( fd, statebuf.buf, statebuf.size );
...
status = reader.GetFileState( statebuf );
write( fd, statebuf.buf, statebuf.size );
...

status = UninitFileState( statebuf );
```

6.3.8 Restore state from persistent storage

To restore the state, do something like this:

```
ReadUserLog::FileState     statebuf;
status = ReadUserLog::InitFileState( statebuf );

read( fd, statebuf.buf, statebuf.size );

ReadUserLog                reader;
status = reader.initialize( statebuf );

status = UninitFileState( statebuf );
....
```

6.3.9 API Reference

- `static bool ReadUserLog::InitFileState(ReadUserLog::FileState &state)`
  Synopsis: Initialize a file state buffer
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `ReadUserLog::FileState & state`
    The file state buffer to initialize.
- `static bool ReadUserLog::UninitFileState(ReadUserLog::FileState &state)`
  Synopsis: Clean up a file state buffer and free allocated memory
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `ReadUserLog::FileState & state`
    The file state buffer to un-initialize.
- `bool ReadUserLog::GetFileState(ReadUserLog::FileState &state) const`
  Synopsis: Get the current state to persist it or save it off to disk
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `ReadUserLog::FileState & state`
    The file state buffer to read the state into.
- `bool ReadUserLog::SetFileState(const ReadUserLog::FileState &state)`
  Synopsis: Use this method to set the current state, after restoring it.
  NOTE: The state buffer is *NOT* automatically updated; a call *MUST* be made to the `GetFileState()` method each time before persisting the buffer to disk, or however else is chosen to persist its contents.
  Returns: `bool`; `true` if successful, `false` otherwise
  Method parameters:
  - `const ReadUserLog::FileState & state`
    The file state buffer to restore from.

6.3.10 Access to the persistent state data

If the application needs access to the data elements in a persistent state, it should instantiate a `ReadUserLogStateAccess` object.

- Constructors / Destructors
  - `ReadUserLogStateAccess::ReadUserLogStateAccess(const ReadUserLog::FileState &state)`
    Synopsis: Constructor default
    Returns: None
    Constructor parameters:
    * `const ReadUserLog::FileState & state`
      Reference to the persistent state data to initialize from.
  - `ReadUserLogStateAccess::~ReadUserLogStateAccess(void)`
    Synopsis: Destructor
    Returns: None
    Destructor parameters:
    * None.
- Accessor Methods
  - `bool ReadUserLogFileState::isInitialized(void) const`
    Synopsis: Checks if the buffer initialized
    Returns: `bool`; `true` if successfully initialized, `false` otherwise
    Method parameters:
    * None.
  - `bool ReadUserLogFileState::isValid(void) const`
    Synopsis: Checks if the buffer is valid for use by `ReadUserLog::initialize()`
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:
    * None.
  - `bool ReadUserLogFileState::getFileOffset(unsigned long &pos) const`
    Synopsis: Get position within individual file.
    NOTE: Can return an error if the result is too large to be stored in a `long`.
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:
    * `unsigned long & pos`
      Byte position within the current log file
  - `bool ReadUserLogFileState::getFileEventNum(unsigned long &num) const`
    Synopsis: Get event number in individual file.
    NOTE: Can return an error if the result is too large to be stored in a `long`.
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:

    * `unsigned long & num`
      Event number of the current event in the current log file

  - `bool ReadUserLogFileState::getLogPosition(unsigned long &pos) const`
    Synopsis: Position of the start of the current file in overall log.
    NOTE: Can return an error if the result is too large to be stored in a `long`.
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:

    * `unsigned long & pos`
      Byte offset of the start of the current file in the overall logical log stream.

  - `bool ReadUserLogFileState::getEventNumber(unsigned long &num) const`
    Synopsis: Get the event number of the first event in the current file
    NOTE: Can return an error if the result is too large to be stored in a `long`.
    Returns: bool; `true` if successful, `false` otherwise
    Method parameters:

    * `unsigned long & num`
      This is the absolute event number of the first event in the current file in the overall logical log stream.

  - `bool ReadUserLogFileState::getUniqId(char *buf, int size) const`
    Synopsis: Get the unique ID of the associated state file.
    Returns: bool; `true` if successful, `false` otherwise
    Method parameters:

    * `char * buf`
      Buffer to fill with the unique ID of the current file.
    * `int size`
      Size in bytes of `buf`.
      This is to prevent `ReadUserLogFileState::getUniqId` from writing past the end of `buf`.

  - `bool ReadUserLogFileState::getSequenceNumber(int &seqno) const`
    Synopsis: Get the sequence number of the associated state file.
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:

    * `int & seqno`
      Sequence number of the current file

- Comparison Methods

  - `bool ReadUserLogFileState::getFileOffsetDiff(const ReadUserLogStateAccess &other, unsigned long &pos) const`
    Synopsis: Get the position difference of two states given by `this` and `other`.
    NOTE: Can return an error if the result is too large to be stored in a `long`.
    Returns: `bool`; `true` if successful, `false` otherwise
    Method parameters:

    * `const ReadUserLogStateAccess & other`
      Reference to the state to compare to.
    * `long & diff`
      Difference in the positions

- `bool ReadUserLogFileState::getFileEventNumDiff(const ReadUserLogStateAccess &other, long &diff) const`
  Synopsis: Get event number in individual file.
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: bool; `true` if successful, `false` otherwise
  Method parameters:
  * `const ReadUserLogStateAccess & other`
    Reference to the state to compare to.
  * `long & diff`
    Event number of the current event in the current log file
- `bool ReadUserLogFileState::getLogPosition(const ReadUserLogStateAccess &other, long &diff) const`
  Synopsis: Get the position difference of two states given by `this` and `other`.
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: bool; `true` if successful, `false` otherwise
  Method parameters:
  * `const ReadUserLogStateAccess & other`
    Reference to the state to compare to.
  * `long & diff`
    Difference between the byte offset of the start of the current file in the overall logical log stream and that of `other`.
- `bool ReadUserLogFileState::getEventNumber(const ReadUserLogStateAccess &other, long &diff) const`
  Synopsis: Get the difference between the event number of the first event in two state buffers (this - other).
  NOTE: Can return an error if the result is too large to be stored in a `long`.
  Returns: bool; `true` if successful, `false` otherwise
  Method parameters:
  * `const ReadUserLogStateAccess & other`
    Reference to the state to compare to.
  * `long & diff`
    Difference between the absolute event number of the first event in the current file in the overall logical log stream and that of `other`.

6.3.11 Future persistence API

The `ReadUserLog::FileState` will likely be replaced with a new C++ `ReadUserLog::NewFileState`, or a similarly named class that will self initialize.

Additionally, the functionality of `ReadUserLogStateAccess` will be integrated into this class.

6.4 Chirp

Chirp is a wire protocol and API that supports communication between a running job and a Chirp server. The HTCondor system provides a Chirp server running in the *condor_starter* that allows a job to

1. perform file I/O to and from the submit machine
2. update an attribute in its own job ClassAd
3. append the job event log file

This service is off by default; it may be enabled by placing in the submit description file:

```
+WantIOProxy = True
```

This places the needed attribute into the job ClassAd.

The Chirp protocol is fully documented at http://www3.nd.edu/ ccl/software/chirp.

To provide easier access to this wire protocol, the *condor_chirp* command line tool is shipped with HTCondor. This tool provides full access to the Chirp commands.

6.5 The Command Line Interface

While the usual HTCondor command line tools are often not thought of as an API, they are frequently the best choice for a programmatic interface to the system. They are the most complete, tested and debugged way to work with the system. The major down side to running the tools is that spawning an executable may be relatively slow; many applications do not need an extreme level of performance, making use of the command line tools acceptable. Even some of the HTCondor tools themselves work this way. For example, when *condor_dagman* needs to submit a job, it invokes the *condor_submit* program, just as an interactive user would.

6.6 The HTCondor Perl Module

The HTCondor Perl module facilitates automatic submitting and monitoring of HTCondor jobs, along with automated administration of HTCondor. The most common use of this module is the monitoring of HTCondor jobs. The HTCondor Perl module can be used as a meta scheduler for the submission of HTCondor jobs.

The HTCondor 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.

6.6.1 Subroutines

Submit(submit_description_file) This subroutine takes the action of submitting a job to HTCondor. 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(command_file) This subroutine submits a job to HTCondor 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(DAG_file, DAGMan_args) 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(DAG_file, DAGMan_args) This subroutine submits a *condor_dagman* to HTCondor 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()`.

6.6.2 Examples

The following is an example that uses the HTCondor 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 HTCondor jobs. It submits a second job if the first job successfully completes.

```
#!/s/std/bin/perl

# tell Perl where to find the HTCondor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the HTCondor 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 HTCondor 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 HTCondor 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 HTCondor library
```

```
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the HTCondor 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();
```

6.7 Python Bindings

The Python module provides bindings to the client-side APIs for HTCondor and the ClassAd language.

These Python bindings depend on loading the HTCondor shared libraries; this means the same code is used here as the HTCondor client tools. It is more efficient in terms of memory and CPU to utilize these bindings than to parse the output of the HTCondor client tools when writing applications in Python.

6.7.1 `htcondor` Module

The `htcondor` module provides a client interface to the various HTCondor daemons. It tries to provide functionality similar to the HTCondor command line tools.

`htcondor` module functions:

`platform( )` Returns the platform of HTCondor this module is running on.
`version( )` Returns the version of HTCondor this module is linked against.
`reload_config( )` Reload the HTCondor configuration from disk.
`send_command( ad, (DaemonCommands)dc, (str)target = None)` Send a command to an HTCondor daemon specified by a location ClassAd. `ad` is a ClassAd specifying the location of the daemon; typically, found by using `Collector.locate(...)`. `dc` is a command type; must be a member of the enum `DaemonCommands`. `target` is an optional parameter, representing an additional command to send to a daemon. Some commands require additional arguments; for example, sending `DaemonOff` to a *condor_master* requires one to specify which subsystem to turn off.
`read_events( file_obj, is_xml = True )` Read and parse an HTCondor event log file. Returns a Python iterator of ClassAds. Parameter `file_obj` is a file object corresponding to an HTCondor event log. The optional parameter `is_xml` specifies whether the event log is XML-formatted.
`send_alive( ad, pid, timeout )` Send a keep alive message to an HTCondor daemon. Parameter `ad` is a ClassAd specifying the location of the daemon. This ClassAd is typically found by using `Collector.locate(...)`. Parameter `pid` is the process identifier for the keep alive. The default value of `None` uses the value from `os.getpid()`. Parameter `timeout` is the number of seconds that this keep alive is valid. If a new keep alive is not received by the *condor_master* in time, then the process will be terminated. The default value is controlled by configuration variable `NOT_RESPONDING_TIMEOUT`.
`set_subsystem( name, type = Auto )` Set the subsystem name for the object. Parameter `name` is the subsystem name. Parameter `type` is the HTCondor daemon type, taken from the `SubsystemType` enum. The default value of `Auto` infers the type from the `name` parameter.
`lock( file_obj, lock_type )` Take a lock on a file object using the HTCondor locking protocol, which is distinct from typical POSIX locks. Returns a context manager object; the lock is released as this context manager object is destroyed. Parameter `file_obj` is a file object corresponding to the file which should be locked. Parameter `lock_type` specifies the string `"ReadLock"` if the lock should be for reads or `"WriteLock"` if the lock should be for writes.
`enable_debug( )` Enable debugging output from HTCondor, where output is sent to `stderr`. The logging level is controlled by `TOOL_DEBUG`.
`enable_log( )` Enable debugging output from HTCondor, where output is sent to a file. The log level is controlled by `TOOL_DEBUG`, and the file used is controlled by `TOOL_LOG`.
`log( level, msg )` Log a message to the HTCondor logging subsystem. Parameter `level` is the Log category and formatting indicator. Use the `LogLevel` enum to get list of attributes that may be OR'd together. Parameter `msg` is a String message to log.
`poll( active_queries )`

The module object, `param`, is a dictionary-like object providing access to the configuration variables in the current HTCondor configuration.

The `Schedd` class:

`__init__( classad )` Create an instance of the `Schedd` class. Optional parameter `classad` describes the location of the remote *condor_schedd* daemon. If the parameter is omitted, the local *condor_schedd* daemon is used.
`transaction( flags = 0, continue_txn = False )` Start a transaction with the *condor_schedd*. Returns a transaction context manager. Starting a new transaction while one is ongoing is an error. The optional parameter `flags` defaults to 0. Transaction flags are from the the enum `htcondor.TransactionFlags`, and the three flags are `NonDurable`, `SetDirty`, or `ShouldLog`. `NonDurable` is used for performance, as it eliminates extra `fsync()` calls. If the *condor_schedd* crashes before the transaction is written to disk, the transaction will be retried on restart of the *condor_schedd*. `SetDirty` marks the changed ClassAds as dirty, so an update notification is sent to the *condor_shadow* and the *condor_gridmanager*. `ShouldLog` causes changes to the job queue to be logged in the job event log file. The optional parameter `continue_txn` defaults to `false`; set the value to `true` to extend an ongoing transaction.
`act( (JobAction)action, (object)job_spec )` Change status of job(s) in the *condor_schedd* daemon. The integer return value is a `ClassAd` object describing the number of jobs changed. Parameter `action` is the action to perform; must be of the enum `JobAction`. Parameter `job_spec` is the job specification. It can either be a list of job IDs or a string specifying a constraint to match jobs.
`edit( (object)job_spec, (str)attr, (object)value )` Edit one or more jobs in the queue. Parameter `job_spec` is either a list of jobs, with each given as `ClusterId.ProcId` or a string containing a constraint to match jobs against. Parameter `attr` is the attribute name of the attribute to edit. Parameter `value` is the new value of the job attribute. It should be a string, which will be converted to a ClassAd expression, or an `ExprTree` object.
`query( constraint = true, attr_list = [] )` Query the *condor_schedd* daemon for jobs. Returns a list of ClassAds representing the matching jobs, containing at least the requested attributes requested by the second parameter. The optional parameter `constraint` provides a constraint for filtering out jobs. It defaults to `True`. Parameter `attr_list` is a list of attributes for the *condor_schedd* daemon to project along. It defaults to having the *condor_schedd* daemon return all attributes.
`xquery( constraint = true, attr_list = [], limit, opts, name )` Query the *condor_schedd* daemon for jobs. Returns an iterator of ClassAds representing the matching jobs containing at least the list of attributes requested by the second parameter. The optional parameter `constraint` provides a constraint for filtering out jobs. It defaults to `True`. Parameter `attr_list` is a list of attributes for the *condor_schedd* daemon to project along. It defaults to having the *condor_schedd* daemon return all attributes. Parameter `limit` is the maximum number of results this query will return. Parameter `opts` specifies any additional query options. Currently, the only non-default option is `QueryOpts.AutoCluster`, which returns autoclusters in the schedd, not jobs. Parameter `name` provides a *tag* name for the returned query iterator. This string will always be returned from the `tag()` method of the returned iterator. The default value is the *condor_schedd*'s name. This tag is useful to identify different queries when using the `poll()` module function.
`history( (object) requirements, (list) projection, (int) match )` Request history records from the *condor_schedd* daemon. Returns an iterator to a set of ClassAds representing completed jobs. Parameter `requirements` is either an `ExprTree` or a string that can be parsed as an expression. The expression represents the requirements that all returned jobs should match. Parameter `projection` is a list of all the ClassAd attributes that are to be included for each job. The empty list causes all attributes to be included. Parameter `match` is an integer cap on the number of jobs to include.

The `Submit` class:

`__init__( (dict)input = None )` Create an instance of the `Submit` class. Optional parameter `input` is a Python dictionary containing submit file key = value pairs. If omitted, the submit class is initially empty.
`expand( (str)attr )` Expand all macros for the given attribute. Parameter `attr` is the name of the relevant attribute. Returns a string containing the value of the given attribute with all macros expanded.
`queue( (object)txn, (int)count = 1, (object)ad_results = None )` Submit the current object to a remote queue. Parameter `txn` is an active transaction object (see `Schedd.transaction()`). Optional parameter `count` is the number of procs to create (defaults to 1 if not specified). Optional parameter `ad_results` is an object to receive the ClassAd resulting from this submit. Returns the ClusterID of the submitted job(s). Throws a RuntimeError if the submission fails.
`get( (str)attr, (str)default = None )` Gets the value of the specified attribute. Parameter `attr` is the name of the relevant attribute. Optional parameter `default` is a default value to be returned if the attribute is not defined. Returns a string containing the value of the attribute.
`setdefault( (str)attr, (str)default)` Set a default value for an attribute. Parameter `attr` is the name of the relevant attribute. Parameter `default` is the value to which to set the given attribute if that attribute has not already been set. Returns a string containing the value of the attribute.
`update( (object)submit )` Copy the contents of a given Submit object into the current object. Parameter `submit` is the Submit object to copy.

The `Collector` class:

`__init__( pool = None )` Create an instance of the `Collector` class. Optional parameter `pool` is a string with host:port pair specified or a list of pairs. If omitted, the value of configuration variable `COLLECTOR_HOST` is used.
`locate( (DaemonTypes)daemon_type, (str)name )` Query the *condor_collector* for a particular daemon. Returns the ClassAd of the requested daemon. Parameter `daemon_type` is the type of daemon; must be of the enum `DaemonTypes`. Optional parameter `name` is the name of daemon to locate. If not specified, it searches for the local daemon.
`locateAll( (DaemonTypes)daemon_type )` Query the *condor_collector* daemon for all ClassAds of a particular type. Returns a list of matching ClassAds. Parameter `daemon_type` is the type of daemon; must be of the enum `DaemonTypes`.
`query( (AdTypes)ad_type, constraint=True, projection=[], (str)statistics = ” )` Query the contents of a *condor_collector* daemon. Returns a list of ClassAds that match the `constraint` parameter. Optional parameter `ad_type` is the type of ClassAd to return, where the types are from the enum `AdTypes`. If not specified, the type will be `ANY_AD`. Optional parameter `constraint` is a constraint for the ClassAd query. It defaults to `True`. Optional parameter `projection` is a list of attributes. If specified, the returned ClassAds will be projected along these attributes. Optional parameter `statistics` is a list of statistics attributes to include, if they exist for the specified daemon.
`advertise( ad_list, command=UPDATE_AD_GENERIC, use_tcp = True )` Advertise a list of ClassAds into the *condor_collector*. Parameter `ad_list` is the list of ClassAds to advertise. Optional parameter `command` is a command for the *condor_collector*. It defaults to `UPDATE_AD_GENERIC`. Other commands, such as `UPDATE_STARTD_AD`, may require reduced authorization levels. Optional parameter `use_tcp` causes updates to be sent via TCP. Defaults to `True`.
`directQuery( (Collector)arg1, (DaemonTypes)daemon_type, (str)name = ”, (list)projection = [], (str)statistics = ” )` Query the specified daemon directly, instead of using the ClassAd from the *condor_collector* daemon. Returns the ClassAd of the specified daemon, after obtaining it from the daemon. Parameter `arg1` is the *condor_collector* that will identify where to find the specified daemon. Parameter `daemon_type` specified a daemon with an enum from `DaemonTypes`. Optional parameter `name` specifies the daemon's name. If not specified, the local daemon is used. Optional parameter `projection` is a list of attributes requested, to obtain only a subset of the attributes from the ClassAd. Optional parameter `statistics` is a list of statistics attributes to include, if they exist for the specified daemon.

The `Negotiator` class:

`__init__( (ClassAd)ad = None )` Create an instance of the `Negotiator` class. Optional parameter `ad` is a ClassAd containing the location of the *condor_negotiator* daemon. If omitted, uses the local pool.
`deleteUser( (str)user )` Delete a user from the accounting. `user` is a fully-qualified user name, `"USER@DOMAIN"`.
`getPriorities( [(bool)rollup = False ] )` Retrieve the pool accounting information. Returns a list of accounting ClassAds. Optional parameter `rollup` identifies if accounting information, as applied to hierarchical group quotas, should be summed for groups and subgroups (`True`) or not (`False`, the default).
`getResourceUsage( (str)user )` Get the resource usage for a specified user. Returns a list of ClassAd attributes. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`.
`resetAllUsage( )` Reset all usage accounting.
`resetUsage( (str)user )` Reset all usage accounting of the specified `user`. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`; resets the usage of only this user.
`setBeginUsage( (str)user, (time_t)value )` Initialize the time that a user begins using the pool. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `value` is the time of initial usage.
`setLastUsage( (str)user, (time_t)value )` Set the time that a user last began using the pool. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `value` is the time of last usage.
`setFactor( (str)user, (float)factor )` Set the priority factor of a specified user. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `factor` is the priority factor to be set for the user; must be greater than or equal to 1.0.
`setPriority( (str)user, (float)prio )` Set the real priority of a specified user. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `prio` is the priority to be set for the user; must be greater than 0.0.
`setUsage( (str)user, (float)usage )` Set the accumulated usage of a specified user. Parameter `user` is a fully-qualified user name, `"USER@DOMAIN"`. Parameter `usage` is the usage to be set for the user.

The `Startd` class:

`__init__( (ClassAd)ad = None )` Create an instance of the `Startd` class. Optional parameter `ad` is a ClassAd describing the claim (optional) and the startd location. If omitted, the local startd is assumed.
`drainJobs( (int)how_fast = Graceful, (bool)resume_on_completion = false, (expr)check_expr = true )` Begin draining jobs from the startd. Optional parameter `drain_type` is how fast to drain the jobs (from the DrainTypes enum: `Fast`, `Graceful` or `Quick`) (defaults to `Graceful` if not specified). Parameter `resume_on_completion` is `True` if the startd should start accepting jobs again once draining is complete, `False` if it should remain in the drained state (defaults to `False` if not specified). Optional parameter `check_expr` is an expression that must be `True` for all slots for draining to begin (defaults to `True` if not specified). Returns a (string) `request_id` that can be used to cancel draining.
`cancelDrainJobs( (object)request_id = None )` Cancel a draining request. Optional parameter `request_id` specifies a draining request to cancel; if not specified, all draining requests for this startd are canceled.

The `SecMan` class accesses the internal security object. This class allows access to the security layer of HTCondor.

Currently, this is limited to resetting security sessions and doing test authorizations against remote daemons.

If a security session becomes invalid, for example, because the remote daemon restarts, reuses the same port, and the client continues to use the session, then all future commands will fail with strange connection errors. This is the only mechanism to invalidate in-memory sessions.

`__init__( )` Create a `SecMan` object.
`invalidateAllSessions( )` Invalidate all security sessions. Any future connections to a daemon will cause a new security session to be created.

`ping ( (ClassAd)ad, (str)command )`
or
`ping ( (string)sinful, (str)command )`
Perform a test authorization against a remote daemon for a given command.
Returns the ClassAd of the security session.
Parameter `ad` is the ClassAd of the daemon as returned by `Collector.locate`; alternately, the sinful string can be given directly as the first parameter.
Optional parameter `command` is the DaemonCore command to try; if not given, `DC_NOP` will be used.

The `Param` class provides a dictionary-like interface to the current configuration.

The Param class:

`__getitem__( (str)attr )` Returns the configuration for variable `attr` as an object.
`__setitem__( (str)attr, (str)value )` Sets the configuration variable `attr` to the `value`.
`__contains__( (str)attr )` Determines whether the configuration contains a setting for configuration variable `attr`. Returns `true` if the configuration does contain a setting for `attr`, and it returns false otherwise. Parameter `attr` is the name of the configuration variable.
`__iter__( )` Description not yet written.
`__len__( )` Returns the number of items in the configuration.
`setdefault( (str)attr, (str)value )` Behaves like the corresponding Python dictionary method. If `attr` is not set in the configuration, it sets `attr` to `value` in the configuration. Returns the `value` as an object.
`get( )` `get` description not yet written.
`keys( )` Return a list of configuration variable names that are defined in the configuration files.
`items( )` Returns an iterator of tuples. Each item returned by the iterator is a tuple representing a pair (attribute,value) in the configuration.
`update( source )` Behaves like the corresponding Python dictionary method. Updates the current configuration to match the one in object `source`.

The `RemoteParam` class provides a dictionary-like interface to the configuration of daemons.

The RemoteParam class:

__getitem__((str)attr) Returns the configuration for variable attr as an object.
__setitem__((str)attr, (str)value) Sets the configuration variable attr to the value.
__contains__((str)attr) Determines whether the configuration contains a setting for configuration variable attr. Returns true if the configuration does contain a setting for attr, and it returns false otherwise. Parameter attr is the name of the configuration variable.
__iter__( ) Description not yet written.
__len__( ) Returns the number of items in the configuration.
__delitem__((str)attr) If the configuration variable specified by attr is in the configuration, set its value to the null string. Parameter attr is the name of the configuration variable to change.
setdefault((str)attr, (str)value) Behaves like the corresponding Python dictionary method. If attr is not set in the configuration, it sets attr to value in the configuration. Returns the value as an object.
get( ) get description not yet written.
keys( ) Return a list of configuration variable names that are defined for the daemon.
items( ) Returns an iterator of tuples. Each item returned by the iterator is a tuple representing a pair (attribute,value) in the configuration.
update(source) Behaves like the corresponding Python dictionary method. Updates the current configuration to match the one in object source.
refresh( ) Rebuilds the dictionary corresponding to the current configuration of the daemon.

The Claim class provides access to HTCondor's Compute-On-Demand facilities.

The Claim class:

`__init__( classad )` Create a Claim object. The `classad` argument provides a ClassAd describing the startd to claim.
`requestCOD( constraint, lease_duration )` Request a claim from the *condor_startd* represented by this object. The `constraint` specifies which slot in the startd to claim (defaults to 'true', which will result in the first slot becoming claimed). The `lease_duration` indicates how long the claim should be valid for. On success, the `Claim` object will represent a valid claim on the remote startd.
`release( (VacateTypes)vacate_type )` Release a *condor_startd* from this claim and shut down any running job. The `vacate_type` argument indicates the type of vacate to perform (Fast or Graceful); must be from VacateTypes enum.
`activate( (ClassAd)ad )` Activate a claim using a given job ad. The `ad` must describe a job to run.
`suspend()` Suspend an activated claim.
`renew()` Renew the lease on an existing claim.
`resume()` Resume a temporarily suspended claim.
`deactivate()` Deactivate a claim; shuts down the currently-running job, but holds onto the claim for future use.
`delegateGSIProxy()` Send an x509 proxy credential to an activated claim.

Module enums:

AdTypes A list of types used as values for the MyType ClassAd attribute. These types are only used by the HTCondor system, not the ClassAd language. Typically, these specify different kinds of daemons.
DaemonCommands A list of commands which can be sent to a remote daemon.
DaemonTypes A list of types of known HTCondor daemons.
JobAction A list of actions that can be performed on a job in a *condor_schedd*.
SubsystemType Distinguishes subsystems within HTCondor. Values may be Master, Collector, Negotiator, Schedd, Shadow, Startd, Starter, GAHP, Dagman, SharedPort, Daemon, Tool, Submit, or Job.
LogLevel The level at which events are logged. Values may be Always, Error, Status, Job, Machine, Config, Protocol, Priv, DaemonCore, Security, Network, Hostname, Audit, Terse, Verbose, FullDebug, SubSecond, Timestamp, PID, or NoHeader.

6.7.2 Sample Code using the htcondor Python Module

This sample code illustrates interactions with the htcondor Python Module.

```
$ python
Python 2.6.6 (r266:84292, Jun 18 2012, 09:57:52)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import htcondor
>>> import classad
>>> coll = htcondor.Collector("red-condor.unl.edu")
>>> results = coll.query(htcondor.AdTypes.Startd, "true", ["Name"])
>>> len(results)
3812
>>> results[0]
[ Name = "slot1@red-d20n35"; MyType = "Machine"; TargetType = "Job"; CurrentTime = time() ]
>>> scheddAd = coll.locate(htcondor.DaemonTypes.Schedd, "red-gw1.unl.edu")
>>> scheddAd["ScheddIpAddr"]
'<129.93.239.132:53020>'
>>> schedd = htcondor.Schedd(scheddAd)
>>> results = schedd.query('Owner =?= "cmsprod088"', ["ClusterId", "ProcId"])
>>> len(results)
63
>>> results[0]
[ MyType = "Job"; TargetType = "Machine"; ServerTime = 1356722353; ClusterId = 674143; ProcId = 0; CurrentTim
>>> htcondor.param["COLLECTOR_HOST"]
'hcc-briantest.unl.edu'
>>> schedd = htcondor.Schedd() # Defaults to the local schedd.
>>> results = schedd.query()
>>> results[0]["RequestMemory"]
ifthenelse(MemoryUsage isnt undefined,MemoryUsage,( ImageSize + 1023 ) / 1024)
>>> results[0]["RequestMemory"].eval()
```

```
1L
>>> ad=classad.parse(open("test.submit.ad"))
>>> print schedd.submit(ad, 2) # Submits two jobs in the cluster; edit test.submit.ad to preference.
110
>>> print schedd.act(htcondor.JobAction.Remove, ["111.0", "110.0"])'
    [
        TotalNotFound = 0;
        TotalPermissionDenied = 0;
        TotalAlreadyDone = 0;
        TotalJobAds = 2;
        TotalSuccess = 2;
        TotalChangedAds = 1;
        TotalBadStatus = 0;
        TotalError = 0
    ]
>>> print schedd.act(htcondor.JobAction.Hold, "Owner =?= \"bbockelm\"")'
    [
        TotalNotFound = 0;
        TotalPermissionDenied = 0;
        TotalAlreadyDone = 0;
        TotalJobAds = 2;
        TotalSuccess = 2;
        TotalChangedAds = 1;
        TotalBadStatus = 0;
        TotalError = 0
    ]
>>> schedd.edit('Owner =?= "bbockelm"', "Foo", classad.ExprTree('"baz"'))
>>> schedd.edit(["110.0"], "Foo", '"bar"')
>>> coll = htcondor.Collector()
>>> master_ad = coll.locate(htcondor.DaemonTypes.Master)
>>> htcondor.send_command(master_ad, htcondor.DaemonCommands.Reconfig) # Reconfigures the local master and al
>>> htcondor.version()
'$CondorVersion: 7.9.4 Jan 02 2013 PRE-RELEASE-UWCS $'
>>> htcondor.platform()
'$CondorPlatform: X86_64-ScientificLinux_6.3 $'
```

The bindings can use a dictionary where a ClassAd is expected. Here is an example that uses the ClassAd:

```
htcondor.Schedd().submit(classad.ClassAd({"Cmd": "/bin/echo"}))
```

This same example, using a dictionary instead of constructing a ClassAd:

```
htcondor.Schedd().submit({"Cmd": "/bin/echo"})
```

6.7.3 ClassAd Module

The `classad` module class provides a dictionary-like mechanism for interacting with the ClassAd language. `classad` objects implement the iterator interface to iterate through the `classad`'s attributes. The constructor can take a dictionary, and the object can take lists, dictionaries, and ClassAds as values.

`classad` module functions:

`parseOne( input, parser=Auto )` Parse the entire `input` into a single ClassAd. In the presence of multiple ClassAds or blank lines, continue to merge ClassAds together until the entire string is consumed. Returns a `classad` object. Parameter `input` is a string-like object or a file pointer. Parameter `parser` specifies which ClassAd parser to use.
`parseNext( input, parser=Auto )` Parse the next ClassAd in the input string. Advances the `input` object to point after the consumed ClassAd. Returns a `classad` object. Parameter `input` is a file-like object. Parameter `parser` specifies which ClassAd parser to use.
`parse( input )` *This method is no longer used.* Parse input into a ClassAd. Returns a ClassAd object. Parameter `input` is a string-like object or a file pointer.
`parseOld( input )` *This method is no longer used.* Parse old ClassAd format input into a ClassAd. Returns a ClassAd object. Parameter `input` is a string-like object or a file pointer.
`version( )` Return the version of the linked ClassAd library.
`lastError( )` Return the string representation of the last error to occur in the ClassAd library.
`Attribute( name )` Given the string `name`, return an `ExprTree` object which is a reference to an attribute of that name. The ClassAd expression `foo == 1` can be constructed by the python `Attribute("foo") == 1`.
`Function( name, arg1, arg2, ... )` Given function name `name`, and zero-or-more arguments, construct an `ExprTree` which is a function call expression. The function is not evaluated. The ClassAd expression `strcat("hello ", "world")` can be constructed by the python `Function("strcat", "hello ", "world")`.
`Literal( obj )` Given python object `obj`, convert it to a ClassAd literal. Python strings, floats, integers, and booleans have equivalent literals.
`register( function, name=None )` Given the python function `function`, register it as a ClassAd function. This allows the invocation of the python function from within a ClassAd evaluation context. The optional parameter, `name`, provides an alternate name for the function within the ClassAd library.
`registerLibrary( path )` Given a file system `path`, attempt to load it as a shared library of ClassAd functions. See the documentation for configuration variable `CLASSAD_USER_LIBS` for more information about loadable libraries for ClassAd functions.

Standard Python object methods for the `ClassAd` class:

`__init__( str )` Create a ClassAd object from string, `str`, passed as a parameter. The string must be formatted in the new ClassAd format.
`__len__( )` Returns the number of attributes in the ClassAd; allows `len(object)` semantics for ClassAds.
`__str__( )` Converts the ClassAd to a string and returns the string; the formatting style is new ClassAd, with square brackets and semicolons. For example, `[ Foo = "bar"; ]` may be returned.

The `classad` object has the following dictionary-like methods:

`items( )` Returns an iterator of tuples. Each item returned by the iterator is a tuple representing a pair (attribute,value) in the ClassAd object.
`values( )` Returns an iterator of objects. Each item returned by the iterator is a value in the ClassAd. If the value is a literal, it will be cast to a native Python object, so a ClassAd string will be returned as a Python string.
`keys( )` Returns an iterator of strings. Each item returned by the iterator is an attribute string in the ClassAd.
`get( attr, value )` Behaves like the corresponding Python dictionary method. Given the `attr` as key, returns either the value of that key, or if the key is not in the object, returns `None` or the optional second parameter when specified.
`__getitem__( attr )` Returns (as an object) the value corresponding to the attribute `attr` passed as a parameter. ClassAd values will be returned as Python objects; ClassAd expressions will be returned as `ExprTree` objects.
`__setitem__( attr, value )` Sets the ClassAd attribute `attr` to the `value`. ClassAd values will be returned as Python objects; ClassAd expressions will be returned as `ExprTree` objects.
`setdefault( attr, value )` Behaves like the corresponding Python dictionary method. If called with an attribute, `attr`, that is not set, it will set the attribute to the specified `value`. It returns the value of the attribute. If called with an attribute that is already set, it does not change the object.
`update( object )` Behaves like the corresponding Python dictionary method. Updates the ClassAd with the key/value pairs of the given object. Returns nothing.

Additional methods:

`eval( attr )` Evaluate the value given a ClassAd attribute `attr`. Throws `ValueError` if unable to evaluate the object. Returns the Python object corresponding to the evaluated ClassAd attribute.
`lookup( attr )` Look up the `ExprTree` object associated with attribute `attr`. No attempt will be made to convert to a Python object. Returns an `ExprTree` object.
`printOld( )` Print the ClassAd in the old ClassAd format. Returns a string.
`quote( str )` Converts the Python string, `str`, into a ClassAd string literal. Returns the string literal.
`unquote( str )` Converts the Python string, `str`, escaped as a ClassAd string back to a Python string. Returns the Python string.
`parseAds( input, parser=Auto )` Given `input` of a string or file, return an iterator of ClassAds. Parameter `parser` tells which ClassAd parser to use. Note that automatic selection of ClassAd parser does not work on stream input. Returns an iterator.
`parseOldAds( input )` *This method is no longer used.* Given `input` of a string or file, return an iterator of ClassAds where the ClassAds are in the Old ClassAd format. Returns an iterator.
`flatten( expression )` Given `ExprTree` object `expression`, perform a partial evaluation. All the attributes in `expression` and defined in this object are evaluated and expanded. Any constant expressions, such as `1 + 2`, are evaluated. Returns a new `ExprTree` object.
`matches( ad )` Given `ClassAd` object `ad`, check to see if this object matches the `Requirements` attribute of `ad`. Returns `true` if it does.
`symmetricMatch( ad )` Returns `true` if the given `ad` matches this and this matches `ad`. Equivalent to `self.matches(ad) and ad.matches(self)`.
`externalRefs( expr )` Returns a python list of external references found in `expr`. In this context, an external reference is any attribute in the expression which is *not* found in the `ClassAd`.
`internalRefs( expr )` Returns a python list of internal references found in `expr`. In this context, an internal reference is any attribute in the expression which is found in the `ClassAd`.

The `ExprTree` class object represents an expression in the ClassAd language. The python operators for

ExprTree have been overloaded so, if e1 and e2 are ExprTree objects, then e1 + e2 is also a ExprTree object. Lazy-evaluation is used, so an expression "foo" + 1 does not produce an error until it is evaluated with a call to bool() or the .eval() class member.

ExprTree class methods:

__init__(str) Parse the string str to create an ExprTree.
__str__( ) Represent and return the ClassAd expression as a string.
__int__( ) Converts expression to an integer (evaluating as necessary).
__float__( ) Converts expression to a float (evaluating as necessary).
eval( ) Evaluate the expression and return as a ClassAd value, typically a Python object.

Module enums:

Parser Tells which ClassAd parser to use. Values may be Auto, Old, or New.

6.7.4 Sample Code using the classad Module

This sample Python code illustrates interactions with the classad module.

```
$ python
Python 2.6.6 (r266:84292, Jun 18 2012, 09:57:52)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import classad
>>> ad = classad.ClassAd()
>>> expr = classad.ExprTree("2+2")
>>> ad["foo"] = expr
>>> print ad["foo"].eval()
4
>>> ad["bar"] = 2.1
>>> ad["baz"] = classad.ExprTree("time() + 4")
>>> print list(ad)
['bar', 'foo', 'baz']
>>> print dict(ad.items())
{'baz': time() + 4, 'foo': 2 + 2, 'bar': 2.100000000000000E+00}
>>> print ad
    [
        bar = 2.100000000000000E+00;
        foo = 2 + 2;
        baz = time() + 4
    ]
```

```
>>> ad2=classad.parseOne(open("test_ad", "r"));
>>> ad2["error"] = classad.Value.Error
>>> ad2["undefined"] = classad.Value.Undefined
>>> print ad2
    [
        error = error;
        bar = 2.100000000000000E+00;
        foo = 2 + 2;
        undefined = undefined;
        baz = time() + 4
    ]
>>> ad2["undefined"]
classad.Value.Undefined
```

Here is an example that illustrates the dictionary properties of the constructor.

```
>>> classad.ClassAd({"foo": "bar"})
[ foo = "bar" ]
>>> ad = classad.ClassAd({"foo": [1, 2, 3]})
>>> ad
[ foo = { 1,2,3 } ]
>>> ad["foo"][2]
3L
>>> ad = classad.ClassAd({"foo": {"bar": 1}})
>>> ad
[ foo = [ bar = 1 ] ]
>>> ad["foo"]["bar"]
1L
```

Here are examples that illustrate the `get` method.

```
>>> ad = classad.ClassAd({"foo": "bar"})
>>> ad
[ foo = "bar" ]
>>> ad["foo"]
'bar'
>>> ad.get("foo")
'bar'
>>> ad.get("foo", 2)
'bar'
>>> ad.get("baz", 2)
2
>>> ad.get("baz")
>>>
```

Here are examples that illustrate the `setdefault` method.

```
>>> ad = classad.ClassAd()
```

```
>>> ad
[  ]
>>> ad["foo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'foo'
>>> ad.setdefault("foo", 1)
1
>>> ad
[ foo = 1 ]
>>> ad.setdefault("foo", 2)
1L
>>> ad
[ foo = 1 ]
```

Here is an example that illustrates the use of the iterator parseAds method on a history log.

```
>>> import classad
>>> import os
>>> fd = os.popen("condor_history -l -match 4")
>>> ads = classad.parseAds(fd, classad.Parser.Old)
>>> print [ad["ClusterId"] for ad in ads]
[23389L, 23388L, 23386L, 23387L]
>>>
```

Chapter 7

Platform-Specific Information

The HTCondor Team strives to make HTCondor work the same way across all supported platforms. However, because HTCondor 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 HTCondor on different computing platforms and operating systems.

7.1 Linux

This section provides information specific to the Linux port of HTCondor. Linux is a difficult platform to support. It changes frequently, and HTCondor has some extremely system-dependent code, such as the checkpointing library.

HTCondor 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 HTCondor jobs. This matters for HTCondor's standard universe, which provides checkpoints and remote system calls.

The HTCondor 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 HTCondor can aim support. HTCondor 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 HTCondor 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 HTCondor that works with one release of Red Hat might not work with newer releases. The following sections describe the details of HTCondor's support for the currently available versions of Red Hat Linux on x86 architecture machines.

7.1.1 Linux Address Space Randomization

Modern versions of Red Hat and Fedora do address space randomization, which randomizes the memory layout of a process to reduce the possibility of security exploits. This makes it impossible for standard universe jobs to resume execution using a checkpoint. When starting or resuming a standard universe job, HTCondor disables the randomization.

To run a binary compiled with *condor_compile* in standalone mode, either initially or in resumption mode, manually disable the address space randomization by modifying the command line. For a 32-bit architecture, assuming an HTCondor-linked binary called *myapp*, invoke the standalone executable with:

```
setarch i386 -L -R ./myapp
```

For a 64-bit architecture, the resumption command will be:

```
setarch x86_64 -L -R ./myapp
```

Some applications will also need the **-B** option.

The command to resume execution using the checkpoint must also disable address space randomization, as the 32-bit architecture example:

```
setarch i386 -L -R myapp -_condor_restart myapp.ckpt
```

7.2 Microsoft Windows

Windows is a strategic platform for HTCondor, and therefore we have been working toward a complete port to Windows. Our goal is to make HTCondor every bit as capable on Windows as it is on Unix – or even more capable.

Porting HTCondor from Unix to Windows is a formidable task, because many components of HTCondor must interact closely with the underlying operating system. Provided is a clipped version of HTCondor 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 HTCondor on Windows. In order to effectively use HTCondor, first read the overview chapter (section 1.1) and the user's manual (section 2.1). If administrating or customizing the policy and set up of HTCondor, 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 HTCondor on Windows. For information on installing HTCondor for Windows, see section 3.2.3.

7.2.1 Limitations under Windows

In general, this release for Windows works the same as the release of HTCondor 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.
- **grid** universe jobs may not be submitted from a Windows platform, unless the grid type is **condor**.
- Accessing files via a network share that requires a Kerberos ticket (such as AFS) is not yet supported.

7.2.2 Supported Features under Windows

Except for those items listed above, most everything works the same way in HTCondor as it does in the Unix release. This release is based on the HTCondor Version 8.6.10 source tree, and thus the feature set is the same as HTCondor Version 8.6.10 for Unix. For instance, all of the following work in HTCondor:

- 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 an HTCondor central manager on Windows.
- Security mechanisms.
- HTCondor 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, HTCondor can automatically suspend a job whenever keyboard/mouse or non-HTCondor created CPU activity is detected, and continue the job after the machine has been idle for a specified amount of time.
- HTCondor correctly manages jobs which create multiple processes. For instance, if an HTCondor job spawns multiple processes and HTCondor 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 HTCondor by e-mail (standard SMTP) and/or by log files.
- HTCondor includes a friendly GUI installation and set up program, which can perform a full install or deinstall of HTCondor. 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.
- HTCondor can give a job access to the running user's Registry hive.

7.2.3 Secure Password Storage

In order for HTCondor to operate properly, it must at times be able to act on behalf of users who submit jobs. This is required on submit machines, so that HTCondor 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. On Unix systems, arbitrarily changing what user HTCondor performs its actions as is easily done when HTCondor is started with root privileges. On Windows, however, performing an action as a particular user or on behalf of a particular user requires knowledge of that user's password, even when running at the maximum privilege level. HTCondor provides secure password storage through the use of the *condor_store_cred* tool. Passwords managed by HTCondor are encrypted and stored in a secure location within the Windows registry. When HTCondor needs to perform an action as or on behalf of a particular user, it uses the securely stored password to do so. This implies that a password is stored for every user that will submit jobs from the Windows submit machine.

A further feature permits HTCondor to execute the job itself under the security context of its submitting user, specifying the **run_as_owner** command in the job's submit description file. With this feature, it is necessary to configure and run a centralized *condor_credd* daemon to manage the secure password storage. This makes each user's password available, via an encrypted connection to the *condor_credd*, to any execute machine that may need it.

By default, the secure password store for a submit machine when no *condor_credd* is running is managed by the *condor_schedd*. This approach works in environments where the user's password is only needed on the submit machine.

7.2.4 Executing Jobs as the Submitting User

By default, HTCondor executes jobs on Windows using dedicated run accounts that have minimal access rights and privileges, and which are recreated for each new job. As an alternative, HTCondor 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 are not available to the low-privilege run account.

This feature requires use of a *condor_credd* daemon for secure password storage and retrieval. With the *condor_credd* daemon running, the user's password must be stored, using the *condor_store_cred* tool. Then, a user that wants a job to run using their own account places into the job's submit description file

```
run_as_owner = True
```

7.2.5 The condor_credd Daemon

The *condor_credd* daemon manages secure password storage. A single running instance of the *condor_credd* within an HTCondor pool is necessary in order to provide the feature described in section 7.2.4, where a job runs as the submitting user, instead of as a temporary user that has strictly limited access capabilities.

It is first necessary to select the single 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 file `etc\condor_config.local.credd` from the HTCondor 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 variables specify security for the communication among HTCondor daemons. These variables must be set for all machines in the pool. The following example settings are in the comments contained in the `etc\condor_config.local.credd` example file. These sample settings rely on the `PASSWORD` method for authentication among daemons, including communication with the *condor_credd* daemon. The `LOCAL_CREDD` variable must be customized to point to the machine hosting the *condor_credd* and the `ALLOW_CONFIG` variable will be customized, if needed, to refer to an administrative account that exists on all HTCondor nodes.

```
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 example above can be modified to meet the needs of your pool, providing the following conditions are met:

1. The requesting client must use an authenticated connection
2. The requesting client must have an encrypted connection
3. The requesting client must be authorized for `DAEMON` level access.

Using a pool password on Windows

In order for `PASSWORD` authenticated communication to work, a *pool password* must be chosen and distributed. The chosen pool password must be stored identically for each machine. The pool password first should be stored on the *condor_credd* host, then on the other machines in the pool.

To store the pool password on a Windows machine, run

```
condor_store_cred add -c
```

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, that is 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 for each Windows machine using a command of the form

```
condor_store_cred add -c -n exec01.cs.wisc.edu
```

Once the pool password is distributed, but before submitting jobs, all machines must reevaluate their configuration, so 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.

7.2.6 Executing Jobs with the User's Profile Loaded

HTCondor 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 profile is cleaned before a subsequent job which uses the dedicated run account begins. 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 job that is 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.

7.2.7 Using Windows Scripts as Job Executables

HTCondor has added support for scripting jobs on Windows. Previously, HTCondor jobs on Windows were limited to executables or batch files. With this new support, HTCondor determines how to interpret the script using the file name's extension. Without a file name extension, the file will be treated as it has been in the past: as a Windows executable.

This feature may not require any modifications to HTCondor's configuration. An example that does not require administrative intervention are Perl scripts using *ActivePerl*.

Windows Scripting Host scripts do require configuration to work correctly. The configuration variables set values to be used in registry look up, which results in a command that invokes the correct interpreter, with the correct command

line arguments for the specific scripting language. In Microsoft nomenclature, *verbs* are actions that can be taken upon a given a file. The familiar examples of **Open**, **Print**, and **Edit**, can be found on the context menu when a user right clicks on a file. The command lines to be used for each of these verbs are stored in the registry under the `HKEY_CLASSES_ROOT` hive. In general, a registry look up uses the form:

```
HKEY_CLASSES_ROOT\<FileType>\Shell\<OpenVerb>\Command
```

Within this specification, `<FileType>` is the name of a file type (and therefore a scripting language), and is obtained from the file name extension. `<OpenVerb>` identifies the verb, and is obtained from the HTCondor configuration.

The HTCondor configuration sets the selection of a verb, to aid in the registry look up. The file name extension sets the name of the HTCondor configuration variable. This variable name is of the form:

```
OPEN_VERB_FOR_<EXT>_FILES
```

`<EXT>` represents the file name extension. The following configuration example uses the `Open2` verb for a *Windows Scripting Host* registry look up for several scripting languages:

```
OPEN_VERB_FOR_JS_FILES  = Open2
OPEN_VERB_FOR_VBS_FILES = Open2
OPEN_VERB_FOR_VBE_FILES = Open2
OPEN_VERB_FOR_JSE_FILES = Open2
OPEN_VERB_FOR_WSF_FILES = Open2
OPEN_VERB_FOR_WSH_FILES = Open2
```

In this example, HTCondor specifies the `Open2` verb, instead of the default `Open` verb, for a script with the file name extension of `wsh`. The *Windows Scripting Host*'s `Open2` verb allows standard input, standard output, and standard error to be redirected as needed for HTCondor jobs.

A common difficulty is encountered when a script interpreter requires access to the user's registry. Note that the user's registry is different than the root registry. If not given access to the user's registry, some scripts, such as *Windows Scripting Host* scripts, will fail. The failure error message appears as:

```
CScript Error: Loading your settings failed. (Access is denied.)
```

The fix for this error is to give explicit access to the submitting user's registry hive. This can be accomplished with the addition of the **load_profile** command in the job's submit description file:

```
load_profile = True
```

With this command, there should be no registry access errors. This command should also work for other interpreters. Note that not all interpreters will require access. For example, *ActivePerl* does not by default require access to the user's registry hive.

7.2.8 How HTCondor for Windows Starts and Stops a Job

This section provides some details on how HTCondor starts and stops jobs. This discussion is geared for the HTCondor administrator or advanced user who is already familiar with the material in the Administrator's Manual and wishes to know detailed information on what HTCondor does when starting and stopping jobs.

When HTCondor 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-slot<X>`, where `<X>` is the slot number of the *condor_starter*. This account is added to group `Users` by default. The default group may be changed by setting configuration variable `DYNAMIC_RUN_ACCOUNT_LOCAL_GROUP`. This step is skipped if the job is to be run using the submitting user's account, as specified in section 7.2.4.
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 HTCondor's configuration file. HTCondor 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. 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* daemon contacts the *condor_shadow* daemon, which is running on the submitting machine, and the *condor_starter* 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 *condor_starter* spawns the user's executable. Its current working directory set to the temporary working directory.

While the job is running, the *condor_starter* closely monitors the CPU usage and image size of all processes started by the job. Every 20 minutes the *condor_starter* sends this information, along with the total size of all files contained in the job's temporary working directory, to the *condor_shadow*. The *condor_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 *condor_starter* first terminates any processes started by the job which could still be around if the job did not clean up after itself. The *condor_starter* examines the job's temporary working directory for any files which have been created or modified and sends these files back to the *condor_shadow* running on the submit machine. The *condor_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 *condor_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. This is the one started by the *condor_starter*. The WM_CLOSE message is the preferred way to terminate a process on Windows, since this method allows the job to clean up and free any resources it may have allocated. When the job

exits, the *condor_starter* cleans up any processes left behind. At this point, if **when_to_transfer_output** is set to `ON_EXIT` (the default) in the job's submit description file, the job switches states, from Running to Idle, and no files are transferred back. If **when_to_transfer_output** is set to `ON_EXIT_OR_EVICT`, then files in the job's temporary working directory which were changed or modified are first sent back to the submitting machine. If exactly which files to transfer is specified with **transfer_output_files**, then this modifies the files transferred and can affect the state of the job if the specified files do not exist. On an eviction, the *condor_shadow* places these intermediate files into a subdirectory created in the `$(SPOOL)` directory on the submitting machine. The job is then switched back to the Idle state until HTCondor finds a different machine on which to run. When the job is started again, HTCondor places into the job's temporary working directory the executable and input files as before, *plus* any files stored in the submit machine's `$(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 *condor_starter* deletes the temporary working directory, the temporary account if one was created, the Window Station and the Desktop before exiting. If the *condor_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 HTCondor is restarted.

7.2.9 Security Considerations in HTCondor for Windows

On the execute machine (by default), the user job is run using the access token of an account dynamically created by HTCondor 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 HTCondor 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 7.2.4), 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, HTCondor 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, HTCondor 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, HTCondor is not able to check and see if the now-defunct submitting user has read/write access to a given file. In this case, HTCondor 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.

HTCondor 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 HTCondor'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. HTCondor 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 HTCondor 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 HTCondor installation program will automatically set permissions on the entire HTCondor release directory so that only Administrators have write access.

Finally, HTCondor has all the IP/Host-based security mechanisms present in the full-blown version of HTCondor. See section 3.8.9 starting on page 421 for complete information on how to allow/deny access to HTCondor based upon machine host name or IP address.

7.2.10 Network files and HTCondor

HTCondor 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 7.2.4). 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, HTCondor 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 HTCondor use a special account

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 HTCondor 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 HTCondor 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 HTCondor use a special account

Create a permanent account (called condor-guest in this description) under which HTCondor 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 HTCondor jobs.

Securely store the password of the condor-guest user in the Windows registry using *condor_store_cred* on all Windows machines.

Tell HTCondor to use the condor-guest user as the owner of jobs, when required. Details for this are in section 3.8.13.

7.2.11 Interoperability between HTCondor for Unix and HTCondor for Windows

Unix machines and Windows machines running HTCondor can happily co-exist in the same HTCondor 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** command 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 HTCondor central manager, even if there are both Unix and Windows machines in the pool. The HTCondor central manager itself can run on either Unix or Windows; there is no advantage to choosing one over the other.

7.2.12 Some differences between HTCondor for Unix -vs- HTCondor for Windows

- On Unix, we recommend the creation of a `condor` account when installing HTCondor. On Windows, this is not necessary, as HTCondor is designed to run as a system service as user LocalSystem.
- On Unix, HTCondor finds the `condor_config` main configuration file by looking in ~condor, in `/etc`, or via an environment variable. On Windows, the location of `condor_config` file is determined via the registry

key `HKEY_LOCAL_MACHINE/Software/Condor`. Override this value by setting an environment variable named `CONDOR_CONFIG`.

- On Unix, in the vanilla universe at job vacate time, HTCondor sends the job a softkill signal defined in the submit description file, which defaults to SIGTERM. On Windows, HTCondor sends a WM_CLOSE message to the job at vacate time.
- On Unix, if one of the HTCondor daemons has a fault, a core file will be created in the `$(Log)` directory. On Windows, 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 HTCondor developers to fix the problem.

7.3 Macintosh OS X

This section provides information specific to the Macintosh OS X port of HTCondor. The Macintosh port of HTCondor is more accurately a port of HTCondor to Darwin, the BSD core of OS X. HTCondor uses the Carbon library only to detect keyboard activity, and it does not use Cocoa at all. HTCondor on the Macintosh is a relatively new port, and it is not yet well-integrated into the Macintosh environment.

HTCondor 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.

Chapter 8

Frequently Asked Questions (FAQ)

There are many Frequently Asked Questions maintained on the HTCondor web page, at http://htcondor-wiki.cs.wisc.edu/index.cgi/wiki and on the configuration how-to and recipes page at https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToAdminRecipes

Supported platforms are listed in section 1.5, on page 5. There is also platform-specific information at Chapter 7 on page 641.

Chapter 9

Contrib and Source Modules

9.1 Introduction

Contrib modules are stand alone, separate pieces of code that work together with HTCondor to accomplish some task. These modules are available by following links from the wiki at https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki. Documentation for these modules is either here and identified as a contrib module, or may be within the module itself.

Other features of HTCondor are available within the source code, but are not compiled in to the binaries distributed. To utilize these features, acquire the source code and build it. Enable the feature as described in this documentation.

This chapter documents the HTCondorView Client contrib module, Quill (available with the source code), and using HTCondor with the Hadoop File System (available with the source code).

9.2 Using HTCondor with the Hadoop File System

The Hadoop project is an Apache project, headquartered at http://hadoop.apache.org, which implements an open-source, distributed file system across a large set of machines. The file system proper is called the Hadoop File System, or HDFS, and there are several Hadoop-provided tools which use the file system, most notably databases and tools which use the map-reduce distributed programming style.

Distributed with the HTCondor source code, HTCondor provides a way to manage the daemons which implement an HDFS, but no direct support for the high-level tools which run atop this file system. There are two types of daemons, which together create an instance of a Hadoop File System. The first is called the Name node, which is like the central manager for a Hadoop cluster. There is only one active Name node per HDFS. If the Name node is not running, no files can be accessed. The HDFS does not support fail over of the Name node, but it does support a hot-spare for the Name node, called the Backup node. HTCondor can configure one node to be running as a Backup node. The second kind of daemon is the Data node, and there is one Data node per machine in the distributed file system. As these are both implemented in Java, HTCondor cannot directly manage these daemons. Rather, HTCondor provides a small DaemonCore daemon, called *condor_hdfs*, which reads the HTCondor configuration file, responds to HTCondor

commands like *condor_on* and *condor_off*, and runs the Hadoop Java code. It translates entries in the HTCondor configuration file to an XML format native to HDFS. These configuration items are listed with the *condor_hdfs* daemon in section 9.2.1. So, to configure HDFS in HTCondor, the HTCondor configuration file should specify one machine in the pool to be the HDFS Name node, and others to be the Data nodes.

Once an HDFS is deployed, HTCondor jobs can directly use it in a vanilla universe job, by transferring input files directly from the HDFS by specifying a URL within the job's submit description file command **transfer_input_files**. See section 3.14.2 for the administrative details to set up transfers specified by a URL. It requires that a plug-in is accessible and defined to handle `hdfs` protocol transfers.

9.2.1 condor_hdfs Configuration File Entries

These macros affect the *condor_hdfs* daemon. Many of these variables determine how the *condor_hdfs* daemon sets the HDFS XML configuration.

HDFS_HOME The directory path for the Hadoop file system installation directory. Defaults to `$(RELEASE_DIR)/libexec`. This directory is required to contain

- directory `lib`, containing all necessary jar files for the execution of a Name node and Data nodes.
- directory `conf`, containing default Hadoop file system configuration files with names that conform to `*-site.xml`.
- directory `webapps`, containing JavaServer pages (jsp) files for the Hadoop file system's embedded server.

HDFS_NAMENODE The host and port number for the HDFS Name node. There is no default value for this required variable. Defines the value of `fs.default.name` in the HDFS XML configuration.

HDFS_NAMENODE_WEB The IP address and port number for the HDFS embedded web server within the Name node with the syntax of `a.b.c.d:portnumber`. There is no default value for this required variable. Defines the value of `dfs.http.address` in the HDFS XML configuration.

HDFS_DATANODE_WEB The IP address and port number for the HDFS embedded web server within the Data node with the syntax of `a.b.c.d:portnumber`. The default value for this optional variable is 0.0.0.0:0, which means bind to the default interface on a dynamic port. Defines the value of `dfs.datanode.http.address` in the HDFS XML configuration.

HDFS_NAMENODE_DIR The path to the directory on a local file system where the Name node will store its meta-data for file blocks. There is no default value for this variable; it is required to be defined for the Name node machine. Defines the value of `dfs.name.dir` in the HDFS XML configuration.

HDFS_DATANODE_DIR The path to the directory on a local file system where the Data node will store file blocks. There is no default value for this variable; it is required to be defined for a Data node machine. Defines the value of `dfs.data.dir` in the HDFS XML configuration.

HDFS_DATANODE_ADDRESS The IP address and port number of this machine's Data node. There is no default value for this variable; it is required to be defined for a Data node machine, and may be given the value `0.0.0.0:0` as a Data node need not be running on a known port. Defines the value of `dfs.datanode.address` in the HDFS XML configuration.

HDFS_NODETYPE This parameter specifies the type of HDFS service provided by this machine. Possible values are HDFS_NAMENODE and HDFS_DATANODE. The default value is HDFS_DATANODE.

HDFS_BACKUPNODE The host address and port number for the HDFS Backup node. There is no default value. It defines the value of the HDFS dfs.namenode.backup.address field in the HDFS XML configuration file.

HDFS_BACKUPNODE_WEB The address and port number for the HDFS embedded web server within the Backup node, with the syntax of hdfs://<host_address>:<portnumber>. There is no default value for this required variable. It defines the value of dfs.namenode.backup.http-address in the HDFS XML configuration.

HDFS_NAMENODE_ROLE If this machine is selected to be the Name node, then the role must be defined. Possible values are ACTIVE, BACKUP, CHECKPOINT, and STANDBY. The default value is ACTIVE. The STANDBY value exists for future expansion. If HDFS_NODETYPE is selected to be Data node (HDFS_DATANODE), then this variable is ignored.

HDFS_LOG4J Used to set the configuration for the HDFS debugging level. Currently one of OFF, FATAL, ERROR, WARN, INFODEBUG, ALL or INFO. Debugging output is written to $(LOG)/hdfs.log. The default value is INFO.

HDFS_ALLOW A comma separated list of hosts that are authorized with read and write access to the invoked HDFS. Note that this configuration variable name is likely to change to HOSTALLOW_HDFS.

HDFS_DENY A comma separated list of hosts that are denied access to the invoked HDFS. Note that this configuration variable name is likely to change to HOSTDENY_HDFS.

HDFS_NAMENODE_CLASS An optional value that specifies the class to invoke. The default value is org.apache.hadoop.hdfs.server.namenode.NameNode.

HDFS_DATANODE_CLASS An optional value that specifies the class to invoke. The default value is org.apache.hadoop.hdfs.server.datanode.DataNode.

HDFS_SITE_FILE The optional value that specifies the HDFS XML configuration file to generate. The default value is hdfs-site.xml.

HDFS_REPLICATION An integer value that facilitates setting the replication factor of an HDFS, defining the value of dfs.replication in the HDFS XML configuration. This configuration variable is optional, as the HDFS has its own default value of 3 when not set through configuration.

9.3 Quill

Quill is an optional component of HTCondor that maintains a mirror of HTCondor 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.

As of HTCondor version 7.5.5, Quill is distributed only with the source code. It is not included in the builds of HTCondor provided by UW, but it is available as a feature that can be enabled by those who compile HTCondor from the source code. Find the code within the condor_contrib directory, in the directories condor_tt and condor_dbmsd.

9.3.1 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 HTCondor 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 a single line that has fields separated by colons and is properly terminated by an operating system specific newline character (Unix) or CRLF (Windows). 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` . HTCondor 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 HTCondor release directory:

```
psql [databasename] [username] < common_createddl.sql
psql [databasename] [username] < pgsql_createddl.sql
```

where `[username]` will be `quillwriter`.

After *PostgreSQL* is configured and running, HTCondor 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 HTCondor 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 HTCondor 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.

Only 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
```

Configuration Variables

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.

QUILL_ARGS Arguments to be passed to the *condor_quill* daemon upon its invocation.

QUILL_LOG Path to the Quill daemon's log file.

QUILL_ENABLED 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.

QUILL_NAME 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 3.5.7 on page 242 for defaults and composition of valid HTCondor daemon names.

QUILL_USE_SQL_LOG In order for Quill to store historical job information or resource information, the HTCondor 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 `QUILL_USE_SQL_LOG` to `False` and set `SCHEDD._QUILL_USE_SQL_LOG` to `True`.

QUILL_DB_NAME A string that identifies a database within a database server.

QUILL_DB_USER A string that identifies 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.

QUILL_DB_TYPE A string that distinguishes between database system types. Defaults to the only database system currently defined, `"PGSQL"`.

QUILL_DB_IP_ADDR 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. 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_POLLING_PERIOD 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. 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_NOT_RESPONDING_TIMEOUT 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.

QUILL_MAINTAIN_DB_CONN 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.

DATABASE_PURGE_INTERVAL 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.

QUILL_JOB_HISTORY_DURATION The number of days after entry into the database that a job will remain in the database. After `QUILL_JOB_HISTORY_DURATION` 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.

QUILL_RUN_HISTORY_DURATION The number of days after entry into the database that extra information about the job will remain in the database. After `QUILL_RUN_HISTORY_DURATION` 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.

QUILL_RESOURCE_HISTORY_DURATION The number of days after entry into the database that a resource record will remain in the database. After QUILL_RESOURCE_HISTORY_DURATION 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.

QUILL_DBSIZE_LIMIT At intervals of time set by DATABASE_PURGE_INTERVAL, the *condor_quill* daemon estimates the size of the database. If the size of the database exceeds the limit set by this variable, the *condor_quill* daemon will e-mail the administrator a warning. This size is given in Gbytes, and defaults to 20.

QUILL_MANAGE_VACUUM 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.

QUILL_SHOULD_REINDEX 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.

DATABASE_REINDEX_INTERVAL Because *PostgreSQL* does not aggressively maintain the index structures for deleted tuples, it can lead to bloated index structures. This variable is the interval, in seconds, between re-index 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".

QUILL_IS_REMOTELY_QUERYABLE A boolean value that defaults to True. 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. Note that even if False, one can still query the job queue at the remote *condor_schedd* daemon.

QUILL_DB_QUERY_PASSWORD Defines the password string needed by *condor_q* to gain read access for remotely querying the Quill database. 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.

QUILL_ADDRESS_FILE When defined, it specifies the path and file name of a local file that contains 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 re-indexing 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.

9.3.2 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 -avgqueuetime
   ```

   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.

9.3.3 Quill and Security

There are several layers of security in Quill, some provided by HTCondor 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 HTCondor.

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 HTCondor 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 HTCondor query tools.

9.3.4 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 HTCondor 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 throwns 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.
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.
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)	HTCondor daemon 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)	HTCondor 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 transfered 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 transfered.
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 HTCondor 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 HTCondor-G placed the job on hold.
condorversion	varchar(4000)	varchar(4000)	Version of HTCondor 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 HTCondor 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 HTCondor should transfer files to and from the machine where the job runs.
transferfiles	varchar(4000)	varchar(4000)	Depreciated. 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 HTCondor.
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.
rscbytessent	numeric(38)	numeric(38)	Number of remote system call bytes sent to the job.
rscbytesrecvd	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)	HTCondor state of the machine.
activity	varchar(4000)	varchar(4000)	HTCondor 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 HTCondor
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 HTCondor 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 HTCondor 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)	HTCondor state of the machine.
activity	varchar(4000)	varchar(4000)	HTCondor 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 HTCondor
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 HTCondor 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 HTCondor 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.
attname	varchar(4000)	varchar(4000)	The attribute name.
attval	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.

9.4 The HTCondorView Client Contrib Module

The HTCondorView Client contrib module is used to automatically generate World Wide Web pages to display usage statistics of an HTCondor pool. Included in the module is a shell script which invokes the *condor_stats* command to retrieve pool usage statistics from the HTCondorView server, and generate HTML pages from the results. Also included is a Java applet, which graphically visualizes HTCondor usage information. Users can interact with the applet to customize the visualization and to zoom in to a specific time frame. Figure 9.1 on page 685 is a screen shot of a web page created by HTCondorView.

After unpacking and installing the HTCondorView Client, a script named *make_stats* can be invoked to create HTML pages displaying HTCondor usage for the past hour, day, week, or month. By using the Unix *cron* facility to periodically execute *make_stats*, HTCondor pool usage statistics can be kept up to date automatically. This simple model allows the HTCondorView Client to be easily installed; no Web server CGI interface is needed.

9.4.1 Step-by-Step Installation of the HTCondorView Client

1. Make certain that the HTCondorView Server is configured. Section 3.14.6 describes configuration of the server. The server logs information on disk in order to provide a persistent, historical database of pool statistics. The HTCondorView 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 HTCondorView 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.

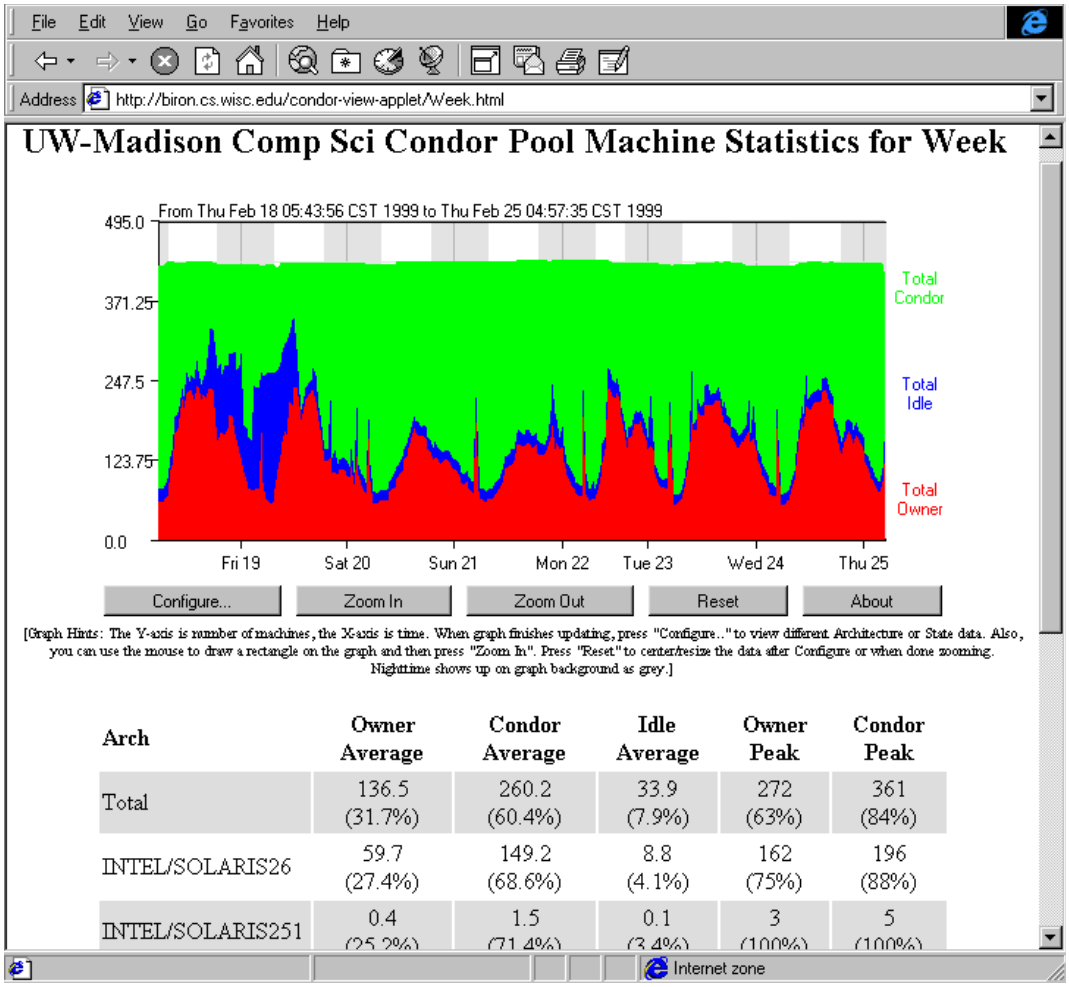

Figure 9.1: Screen shot of HTCondorView Client

3. Download the *view_client* contrib module. Follow links for contrib modules from the wiki at https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki.

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 HTCondor 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 HTCondor 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. The command

   ```
   crontab cronentries
   ```

   can set the crontab file; note that this command overwrites the current, existing crontab file with the entries from the file cronentries.

8. Point the web browser at the VIEWDIR directory to complete the installation.

9.5 Job Monitor/Log Viewer

The HTCondor Job Monitor is a Java application designed to allow users to view user log files. It is identified as the Contrib Module called HTCondor Log Viewer.

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.

9.5.1 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 HTCondor 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 does not produce a checkpoint by the HTCondor program may produce the 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.

9.5.2 Events

There are two basic kinds of events: checkpoint events and error events. Plus, advanced users can ask to see more events.

9.5.3 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.

9.5.4 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.

9.5.5 Keyboard and Mouse Shortcuts

1. The Keyboard shortcuts:
    - Arrows - an approximate ten percent scroll bar movement

- PageUp and PageDown - an approximate one hundred percent scroll bar movement
- Control + Left or Right - approximate one hundred percent scroll bar movement
- End and Home - scroll bar 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

Chapter 10

Version History and Release Notes

10.1 Introduction to HTCondor Versions

This chapter provides descriptions of what features have been added or bugs fixed for each version of HTCondor. The first section describes the HTCondor 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 HTCondor versions found in that series.

10.1.1 HTCondor Version Number Scheme

Starting with version 6.0.1, HTCondor adopted a new, hopefully easy to understand version numbering scheme. It reflects the fact that HTCondor 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 HTCondor 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 HTCondor now have exactly three numbers, separated by "."

- The first number represents the major version number, and will change very infrequently.
- *The thing that determines whether a version of HTCondor 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.

10.1.2 The Stable Release Series

People expecting the stable, production HTCondor 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 HTCondor 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 HTCondor when they appear, since they probably fix bugs you care about. Hopefully, there will not be many minor version releases for any given stable series.

10.1.3 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 HTCondor 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 HTCondor Team, should you install a development version at your site.

After the feature set of the development series is satisfactory to the HTCondor 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.

10.2 Upgrading from the 8.4 series to the 8.6 series of HTCondor

Upgrading from the 8.4 series of HTCondor to the 8.6 series will bring new features introduced in the 8.5 series of HTCondor. These new features include the following (note that this list contains only the most significant changes; a full list of changes can be found in the version history: 10.4):

- *condor_q*-related changes:
  - *condor_q* now defaults to showing only the current user's jobs. (Ticket #5271). Similarly, *condor_qedit* defaults to editing only jobs owned by the current user. (Ticket #5889). (The previous behavior of both commands can be restored by setting `CONDOR_Q_ONLY_MY_JOBS` to `False` – see 3.5.9.)
  - *condor_q* now defaults to batch mode, which produces a single line of output summarizing a batch of jobs (see 829). (Ticket #5708). (The previous behavior can be restored by setting `CONDOR_Q_DASH_BATCH_IS_DEFAULT` to `False` – see 3.5.9.)

  - *condor_q* (and *condor_history* and *condor_status*) can now read and write JSON, XML, and new ClassAd formats (see 829, 796, and 886). (Ticket #5688). (Ticket #5844). (Ticket #5820).

- Job submission-related changes:
  - Added the ability for the *condor_schedd* to transform job ClassAds upon job submission (see section 3.7.2).
  - Added the ability to group jobs into batches, and assign names to the batches, using the new **-batch** arguments to *condor_submit* and *condor_submit_dag*.
  - Added support in the submit language for retrying jobs if they fail (see 911).
- *condor_dagman*-related changes:
  - Added the ability to define SCRIPTS, VARS, etc., for all nodes in a DAG with a single command (see section 2.10.9). (Ticket #5729).
  - Simplified how DAG node priorities work (see section 2.10.9). This means that existing DAGs that use the node priority feature will run differently than they have in the past. (Ticket #4024). (Ticket #5749).
  - Added the new splice connection feature (see section 2.10.9), which allows more flexible dependencies between splices. (Ticket #5213).
- HTCondor can now use IPv6 interfaces; it prefers IPv4 if both IPv4 and IPv6 are available. (Ticket #5104).
- HTCondor now has initial support for Singularity containers (see section 3.17). (Ticket #5828).
- *condor_status* can now display a single line of output for each machine (rather than a line per slot). (Ticket #5596).
- A number of improvements to the Python bindings including: submission (Ticket #5666). (Ticket #4916).; draining (Ticket #5507).; per-thread security contexts (Ticket #5632).; Computing-On-Demand support (Ticket #5130).; and multiple query support (Ticket #5187).
- Jobs can now be submitted to the Slurm batch scheduling system via the new **slurm** type in the grid universe. (Ticket #5515).
- Numerous improvements to Docker support, including (Ticket #5680).; (Ticket #5760).; (Ticket #5761).; (Ticket #5750).; (Ticket #5740).; (Ticket #5609).; (Ticket #5456).

Upgrading from the 8.4 series of HTCondor to the 8.6 series will also introduce changes that administrators and users of sites running from an older HTCondor version should be aware of when planning an upgrade. Here is a list of items that administrators should be aware of.

- Shared port (see section 3.9.2) is now enabled by default; set `USE_SHARED_PORT` to `False` to disable it. Note that this configuration macro does not control the HAD or replication daemon's use of shared port; use `HAD_USE_SHARED_PORT` or `REPLICATION_USE_SHARED_PORT` instead. See section 3.13.2 for more details on how to configure HAD (and/or the replication daemon) to work with shared port, since just activating shared port without any other configuration change will not work. (Ticket #3813). (Ticket #5103).
- To mitigate performance problems, `LOWPORT` and `HIGHPORT` no longer restrict outbound port ranges on Windows. To re-enable this functionality, set `OUT_LOWPORT` and `OUT_HIGHPORT` (see 3.5.4 and 3.5.4). (Ticket #4711).

- Cgroups (see section 3.14.12) are now enabled by default. This means that if you have partitionable slots, jobs need to get **request_memory** correct. (Ticket #5936).
- By default, *condor_q* queries only the current user's jobs, unless the current user is a queue superuser or the `CONDOR_Q_ONLY_MY_JOBS` configuration macro is set to `False`. (Ticket #5271).
- Added support for immutable and protected job attributes, which makes SUBMIT_REQUIREMENTS more useful (see section 3.5.9). (Ticket #5065).
- By default, the *condor_schedd* no longer changes the ownership of spooled job files (they remain owned by the submitting user). (Ticket #5226).
- When `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION` is set to `True`, the related authorizations are now automatically enabled. (Ticket #5304). (See 3.5.24 for details.)
- The master can now run an administrator-defined script at shutdown; see section 3.5.7 for details. (Ticket #5590).

10.3 Stable Release Series 8.6

This is a stable release series of HTCondor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 8.6.x releases. New features will be added in the 8.7.x development series.

The details of each version are described below.

Version 8.6.10

Release Notes:

- HTCondor version 8.6.10 released on March 13, 2018.

New Features:

- None.

Bugs Fixed:

- Fixed a bug that caused *condor_preen* to crash before it finished cleaning the spool directory and leave a core file of its own in the log directory. This problem occurred on submit nodes that had running jobs when *condor_preen* was invoked. (Ticket #6521).
- Improved the systemd configuration to clean up HTCondor processes on shutdown in the event that the *condor_master* fails to do so. (Ticket #6539).

- HTCondor daemons will do fast shutdown whenever their parent process exits unexpectedly. (Ticket #6539).
- Fixed a bug that would cause *condor_q* to crash if the hostname was longer than 64 bytes. (Ticket #6594).
- Fixed a bug where if an administrator configured a Concurrency Limit whose name ended in a number, *condor_userprio* **-allusers** would show additional bogus user entries. (Ticket #6542).
- Fixed a bug where the *condor_starter* would crash when talking to a shadow running a condor version older than 8.5 and match authentication was enabled. (Ticket #6520).
- Fixed a bug in Python API *htcondor.Secman().ping()* method which would sometimes result in a RunTimeError exception. (Ticket #6562).
- Fixed a bug where `policy: want_hold_if` would always evict standard universe jobs instead of putting them on hold. Instead, this policy now ignores standard universe jobs entirely. This means that the metaknobs `policy: hold_if_memory_exceeded` and `policy: hold_if_cpus_exceeded` will also ignore standard universe jobs entirely (instead of its previous bad behavior of of letting standard universe jobs use more than their requested memory until the first time they were evicted, whereafter each restart would be immediately evicted). (Ticket #6583).
- The metaknob `policy: hold_if_memory_exceeded` and `policy: preempt_if_memory_exceeded` now ignore VM universe jobs. These jobs can't exceed their requested memory. (Ticket #6583).
- Fixed a bug which mischaracterized the `MemoryUsage` of VM universe jobs. This should allow VM universe jobs to run when `feature: Hold_If_Memory_Exceeded` is enabled. (Ticket #6577).
- Fixed a bug where the *condor_shadow* could accidentally kill itself by not checking if it was attempting to change immutable attributes. (Ticket #6557).
- Fixed a bug that could cause the *condor_collector* to exit with an assertion error under certain (rare) conditions when it has no outgoing connectivity to the Internet. (Ticket #6511).
- Fixed a bug that would cause any daemons interfacing with the CREDMON to retry indefinitely when polling for credentials. (Ticket #6523).
- Fixed a bug that prevented grid-type batch jobs from being removed after an attempt to submit to the underlying batch system failed. (Ticket #6586).
- Fixed a bug in python plugin support for the *condor_collector* that would result in the *condor_collector* switching from writing from the CollectorLog to writing to the ToolLog after a reconfig. (Ticket #6588).
- Fixed a bug in the `$F()` macro expansion in submit and configuration files that would cause a crash if the argument to the macro was a file literal rather than a variable name. (Ticket #6531).
- Fixed a bug that allowed the *condor_schedd* to attempt to run jobs on a dynamic slot that requested more resources than the slot provided. (Ticket #6593).

Version 8.6.9

Release Notes:

- HTCondor version 8.6.9 released on January 4, 2018.

New Features:

- When a daemon crashes, more information about the cause is now written to its log file. (Ticket #6483).

Bugs Fixed:

- Fixed a bug in the group quotas that would give too much surplus quota to some groups when `ACCEPT_SURPLUS` is on and `NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION` is true (the default) (Ticket #6514).
- Fixed a bug in the Python bindings when doing queries that specify a projection with the "attr_list" argument. The bug could could potentially result in memory corruption of the python interpreter process. (Ticket #6468).
- Reduced the amount of time that *condor_preen* will block the *condor_schedd*. *condor_preen* now connects only when specifically needed, and automatically disconnects after `PREEN_MAX_SCHEDD_CONNECTION_TIME` seconds. (Ticket #6490).
- Fixed a bug on Windows that would often result in the job sandbox on the execute node not being deleted when the *condor_schedd* relinquished its claim on the slot before the *condor_starter* had exited. (Ticket #6497).
- Fixed a bug where the *condor_master* stopped sending watchdog notifications to systemd after restarting itself. This resulted in systemd killing the *condor_master* shortly after the restart. (Ticket #6476).
- Updated the systemd configuration to only restart HTCondor upon failure. Otherwise, systemd would restart HTCondor if *condor_off* requested the *condor_master* to exit. (Ticket #6503).
- Fixed a bug with the use of the scheduler parameter `MAX_JOBS_SUBMITTED`. If this limit was ever reached by a submit with more than one proc in the cluster, the limit would be reduced by the difference until the *condor_schedd* was restarted. (Ticket #6460).
- Fixed a bug that caused very large RequestDisk requests to fail, and cause the Disk attribute in the machine ad to go negative. (Ticket #6467).
- Fixed a bug with the `RESERVED_DISK` parameter that would not accept an argument larger than 2 Gigabytes. (Ticket #6472).
- Improved validation of the lengths of messages in `PASSWORD` and `SSL` authentication methods. (Ticket #6493).
- Fixed a problem where the VM universe would be taken offline on the execute node, if the qcow2 disk image was corrupt. The offending job is now put on hold with an appropriate hold message. (Ticket #6505).
- Fixed a problem which would prevent Java universe jobs from working when using a relative path name to a jar file and submitting from Linux to Windows or vice versa. (Ticket #6474).

- Fixed a bug on 32 bit Linux systems that caused the starter to crash on startup if cgroup limits were enabled. (Ticket #6501).
- Fixed a bug in *Startd Cron* (see 4.4.3) where, in effect, `SlotMergeConstraint` was ignored. (Ticket #6488).
- Fixed a bug when IPv6 is enabled which could cause the *condor_startd* to crash when spawning a starter. (Ticket #6462).
- Fixed a bug in *condor_q* which could cause the DONE amount to be incorrect when multiple clusters shared a batch name. (Ticket #6469).
- Fixed issue on newer versions of Linux where core files generated by a daemon were not usable by gdb. A side effect of this fix is that the configuration parameter `CORE_FILE_NAME` no longer has any effect on Linux. (Ticket #6482).
- *condor_chirp* will now no longer abort when given a command that it cannot successfully execute, such as fetching a file that does not exist. (Ticket #6402).
- Removed unneeded `copy_to_spool` statement from default interactive submit file. (Ticket #6315).

Version 8.6.8

Release Notes:

- HTCondor version 8.6.8 released on November 14, 2017.

New Features:

- None.

Bugs Fixed:

- *Security Item*: This release of HTCondor fixes a security-related bug described at http://htcondor.org/security/vulnerabilities/HTCONDOR-2017-0001.html. (Ticket #6455).

Version 8.6.7

Release Notes:

- HTCondor version 8.6.7 released on October 31, 2017.

New Features:

- Added support for HTTPS transfers in the `curl_plugin` utility. (Ticket #6253).
- Job attributes that are recognized by the *batch_gahp* but not by HTCondor can now be specified in the job ad without using a prefix of `Remote_`. (Ticket #6422).

Bugs Fixed:

- Fixed a bug that caused systems using cgroup memory limits to not properly reset the memory limit after the first use of a slot. The memory limit would get reused from the previous slot value. (Ticket #6414).
- Added SELinux type enforcement rules to allow *condor_ssh_to_job* to function on Enterprise Linux 7. (Ticket #6362).
- Asking systemd to stop condor now allows the HTCondor daemons to properly clean up, instead of simply immediately sending a SIGKILL. As a result, HTCondor daemons stopped via systemd will no longer continue to appear alive with *condor_status*. (Ticket #6096).
- Fixed problems in python bindings when using the Submit class to submit jobs specifying environment variables or file redirection. (Ticket #6420).
- Change the default value of STARTD_RECOMPUTE_DISK_FREE to false, so that the Disk attribute is mostly correct for partitionable slots. (Ticket #6424).
- Docker universe now sets the cgroup cpu-shares field to 100 times the number of requested cores, which matches vanilla universe. (Ticket #6423).
- MOUNT_UNDER_SCRATCH when used in Docker universe can now be an expression, not just a literal string. This matches the way it works in vanilla universe. (Ticket #6401).
- Fixed a bug that could cause the *condor_startd* to crash when spawning a *condor_starter* with mixed mode networking. (Ticket #6461).
- Fixed a bug that caused the *condor_collector* on Windows to refuse connections whenever the number of open sockets was more than 820 even though space was allocated for 1024 open sockets. (Ticket #6425).
- Fixed a bug that caused the configuration variable `DEFAULT_MASTER_SHUTDOWN_SCRIPT` to be ignored on Windows when the *condor_master* was running as a service. (Ticket #6458).
- *condor_status* will now print longer lines when its output is redirected into a pipe, rather than its input coming from one. (Ticket #6381).
- Fixed a crash in *condor_transferer* when a connection can't be established with its peer. (Ticket #6412).
- Fixed a bug that caused *condor_job_router_info* to crash if configuration parameter `JOB_ROUTER_ENTRIES_REFRESH` was set to a positive value. (Ticket #6444).
- Fixed a bug in *condor_history* that caused it to print invalid XML or JSON syntax when reading from multiple history files. (Ticket #6437).
- Fixed a bug in the *condor_schedd* which resulted in the `IsNoopJob` job attribute sometimes being ignored if the the value of this attribute was changed after the job was submitted. (Ticket #6396).

- Fixed a bug that rarely caused slurm jobs to be held. When slurm reports memory utilization and it is a multiple of 1024k, Slurm uses the 'M' suffix. The parsing logic was extended to also interpret the 'M', 'G', 'T', and 'P' suffixes for memory utilization. (Ticket #6431).
- The condor-bosco RPM ensures the *rsync* is installed as required by the Bosco scripts. (Ticket #6439).
- To avoid unnecessary transfers when `copy_to_spool` is set in the submit file, HTCondor no longer copies the executable to the local spool directory more than once for a cluster. (Ticket #6454).

Version 8.6.6

Release Notes:

- HTCondor version 8.6.6 released on September 12, 2017.

New Features:

- None.

Bugs Fixed:

- Fixed a bug that might cause the *condor_schedd* or other daemons to crash when logging on Linux to the syslog facility, and the *condor_reconfig* command was run. (Ticket #6364).
- Fixed a bug that prevented condor daemons from writing out a core file for debugging in the very unlikely event that one of them crashed. (Ticket #6365).
- Fixed a bug where the negotiator would make matches where the daemons involved did not share an IP version, and thus could not talk to each other. (Ticket #6351).
- HTCondor now works properly with systemd's watchdog feature on all flavors of Linux. Previously, the *condor_master* wouldn't send alive messages to systemd if systemd wasn't part of the Linux distribution's standard configuration. This would result in systemd killing the HTCondor daemons after a short period of time. (Ticket #6385).
- Fixed handling of backslashes in string values in old ClassAds format in the python bindings. (Ticket #6382).
- Fixed a bug in how the CPU usage of Slurm jobs is interpreted. (Ticket #6380).
- Fixed a bug that caused a machine claimed by a parallel universe job to stick in the Claimed/Idle state forever. This could only happen if the job was removed as it was in the process of claiming resources. (Ticket #6376).
- Fixed a bug that caused a machine to stick in the Preempting/Vacating state after a job was removed when a `JOB_EXIT_HOOK` was defined. (Ticket #6383).
- Added type enforcement rules for cgroups to HTCondor's SELinux profile. (Ticket #6168).

- Fixed a bug where setting `delegate_job_gsi_credentials_lifetime` to 0 in a submit description file was treated the same as not setting it at all. (Ticket #6375).
- Fixed handling of octal escape sequences in ClassAd strings. (Ticket #6384).
- Updated Boost external to version 1.64. (Ticket #6369).

Version 8.6.5

Release Notes:

- HTCondor version 8.6.5 released on August 1, 2017.

New Features:

- Added avx2 to the set of processor flags advertised by the *condor_startd*. (Ticket #6317).

Bugs Fixed:

- Fixed a bug in socket clean-up that was causing a memory leak. This may have been particularly noticeable in the *condor_collector*. (Ticket #6342).
- Fixed a bug that caused an infinite loop in the *condor_starter* when cgroups were enabled on systems (such as Debian) where the kernel has disabled the memory accounting controller. A job on such a system would go into the "R" state, but never actually start running. (Ticket #6347).
- Fixed a bug where setting `NETWORK_INTERFACE` to an IPv6 address could cause HTCondor daemons to except. (Ticket #6339).
- Fixed a bug where a cross protocol CCB connection would cause the *condor_shadow* or *condor_schedd* to except. (Ticket #6344).
- Fixed a bug where the wildcard '*' in ALLOW or DENY lists was being interpreted as only matching IPv4 addresses. It now properly matches any address family. (Ticket #6340).
- Fixed a bug where reverse resolutions could return the string representation of the address in question instead of failing. This resulted in spurious warnings of the form "WARNING: forward resolution of 2001:630:10:f001::19a0 doesn't match 2001:630:10:f001::19a0!" (Ticket #6338).
- Fixed a bug which prevented using an ImDisk RAM disk as the execute directory on Windows. (Ticket #6324).
- Fixed a bug where removal of a job could cause another job from the same user to also be removed. This was mostly likely to happen when the *condor_schedd* is under heavy load. (Ticket #6353).
- Fixed a bug that cause parallel universe jobs not to start on pools with partitionable slots. (Ticket #6308).

- Fixed a problem, introduced in HTCondor 8.6.4, where the *condor_collector* plugins where loaded but not used. (Ticket #6343).
- Fixed a bug where "*condor_q* **-grid**" did not display the status column for any non-Globus job. (Ticket #6306).
- Fixed bugs in the *condor_schedd* and *condor_negotiator* that could cause jobs to not be negotiated for when `NEGOTIATOR_PREFETCH_REQUESTS` is set to `TRUE`. (Ticket #6336). (Ticket #6312).

Version 8.6.4

Release Notes:

- HTCondor version 8.6.4 released on June 22, 2017.

New Features:

- Python bindings are now available on MacOSX. (Ticket #6244).
- Allow Python modules to be used as *condor_collector* plugin. This undocumented feature is to be used by expert developers only. (Ticket #6213). (Ticket #6295).

Bugs Fixed:

- Fixed a bug with PASSWORD authentication that would sporadically cause it to fail to exchange keys, due to whether or not the first round-trip of communications blocked on reading from the network. (Ticket #6265).
- Pslot preemption now properly handles machine custom resources, such as GPUs. (Ticket #6297).
- Fixed a bug that prevented HTCondor from correctly setting virtual memory cgroup limits when soft physical memory limits were being used. (Ticket #6294).
- Fixed a bug that prevented parallel universe jobs from running that used $$() expansion in submit files. (Ticket #6299).
- Added a new knob, `STARTD_RECOMPUTE_DISK_FREE`, which defaults to true, which tells the startd to periodically recompute and advertise free disk space. Admins can set this to false for partitionable slots whose execute directory is used by HTCondor alone. (Ticket #6301).
- Fixed a bug that could cause *condor_submit* to fail to submit a job with a proxy file to a *condor_schedd* older than 8.5.8, due to the absence of an X.509 CA certificates directory. (Ticket #6258).
- Restored a check in *condor_submit* about whether the job's X.509 proxy has sufficient lifetime remaining. (Ticket #6283).
- Fixed a bug in *condor_dagman* where the DAG status file showed an incorrect status code if submit attempts failed for the final node. (Ticket #6069).

- Bosco now properly identifies CentOS 7 as a supported platform. (Ticket #6303).
- Fixed a bug when Bosco is used to submit jobs to multiple remote clusters. When arguments to *remote_gahp* are provided in the GridResource attribute, jobs could be submitted to the wrong cluster. (Ticket #6277).
- To speed up the installation process on Enterprise Linux 7, the SELinux profile is now reloaded only once, when setting the HTCondor daemons to run in permissive mode. (Ticket #6304).
- Update the systemd configuration on Enterprise Linux 7 to start the *condor_master* after time synchronization is achieved. This prevents unnecessary daemon restarts due to sudden time shifts. (Ticket #6255).
- The *condor_shadow* will now ignore updates of `JobStartDate` from the *condor_starter* since the *condor_schedd* already sets this attribute correctly and the *condor_starter* incorrectly tries to set it even if the job has already run once. A consequence of this fix is that the value of `JobStartDate` that the *condor_startd* uses for policy expressions will be different than the value that the *condor_schedd* uses. Resolving this problem will potentially break existing policy expressions in the *condor_startd*, so it will be be not be changed in the 8.6 series, but fixed in the 8.7 series. (Ticket #6280).
- Fixed a bug where per-instance job attributes like `RemoteHost` would show up in the history file for completed jobs. This bug occurred if a job happened to complete while the *condor_schedd* was in the process of a graceful shutdown. (Ticket #6251).
- The *condor_convert_history* command is present again in this release. (Ticket #6282).
- The parameter `SETTABLE_ATTRS_ADMINISTRATOR` is now correctly appears in *condor_config_val*. (Ticket #6286).

Version 8.6.3

Release Notes:

- HTCondor version 8.6.3 released on May 9, 2017.

Bugs Fixed:

- Fixed a bug that rarely corrupts the *condor_schedd*'s job queue log file when the input sandbox of a job with an X.509 proxy file is spooled. (Ticket #6240).
- Fixed a memory leak in the Python bindings related to logging. (Ticket #6227).

Version 8.6.2

Release Notes:

- HTCondor version 8.6.2 released on April 24, 2017.

New Features:

- Added metaknobs for defining map files for use with the ClassAd usermap function in the *condor_schedd*, and a metaknob for automatically assigning an accounting group to a job based on a mapping of the owner name of the job. (Ticket #6179).
- When the *condor_credd* is polling for credentials, the timeout is now configurable using `CREDD_POLLING_TIMEOUT`.
- The **reverse** option for *condor_q* was changed to **reverse-analyze**, and it now implies **better-analyze**. Formerly, the **reverse** option was ignored unless **-better-analyze** was also specified. (Ticket #6167).

Bugs Fixed:

- Fixed a bug that could cause *condor_store_cred* to fail on Windows due to a case-sensitive check of the user's account name. (Ticket #6200).
- Updated Open MPI helper script to catch and handle SIGTERM and to use bash explicitly. (Ticket #6194).
- Docker Universe jobs now update the RemoteSysCpu attributes for job and in the job log. Previously, this field was always 0. (Ticket #6173).
- Docker universe detection is now more robust in the face of extraneous output to standard error on docker startup. This was preventing Condor from detecting that docker was properly working on hosts. (Ticket #6185).
- Fixed a bug that prevented `SUBMIT_REQUIREMENT` and `JOB_TRANSFORM` expressions from referencing job attributes describing the job's X.509 proxy credential. (Ticket #6188).
- The Linux kernel tuning script no longer adjusts some kernel parameters unless a *condor_schedd* will be started by the master. (Ticket #6208).
- Fixed a bug that caused all but the first in a list of metaknobs to be ignored unless there were commas separating the list items. So `use ROLE : Execute CentralManager` would incorrectly add only the Execute role. Previously, `use ROLE : Execute, CentralManager` would correctly add both roles. (Ticket #6171).
- Worked around a problem with FORTRAN programs built with *condor_compile* and recent versions of gfortran (4.7.2 was OK, 4.8.5 was not), where those executables would not write to standard out if started in the standard universe. Also, updated the checkpointing library to permit *condor_compile* to successfully link FORTRAN (and other) programs calling certain math functions and built against up-to-date versions of glibc. (Ticket #6026).
- The default values for `HAD_SOCKET_NAME` and `REPLICATION_SOCKET_NAME` have changed to enable the documented configuration for using these services with shared port to work. (Ticket #6186).
- Fixed a bug that caused *condor_dagman* to sometimes (rarely, but repeatably) crash when parsing DAGs containing splices. (Ticket #6170).
- The configuration parameters that control when job policy expressions are evaluated now work as documented. Previously, the default value for `PERIODIC_EXPR_INTERVAL` was 300, not 60 as intended. Also, the parameters `MAX_PERIODIC_EXPR_INTERVAL` and `PERIODIC_EXPR_TIMESLICE` were ignored for grid universe jobs. (Ticket #6199).

- Fixed a bug that could cause the Job Router to crash if the `job_queue.log` contained invalid or incomplete records. (Ticket #6195).
- Fixed a bug that caused updates of the job attribute `x509UserProxyExpiration` to be ignored for job policy evaluation when the job was managed by the Job Router. (Ticket #6209).
- Changed the default value of configuration parameters `CREAM_GAHP_WORKER_THREADS` to the value of `GRIDMANAGER_MAX_PENDING_REQUESTS`. This should prevent a back-log of commands in the CREAM GAHP observed by some users. (Ticket #6071).
- Fixed modification of `PYTHONPATH` environment variable that could fail in bash if *set -u* is enabled. (Ticket #6211).
- *bosco_quickstart* no longer assumes that submitting to a Slurm cluster requires the PBS emulation module. (Ticket #6211).
- Fixed a bug that caused *condor_submit* **-dump** to crash when the submit file had an attribute to enable the use of an x509 user proxy. (Ticket #6197).
- Updated the supported platform list in the Bosco installer script to include Ubuntu 16 and Mac OSX 10.12. Also, dropped Ubuntu 12 and Mac OSX 10.6 through 10.9. (Ticket #6178).
- Fixed a bug which in some obscure configurations caused a spurious PERMISSION DENIED error was printed in the StartLog when activating a claim. (Ticket #6172)..
- Fixed a bug which forced the administrator to restart (rather than reconfigure) running daemons after adding an entry to a `DENY_*` authorization list. (Ticket #6172)..

Version 8.6.1

Release Notes:

- HTCondor version 8.6.1 released on March 2, 2017.

New Features:

- *condor_q* now checks to see if authentication and security negotiation are enabled before attempting to request only the current users jobs from the *condor_schedd*. Prior to this change, configurations that disabled security or authentication would also need to set `CONDOR_Q_ONLY_MY_JOBS` to false. (Ticket #6125).
- The CLAIMTOBE authentication method is now in the list of methods for READ access if no list of authentication methods for READ or DEFAULT is specified in the configuration. This change allows sites that use the default host based security model to use *condor_q* **-global** with the only-my-jobs feature without making changes to their security configuration. (Ticket #6125).
- The collector now records the authentication method used to determine the authenticated identity. (Ticket #6122).

Bugs Fixed:

- Update Docker interface to be able to retrieve usage information from running containers and to remove containers when certain errors occurred when using Docker version 1.13. (Ticket #6088).
- In Docker universe, all writes to files in `/tmp` and `/var/tmp` by default write inside the container. There is a limit on the file size within the container, and jobs that write a lot to `/tmp` may hit that. If a docker universe job now runs on a system with `MOUNT_UNDER_SCRATCH` defined, HTCondor now adds those mounts as volume mounts, so file writes do not go to the container, but to the host file system. (Ticket #6080).
- Fixed a bug in *condor_status* **-format** and *condor_q* **-format** that caused the tools to truncate output to the width specified in the format specifier. The most likely manifestation of this bug was that punctuation after the format would not be printed when the format had an explicit width. (Ticket #6120).
- Fixed a bug that caused spurious shared port-related error messages to appear in the `dagman.out` file (by adding the new `DAGMAN_USE_SHARED_PORT` configuration macro). (Ticket #6156).
- Fixed a bug that caused VM universe jobs to fail if the **vm_disk** submit command contained spaces after a comma. (Ticket #6132).
- Fixed a bug that can cause the Job Router and *condor_c-gahp* to crash if they fail to submit a job due to submit transforms or submit requirements. (Ticket #6152).
- Fixed a bug that caused the Job Router to not route any jobs if the `JOB_ROUTER_DEFAULTS` configuration parameter value started with white space. (Ticket #6128).
- Fixed several bugs in how the Job Router writes to job event logs. (Ticket #6092).
- Removed Bosco's attempt to configure a default value for **grid_resource** in the submit description file, as *condor_submit* no longer supports this ability. Also, Bosco now works with Slurm clusters. (Ticket #6106).
- Changed Bosco's configuration of the *condor_ft-gahp* to eliminate worrying error messages in the *condor_ft-gahp*'s log file. (Ticket #6107).
- Fixed a bug that could cause a grid batch job submitted to PBS or Slurm to go on hold when the job's X.509 proxy is refreshed. (Ticket #6136).
- Fixed a bug where the *condor_gridmanager* fails to put a job on hold due to the desired hold reason containing invalid characters. (Ticket #6142).
- Improved the hold reason when submission of a grid-type batch job fails. (Ticket #3377).
- Update helper scripts to work with current versions of Open MPI and MPICH2. (Ticket #6024).
- Fixes a bug that could cause events for local universe jobs to not be written to the global event log. (Ticket #6100).
- Fixed a bug on execute machines that enable PID namespaces that would generate a spurious error message in the daemon log when *condor_off* -fast was issued. (Ticket #6137).
- Fixed a bug that could corrupt the job queue log file such that the *condor_schedd* cannot restart. The bug is mostly likely to occur if the disk becomes full. (Ticket #6153).

- Incremented the ClassAd library version number, since the deprecated iostream interface has been removed. (Ticket #6050). (Ticket #6115).

Version 8.6.0

Release Notes:

- HTCondor version 8.6.0 released on January 26, 2017.

New Features:

- Added two new job ClassAd attributes, `CumulativeRemoteSysCpu` and `CumulativeRemoteUserCpu`, which keep a running total of system and user CPU usage, respectively, across all job restarts. Also, immediately clear attributes `RemoteSysCpu` and `RemoveUserCpu` on job start, instead of on first update. (Ticket #6022).
- Added a new configuration knob, `ALWAYS_REUSEADDR`, which defaults to `True`. When `True`, it tells HTCondor to set the `SO_REUSEADDR` socket option, so that the schedd can run large numbers of very short jobs without exhausting the number of local ports needed for shadows. (Ticket #6040).
- Changed the default value of `IGNORE_LEAF_OOM` to `True`. (Ticket #5775).

Bugs Fixed:

- Fixed a bug causing unnecessarily slow updates from the *condor_startd*. If you depend on the old behavior, set `UPDATE_SPREAD_TIME` to 8. A value of 0 enables the fix. (Ticket #6062).
- Fixed a race condition when running multiple concurrent jobs on the same claim. When the starter exits, it notifies the shadow, which tells the startd to kill the starter. Immediately after the shadows tells the startd, it fetches the next job, and tries to start it. If the starter hasn't completely exited yet (perhaps it needs to clean up a large sandbox), it will notice the shadow has closed the command socket, and the starter will go into disconnected mode, and get confused. This has been fixed. (Ticket #6049).
- Fixed an infelicity with *condor_submit* -i and docker universe, where it would start an interactive shell without a container. Added error message expressing that this combination is not currently supported. (Ticket #6083).
- When a job claimed by the Job Router is held or removed, it is no longer considered a failure of the job route chosen for that job. (Ticket #5968).
- Fixed a bug in recovering a Google Compute Engine (GCE) job if the *condor_gridmanager* restarts during submission of the instance request. (Ticket #6078).
- Fixed a bug that could cause re-installation of a remote cluster to fail in Bosco. (Ticket #6042).
- Fixed a bug with handling the proxy files of grid-type batch jobs when the proxy's file name is a relative path. (Ticket #6053).

- Fixed a bug that caused the *batch_gahp* to crash when a job's X.509 proxy is refreshed and the *batch_gahp* is configured to not create a limited copy of the proxy. (Ticket #6051).
- Fixed a bug in the virtual machine universe where `RequestMemory` and `RequestCPUs` were not changing the resources assigned to the VM created by HTCondor. Now, `VM_Memory` defaults to `RequestMemory`, and the number of CPUs defaults to `RequestCPUs`. (Ticket #5998).

10.4 Development Release Series 8.5

This is the development release series of HTCondor. The details of each version are described below.

Version 8.5.8

Release Notes:

- HTCondor version 8.5.8 released on December 13, 2016.

New Features:

- On Linux, the starter now puts all jobs in a cgroup by default. The default for CGROUP_MEMORY_LIMIT_POLICY is now "none". To disable cgroups, an admin can set the BASE_CGROUP parameter to the empty string. (Ticket #5936).
- Added first-class *condor_submit* commands supporting job retries. (See section 11 for details.) (Ticket #5912).
- *condor_qedit* now defaults to editing only jobs owned by the current user in the same way that *condor_q* does. It also honors the `CONDOR_Q_ONLY_MY_JOBS` configuration variable. (Ticket #5889).
- Added new parameter `DOCKER_VOLUME_DIR_XXX_MOUNT_IF` which is an expression, evaluated in the context of the machine and job ad, which if it evaluates to a string, becomes a docker volume mount. This allows admins to conditionally add docker volumes for certain types of jobs. (Ticket #5758).
- Added initial support for Singularity containers. (Ticket #5828).
- The XferStatsLog file on the submit side now contains TCP statistics for both the shadow point of view, and the starter point of view. The starter side line is prefixed with the words "peer stats from starter". (Ticket #5917).
- Configuration variables of the form `SUBSYS.LOCALNAME.VARIABLE` no longer work. The use of the SUBSYS prefix before LOCALNAME never worked fully, and was only necessary for while as a workaround for a bug that was fixed many years ago. *condor_config_val* and the *condor_master* will now produce warning messages when the configuration has variables that appear to of this form and begin with a known SUBSYS name like MASTER or COLLECTOR. (Ticket #5969).
- The `SLOT_WEIGHT` parameter can now be set on the central manager, instead of all the execute nodes. If the execute nodes set this parameter, it will override the central manager setting. (Ticket #5953).

- New submit command **gce_json_file** can be used with grid-type gce jobs to specify a file that contains JSON object members that should be added to the instance description submitted to the GCE service. (Ticket #5893).
- A number of command-line tools now support bash auto-completion. (Ticket #5924).
- The minimum update time for *condor_dagman* node status files now defaults to 60 seconds. (Ticket #5929).
- Added the new `DAGMAN_REMOVE_NODE_JOBS` configuration macro, which allows users to configure whether *condor_dagman* itself removes its node jobs when it is removed (note that the node jobs are also removed by the *condor_schedd*). This configuration macro defaults to `True`, which represents a change in behavior compared to previous HTCondor versions. (See section 2.10.7 for more details.) (Ticket #5175).
- The *-AllowLogError* argument to *condor_submit_dag* and *condor_dagman*, and the DAGMAN_ALLOW_LOG_ERROR configuration macro, are no longer supported, and generate warnings if used. (Ticket #5630).
- *condor_dagman* now ignores the `DAGMAN_LOG_ON_NFS_IS_ERROR` configuration setting if `ENABLE_USERLOG_LOCKING` is set to `False`. (Ticket #5641).
- Added the *ALL_NODES* option to a number of *condor_dagman* commands (see 2.10.9 for details). (Ticket #5729).
- Changed the previous term "metaknob" to "configuration template" and improved the configuration template documentation. (Ticket #5865).
- The *condor_schedd* receiving a refreshed X.509 proxy credential is now done in a non-blocking fashion. (Ticket #5930).
- The Job Router now performs its automatic job ad transformations when the TRANSLATE_JOB hook is used. These are changes that should happen to all job ads being transformed by the Job Router. (Ticket #5235).
- The `$F()` configuration macro has new options to support conversions of paths to Windows style path separators or to Unix style. When used in *condor_submit* files it can do path completion as well. (Ticket #5938).
- The `$ENV()` configuration macro now supports default values. (Ticket #5882).
- A certificate mapfile can now use literal values rather than regular expressions for the second field. This is useful when only a single identity should be matched. The use of a literal is both more secure and faster to search. The new configuration variable `CERTIFICATE_MAPFILE_ASSUME_HASH_KEYS` enables this behavior, it defaults to false. It will most likely default to true in a future version of HTCondor. (Ticket #5992).
- The ClassAd `userMap` function now uses only commas as the separator for the third field of the map file. This makes it possible to have values with spaces in them. (Ticket #5988).
- The *condor_collector* will now allow more than one *condor_negotiator* to be registered. And a new A new configuration variable `COLLECTOR_ALLOW_ONLY_ONE_NEGOTIATOR`, which defaults to false has been added so that the old behavior can still be configured. (Ticket #5967).
- The Requirements expression for Job transforms in the *condor_schedd* will now ignore the TARGET prefix for attributes in the expression. This makes it easier to convert *condor_job_router* rules to job transforms because the TARGET prefix is required in the *condor_job_router* but refers to nothing in the job transform. (Ticket #5980).

- The **-better-analyze** option of *condor_q* has been improved and the output reorganized. (Ticket #5290).
- A new tool - *condor_transform_ads* has been added. (See section 11 for details.) (Ticket #5805).
- A join function has been added to the ClassAd language. (Ticket #6018).
- *condor_who* has additional options for querying the state and readiness of the various daemons. It has a command that can be used to wait for the daemons to startup with a timeout. (Ticket #5416).
- When submitting a job that has an associated X.509 proxy, or when authenticating to the *condor_schedd* using X.509, the X.509 and VOMS attributes are securely extracted and carried along in the job ClassAd. This allows them to be used, for example, in matchmaking policy and job routing. (Ticket #5064).
- Made *condor_credd* configuration easier by automatically configuring network connections to use encryption.

Bugs Fixed:

- When the Google Compute Engine breaks the results of a query into multiple pages, the *gce_gahp* now retrieves all of the results, instead of just the first page. (Ticket #6010).
- Fixed a bug that caused file transfer to fail when a job created by the Job Router has a different `Owner` than the original job. (Ticket #5348).
- Fixed a bug that could result in "orphan" node jobs staying in the queue when an instance of *condor_dagman* is removed. (Ticket #5702).
- Fixed a regression introduced in v8.5.7 that prevents job preemption due to priority from occurring, because user priority and resources in use information cannot be referenced in `PREEMPTION_REQUIREMENTS`. (Ticket #6014).
- Fixed `COLLECTOR_FORWARD_FILTERING` so that a startd ad update is always forwarded when any of the Claim IDs change. (Ticket #5913).
- Fixed a bug that made the Requirements keyword for job transforms in the *condor_schedd* only work if it was all uppercase on Red Hat 7 and some other platforms that use a newer version of the C++ compiler. (Ticket #5973).
- Fixed a bug that allowed a user to bypass the `MAX_RUNNING_SCHEDULER_JOBS_PER_OWNER` limit by specifying an accounting group or nice_user in their submit file. (Ticket #5949).
- Fixed a bug in *condor_c* and the *condor_job_router* that could cause inaccurate job totals to be reported by *condor_q* **-batch**. (Ticket #6020).

Version 8.5.7

Release Notes:

- HTCondor version 8.5.7 released on September 29, 2016.

Known Issues:

- Preemption due to job priority is likely to fail if PREEMPTION_REQUIREMENTS attempts to reference any resource usage or priority attributes. This issue has been fixed in v8.5.8. If you cannot upgrade to v8.5.8, a work-around for v8.5.7 is to set configuration macro NEGOTIATOR_CROSS_SLOT_PERI'S to True. (Ticket #6014).

New Features:

- Added the capability for the schedd to perform job ClassAd transformations upon job submission (see 3.7.2 for details). (Ticket #5885).
- Added the capability for more flexible connections between splices in DAGs (see 2.10.9 for details). Also added an INCLUDE command to the DAG language (see 2.10.9 for details). (Ticket #5213).
- Simplified the DAG node priority algorithm: the "effective" priority of a node is now simply the sum of the explicit node priority and the overall DAG priority. (See section 2.10.9 for more details.) (Ticket #4024). (Ticket #5749).
- Allow the second argument of the ClassAd ternary operator (expression ? value1 : value2) to be omitted. This new syntax means: evaluate the expression, and if it evaluated to a defined value or error, return it. If undefined, return value2. (Ticket #5782).
- The time is now included after the SCHEDD or SUBMITTER name in the banner of *condor_q* output. (Ticket #5895).
- *condor_status* has a new **-data** option that, when used with **-schedd** will show data transfer information; and **-run** will show information about running jobs when used with **-schedd**. (Ticket #3938).
- *condor_q* **-batch** will now show Total and Completed counts for non-DAG jobs when querying a scheduler that is at least version 8.5.7 (Ticket #5874).
- *condor_status* and *condor_q* now support reading and writing ClassAds in xml, json, and "new ClassAd" form as well as the traditional long form. (Ticket #5844). (Ticket #5820).
- HTCondor daemons now respect <LOCALNAME>.<SUBSYSTEM>_LOG if passed a -local-name parameter, and default to using $(LOG)/<Localname>Log if the former is not set. (Ticket #5768).
- HTCondor now automatically passes the -local-name parameter to a DC daemon if its entry in the DAEMON_LIST is not in the default DC_DAEMON_LIST. This should result in simpler and less error-prone configuration. (Ticket #5768).
- HTCondor now detects if an entry in DAEMON_LIST shares a binary with an entry in DC_DAEMON_LIST and marks the former as a DC daemon if so. This should result in simpler and less error-prone configuration. (Ticket #5767).
- Increase the resolution of file transfer timing statistics in the XferStatsLog to hundreds of a second. (Ticket #5898).

- The default host based security meta-knob now works in IPv6 only networks out of the box. (Ticket #5894).
- Old HAD configurations, with or without replication, should now work by default (without shared port). (Ticket #5769).
- HTCondor no longer gives up if a bad networking configuration is detected while running a tool. This allows *condor_config_val* to be used to debug the problem. (Ticket #5532).
- The *condor_negotiator* by default no longer cross advertises the user priority and resources in use from every slot in a machine ad to every other slot in that machine ad. `NEGOTIATOR_CROSS_SLOT_PRIOS` = true re-enables the old behavior. The accounting information for the current user of the slot remains advertised. (Ticket #5785).
- New submit attribute **gce_preemptible** allows the creation of preemptible Google Compute Engine (GCE) instances. These instances have a lower price, but can be interrupted at any time. Also added support for service accounts with GCE. (Ticket #5821).
- When submitting jobs to Slurm via the grid universe, the Slurm partition can now be specified using the **batch_queue** submit command. (Ticket #5780).
- Some old STARTD policy helper configuration variables were moved into two new configuration templates – `FEATURE : UWCS_DESKTOP_POLICY_VALUES` and `FEATURE : TESTINGMODE_POLICY_VALUES` (Ticket #5871).
- *condor_submit* on Windows will no longer insert the OSVERSIONINFO fields like `WindowsMajorVersion` into each job automatically. This is controlled by a new configuration variable `SUBMIT_PUBLISH_WINDOWS_OSVERSIONINFO` which defaults to false. (Ticket #5873).
- Added the option to cache the output of commands used in configuration files, so that the command doesn't have to be re-run every time the configuration file is referenced. Also added error and warning keywords to allow configuration files to report errors and warnings. (Ticket #5781).

Bugs Fixed:

- Fixed a bug in how the HAD daemon checks to see if it and its corresponding replication daemon were configured to be on the same host. (Ticket #5849).
- The EC2 GAHP now handles integer overflows when checking deadlines. This prevents spurious time-outs on 32-bit systems which have been up for more than 28 days. (Ticket #5824).
- Lengthened the watchdog timeout in the systemd service file to 20 minutes. Also, ping systemd at a third of the watchdog interval. (Ticket #5837).
- Fixed a bug that could cause daemons to create a file named `dprintf_failure.SUBSYS` if they failed to find the *mail* program. (Ticket #5854).
- For grid-type `batch` jobs, improved handling of command line arguments and environment variables that contain characters that have meaning to the shell. Previously, the presence of these characters would cause job execution to fail. (Ticket #5747).

- Fixed a bug that caused *condor_config_val* to segfault when the **-name** argument was used and the machine did not exist (Ticket #5818).
- Fixed a bug that caused *condor_q* **-autocluster** to crash unless the **-nobatch** option was also used. (Ticket #5839).
- Fixed a bug in the Python bindings where a thread executed python byte code without holding the global interpreter lock. (Ticket #5864).

Version 8.5.6

Release Notes:

- HTCondor version 8.5.6 released on August 2, 2016.

New Features:

- The default output of *condor_q* is now the **-batch** output. To change the default back to its pre-8.5.6 value, set the new configuration variable `CONDOR_Q_DASH_BATCH_IS_DEFAULT` to `False`. (Ticket #5708).
- A new class – the Submit class – was added to the Python bindings. It allows for the submission of HTCondor jobs via the Python bindings using the same keywords and automatic behavior as *condor_submit*. See section 6.7.1 for details. (Ticket #5666).
- The ability to send *condor_drain* commands is now exposed through the Python bindings. See section 6.7.1 for details. (Ticket #5507).
- The value of the configuration parameter `DOCKER_DROP_ALL_CAPABILITIES` is now no longer just true or false, but a ClassAd expression evaluated in the context of the machine (my) and the job (target). (Ticket #5759).
- When running Docker Universe containers on docker version 1.11 and newer, HTCondor now also sets –no-new-privs, to prevent setuid and setgid programs from running in containers, unless `DOCKER_DROP_ALL_CAPABILITIES` evaluated to false. (Ticket #5680).
- The hostname of the container that Docker Universe jobs run in is now set to a more useful name. Instead of a hash, it now contains the job's owner, the cluster and proc of the job, and the hostname of the machine the container runs on. (Ticket #5760).
- New options have been added to *condor_history*, so that *condor_history* can be used as the the `HISTORY_HELPER` for remote *condor_history*. The options are:
  - **-since** Scanning of the history file stops when an expression becomes true or a job id is read.
  - **-completedsince** Scanning of the history file stops when a job completed earlier than this time is read.
  - **-scanlimit** Used by remote *condor_history* to limit the number of jobs read from the history file.
  - **-attributes** Used by remote *condor_history* to limit the attributes transferred back.

  - **-inherit** Used by remote *condor_history* to define the socket to write results to.
  - **-stream-results** Used by remote *condor_history* so that results can be printed as they arrive.

  (Ticket #5642).

- Condorhistory will default to doing a remote query if there is a `SCHEDD_HOST` configured. This behavior can be defeated by passing the new **-local** argument. (Ticket #5765).
- The high-availability and replication daemons may now use shared port. (Ticket #5726).
- ClassAds can now be represented in JSON format. condor_q, condor_status, and condor_history have a *-json* command line option, which causes their output to be printed in JSON. (Ticket #5688).
- *condor_dagman* now allows commands to be more flexibly ordered within a DAG file. (See section 2.10.3 for details.) (Ticket #5732).
- Any **accounting_group** and **accounting_group_user** values specified for a DAG are now propagated to all jobs of the workflow, including sub-DAGs. (Ticket #5077).
- A new configuration variable `MAX_RUNNING_SCHEDULER_JOBS_PER_OWNER` can be used to limit the number of DAGs that any single user can have running at a time. (Ticket #5568).
- Monitoring the status of PBS and Slurm jobs is now much more efficient. Now, one query to the batch system is done for all jobs, instead of a separate query for each job. (Ticket #5722).
- Simplified how job leases are handled for grid universe jobs. Now, all jobs going to the same remote resource share a single lease time. (Ticket #5625).
- Added several statistics about commands issued to the GAHP server to the grid ads that the *condor_gridmanager* sends to the *condor_collector*:

  - GahpCommandsIssued
  - GahpCommandsTimedOut
  - GahpCommandsInFlight
  - GahpCommandsQueued
  - GahpCommandRuntime

  (Ticket #5698).

- The *condor_shadow*, *condor_starter* and *condor_c-gahp* daemons now log TCP statistics for file transfers. See 3.5.2 for more details. (Ticket #5663).
- Job ads now include `NumJobCompletions`, which counts the number of times a job exited of its own accord (successfully or not) and then successfully completed file transfer (if any was requested). (Ticket #5705).
- Kerberos authentication is now non-blocking, allowing an HTCondor daemon authenticating clients with Kerberos to handle more simultaneous incoming connections. (Ticket #5737).
- Password authentication is now non-blocking, allowing an HTCondor daemon authenticating clients with the PASSWORD method to handle more simultaneous incoming connections. (Ticket #5602).

- The full path to the submit file is now available as an automatic submit variable. (Ticket #5677).
- A new function `userMap()` has been added to the ClassAd language to facilitate the mapping of users to groups in the *condor_schedd* and *condor_job_router* (see 4.1.2 for details). (Ticket #5751).
- Configuration files now support the declaration of multi-line values, the is primarily of use when configuring the *condor_job_router*. (Ticket #5721).
- Configuration templates can now take arguments. (Ticket #5739).
- Improved the performance of the *condor_negotiator* when running with a large number of users or groups. The accounting data is only written to disk when it changes, not unconditionally. (Ticket #5719).

Bugs Fixed:

- Fixed a bug in Docker universe that required the name of a transferred executable to begin with "./" (Ticket #5761).
- Fixed a bug the prevented Docker universe jobs from reporting their network usage correctly. (Ticket #5750).
- *condor_who* now reports docker universe jobs more completely. (Ticket #5740).
- Fixed bugs preventing HTCondor daemons from recognizing an address in Sinful format as its own when operating in mixed (IPv4 and IPv6) mode. One manifestation of this would be errors from the HAD daemon when specifying hosts by name in the `HAD_LIST` or `REPLICATION_LIST`. (Ticket #5728). (Ticket #5776).
- *condor_user_prio* now more correctly shows information about submitters flocking to a pool, but who haven't used any resources. (Ticket #5743).
- No longer leak a file in the user's home directory each time a job is submitted to Slurm. (Ticket #5742).
- Fix a bug that prevented HTCondor from removing jobs from Slurm. (Ticket #5804).
- Fixed a bug when attempting to authenticate using multiple methods wherein if a method failed, the remaining methods were not always attempted. (Ticket #5673).
- Fixed a bug that prevented the condor_schedd from reading the job's X.509 proxy file when writing information to the `SCHEDD_AUDIT_LOG`. (Ticket #5770).
- Fixed a bug in *condor_q* where the SIZE column would not grow as needed to fit the data. (Ticket #5667).
- Fixed a bug where the *condor_schedd* did not treat a user as a queue superuser when it should have if the configuration included a map file, which is common for GSI authentication. (Ticket #5530).
- Lengthen the watchdog timeout in the systemd service file to 1 minute. The previous value of 5 seconds has taken down HTCondor for a single slow DNS query. (Ticket #5819).

Version 8.5.5

Release Notes:

- HTCondor version 8.5.5 released on June 6, 2016.

New Features:

- The EC2 GAHP now rate-limits its requests, and responds to overload warnings with an exponential back-off. Additionally, fewer operations are now performed on a per-job basis (as few as one in some cases). The resulting scalability improvements have been demonstrated to permit a single GAHP to manage ten thousand instances. Because the overload condition is account- and region- specific, the grid manager now launches a GAHP for each account-region pair. We therefore recommend adding `D_PID` to `EC2_GAHP_DEBUG`, for disambiguation, and this is now the default. (Ticket #5561). (Ticket #5588). (Ticket #5620).
- The grid manager now assigns `HoldReasonCode`s and `HoldReasonSubCode`s to EC2 jobs when they go on hold. Values are subject to change until the stable release. (Ticket #5628).
- The grid manager now advertises some metrics from the EC2 GAHP. (Ticket #5580).
- Some Linux distributions for supercomputer compute nodes and others distributions for docker images have no /var/run/utmp. HTCondor no longer aborts when this file is missing, when it tries to determine keyboard idle times, it just assumes these kinds of machines have no keyboards. (Ticket #5624).
- Docker Universe jobs now correctly advertise `RemoteUserCpu` and `RemoteSysCpu` in their job ad and in the job log file. (Ticket #5609).
- A batch name specified for a DAG (with the *condor_submit_dag* **-batch-name** option) is now propagated to all jobs of that DAG, including sub-DAGs. (Ticket #5493).
- The batch name for a *condor_dagman* job (if not set) now defaults to *DagFile+cluster* (where *DagFile* is the primary DAG file of the *condor_dagman* job, and *cluster* is the HTCondor cluster of the *condor_dagman* job). Because the batch name is now propagated throughout a workflow, if no batch name is specified, the batch name for all jobs in the workflow will be *DagFile+cluster* of the top-level *condor_dagman* job. (Ticket #5605).
- The files named in the submit file attributes **vm_disk**, **xen_kernel**, and **xen_initrd** now refer to locations on the execute machine. *condor_submit* no longer modifies these values or checks for their existence on the submit machine. If these files need to be transferred by HTCondor, then they should be listed in **transfer_input_files** and their presence in these vm universe attributes shouldn't include any path information. (Ticket #4167).
- In the python bindings, an `ExprTree` can be cast to an integer or floating point value. (Ticket #5636).
- HTCondor now supports the following systemd features: Socket Activation, Watchdog, Status message, and journald logging. In these release, the Socket Activation is not configured, because the security system is not prepared to properly handle the socket passed in from outside HTCondor. (Ticket #4144).
- Added config knob `DEFAULT_MASTER_SHUTDOWN_SCRIPT` to specify a default program to exec as root upon *condor_master* exit. See Section 3.5.7 for details. (Ticket #5590).

- The python bindings now support a per-thread security context, allowing the modification of various parameters such as the pool password and the X509UserProxy location. (Ticket #5632).

Bugs Fixed:

- Fixed a bug that caused file transfers to fail when using Bosco. (Ticket #5704).

Version 8.5.4

Release Notes:

- HTCondor version 8.5.4 released on May 2, 2016.
- The **deltacloud** type in the grid universe, which allowed submission to Deltacloud services, has been removed. (Ticket #5569).

New Features:

- *condor_status* can now display the utilization of a *condor_startd* with a single line of output for each machine rather than a line per slot. In this release this output is enabled by passing **-compact** to *condor_status* but in a future release this will be the default output of *condor_status*. (Ticket #5596).
- Improved the performance of the *condor_collector* by not computing dropped update statistics, statistics which have never been accessible by users. (Ticket #5566).
- The performance of the *condor_history* tool has been significantly improved. (Ticket #5536).
- *condor_user_prio* now queries the *condor_collector* for accounting information by default, when appropriate. This should be much faster than the older way of querying the *condor_negotiator*. The old path is still available by passing the -negotiator option to the tool. (Ticket #5508)..
- The default value of `DAGMAN_ALWAYS_RUN_POST` has been changed from `True` to `False`. This means that, by default, if the PRE script of a DAG node fails, the POST script of the node will *not* be run. (This had been the default behavior until version 7.7.2. The 7.7.2-8.5.3 behavior can be restored by setting `DAGMAN_ALWAYS_RUN_POST` to `True`, or by passing the new **-AlwaysRunPost** argument to *condor_submit_dag*.) (Ticket #5477).
- The *batch_gahp* can now submit multi-core jobs to HTCondor. (Ticket #5638).
- The *batch_gahp*'s ability to generate a limited X.509 proxy for use by the job on the execute machine can now be disabled, which is now the default. (Ticket #5601).
- The *condor_schedd* will now send submitter ad updates for idle submitters less frequently than updates for submitters that have jobs in the queue. There are two new configuration variables to control this behavior. `ABSENT_SUBMITTER_LIFETIME` is the number of seconds after the last job for that submitter leaves

the queue that the submitter will continue to send updates to the *condor_collector*. It defaults to 1 week. ABSENT_SUBMITTER_UPDATE_RATE is the maximum rate in seconds at which the *condor_schedd* will send updates to the *condor_collector* for a submitter that has no jobs in the queue. It defaults to 5 minutes. (Ticket #5559).

Bugs Fixed:

- Fixed a bug that caused the *condor_schedd* to exit when receiving an updated X.509 proxy for a job. (Ticket #5645).
- In expressions in the Job Router's configuration, attributes no longer require a 'TARGET.' scope prefix. (Ticket #5550).
- Fixed a bug in *condor_q* **-xml** that would put the XML header after the body unless **-stream** was passed. (Ticket #5597).

Version 8.5.3

Release Notes:

- HTCondor version 8.5.3 released on March 24, 2016.

New Features:

- ENABLE_IPV4 and ENABLE_IPV6 both now accept the special value "AUTO", which is true if an interface with the corresponding protocol exists on the host, and false otherwise. (Ticket #5524).
- ENABLE_IPV4 and ENABLE_IPV6 both now default to the special value "AUTO". Additionally, the new configuration macro PREFER_IPV4 is true by default. This macro causes HTCondor to prefer IPv4 over IPv6 when choosing an address to advertise, when choosing the address of daemon looked up in the collector, and when resolving DNS queries. (Ticket #5104).
- New configuration macros added: IPV4_ADDRESS, IPV6_ADDRESS, IP_ADDRESS_IS_V6. (Ticket #5512).
- New attributes have been added to the Submitter ClassAd to indicate the number of Idle and Running jobs for Scheduler universe and for Local universe. (Ticket #5519).
- Jobs can now be submitted to the Slurm batch scheduling system via the new **slurm** type in the grid universe. (Ticket #5515).
- In addition to logging to the file KERNEL_TUNING_LOG, the default LINUX_KERNEL_TUNING_SCRIPT now also logs to syslog and /etc/systcl.d/99-htcondor.conf. (Ticket #5489).
- *condor_history* **-autoformat** now supports the j option to print job ids like *condor_q* does. (Ticket #5558).

- HTCondor is now built and linked with Globus 6.0. (Ticket #5520).
- Pre-size the ClassAd hash table to improve the performance of the *condor_collector* when getting ClassAd updates. (Ticket #5551).
- The negotiator now forwards accounting information to the collector, where it can be easily queried and monitored. (Ticket #5491).

Bugs Fixed:

- Fixed a bug on *condor_history* that could result in truncation of the job id field. (Ticket #5527).

Version 8.5.2

Release Notes:

- HTCondor version 8.5.2 released on February 18, 2015.

New Features:

- ***condor_q* now defaults to showing only the current user's jobs, unless the current user is a queue superuser.** This behavior can be turned off by setting the new configuration option `CONDOR_Q_ONLY_MY_JOBS` to `False` in the configuration for either the *condor_schedd* or *condor_q*. It can also be turned off by passing the new **-allusers** command line option to *condor_q*. (Ticket #5271).
- Added support for immutable job attributes and protected job attributes. (Ticket #5065).
- Improved the speed of *condor_q* and other tools when querying non-local servers. (Ticket #5150).
- The new **-batch** command line option to *condor_q* can be used to show a single line of progress information for a batch of jobs, where a batch is either an entire workflow (a DAG, including sub-DAGs), all of the jobs in a cluster, all of the jobs from a single user that have the same executable specified in their submit file, or all of the jobs from a single user that have the same batch name. (Ticket #4976).
- *condor_submit* and *condor_submit_dag* both have a new command line option **-batch-name** which can be used to set the batch name used by *condor_q* **-batch**. (Ticket #5490).
- *condor_q* and *condor_status* now attempt to avoid truncating data by making use of the full width of the terminal. (Ticket #5429). (Ticket #5459).
- Docker Universe jobs now report and update ResidentSetSize, ImageSize, MemoryUsage, NetworkInput, and NetworkOutput attributes in the job ad and log file. (Ticket #5456).
- Added the capability to set ClassAd attributes for a *condor_dagman* job within the DAG file by using the new `SET_JOB_ATTR` command. (See section 2.10.9 for details.) (Ticket #5107).

- The `dagman.out` file produced by *condor_dagman* now has event timestamps added to the lines that report *condor_dagman* reading a log event. For example:

```
01/13/16 11:29:03 Event: ULOG_SUBMIT for HTCondor Node NodeA (674.0.0) {01/13/16
```

  The timestamp in curly brackets at the end is the actual timestamp of the event. (Ticket #5439).

- The **-run** and **-hold** arguments of *condor_q* used to produce garbage output when used with other formatting options such as **-format**. Now they will always constrain the query, and will also set the output format when used by themselves. (Ticket #11).

- The *condor_status* **-verbose** argument has been removed; the equivalent **-long** argument should be used instead.

Bugs Fixed:

- On Windows, configuring HTCondor to restrict the range of outbound port numbers may cause substantial delays when using the command-line tools. Since we now know that it's not free to do so, `LOWPORT` and `HIGHPORT` no longer restrict the port numbers of outbound connections on Windows. If you still require this functionality, use `OUT_LOWPORT` and `OUT_HIGHPORT`. (Ticket #4711).

- Fixed a bug that could cause a daemon to be in the wrong privilege state when attempting to act as the user. (Ticket #5467).

Version 8.5.1

Release Notes:

- HTCondor version 8.5.1 released on December 21, 2015.

- Shared port is enabled by default. (Ticket #3813). (Ticket #5103).

New Features:

- The *condor_startd* history file now contains the peak memory usage, by an exited job, not the more recent. (Ticket #5436).

- When the *condor_starter* evicts a job, perhaps because it has exceeded a memory limit, it does not transfer back to the submit machine the sandbox of working files. This is consistent with other types of holds. (Ticket #5437).

- The *condor_startd* now advertises the following attributes on Linux machines: CpuFamily CpuModelNumber CacheSize. These are pulled from the /proc/cpuinfo file. (Ticket #5323).

- *condor_q* has a new option **-schedd-constraint** which can be used to constrain the queues displayed when using the **-global** option. (Ticket #5043).

- When an HTCondor-C job is submitted to a remote *condor_schedd*, the remote job ad now includes the attribute `SubmitterGlobalJobId`, whose value is the same as the attribute `GlobalJobId` in the original HTCondor-C job. (Ticket #3472).
- The *condor_schedd* now sets environment variables for scheduler universe jobs so that the jobs can more easily find the *condor_schedd*'s contact information. On machines where there are multiple *condor_schedd*s running, this helps DAGMan and similar applications contact the *condor_schedd* that started them. (Ticket #5166).
- When `SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION` is set to `True`, the related authorizations are now automatically enabled. Previously, `submit-side@matchsession` and `execute-side@matchsession` entries had to be added to the `ALLOW_DAEMON` and `ALLOW_CLIENT` (if set) authorization parameters in order for this feature to work. (Ticket #5304).

Bugs Fixed:

- *condor_history* run on a pool with partitionable slots now shows the correct dynamic slot. (Ticket #4261).

Version 8.5.0

Release Notes:

- HTCondor version 8.5.0 released on October 12, 2015.

New Features:

- The *condor_startd* history file contains two new attributes: BadputCausedByDraining and BadputCausedByPreemption, two boolean-valued attributes which are true if the job was evicted not by a user request. (Ticket #5255).
- The python bindings have a new Claim API, allowing Computing-On-Demand (COD) to be invoked via python. (Ticket #5130).
- The python bindings can now submit multiple distinct processes using the `submitMany` method, similar to a *condor_submit* file with multiple `queue` statements. (Ticket #4916).
- The python bindings now provide improved support for managing multiple concurrent queries. (Ticket #5187).
- As an experimental feature, the python bindings implement the HTCondor negotiation protocol. (Ticket #5125).
- Changed `"Condor"` to `"HTCondor"` in *condor_dagman* output (mainly in the `dagman`.out file). (Ticket #5144).
- The new configuration parameter `JOB_SPOOL_PERMISSIONS` controls the permissions on a job's spool directory managed by the *condor_schedd* on unix. It defaults to the value `user`, which results in a permissions value of `0700`. Other valid values are `group` (permissions `0750`) and `world` (permissions `0755`). Previously, all job spool directories had access permissions of `0755`. (Ticket #4896).

- The *condor_schedd* no longer changes the ownership of spooled job files that it manages. Now, the files are always owned by the submitting user. The previous behavior of changing ownership to/from the `condor` account can be restored by setting the new configuration parameter `CHOWN_JOB_SPOOL_FILES` to `True`. (Ticket #5226).

Bugs Fixed:

- None.

10.5 Stable Release Series 8.4

This is a stable release series of HTCondor. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 8.4.x releases. New features will be added in the 8.5.x development series.

The details of each version are described below.

Version 8.4.12

Release Notes:

- HTCondor version 8.4.12 released on July 13, 2017.

New Features:

- Added a new knob, `STARTD_RECOMPUTE_DISK_FREE`, which defaults to `true`, which tells the *condor_startd* to periodically recompute and advertise free disk space. Administrators can set this to `false` for partitionable slots whose execute directory is used by HTCondor alone. (Ticket #6301).

Bugs Fixed:

- None.

Version 8.4.11

Release Notes:

- HTCondor version 8.4.11 released on January 23, 2017.

New Features:

- Added a new config knob, `IGNORE_LEAF_OOM`, which defaults to `False`. When `True`, it tells HTCondor *not* to kill and hold a job that is within its memory allocation, even if other processes within the same cgroup have exceeded theirs. (Ticket #5775).

Bugs Fixed:

- Fixed a bug where the effects of a startd cron job would not be used by the *condor_startd* when making policy decisions (e.g., evaluating the `START` expression) until an `UPDATE_INTERVAL` had passed. This was generally only noticeable if you set `STARTD_CRON_AUTOPUBLISH` to a value other than `NEVER`, which could cause the startd to reject claims from the negotiator that had been made based on the cron-updated value(s). (Ticket #6057).
- Fixed a bug in pslot preemption where it could cause matching jobs to not start for a long time. (Ticket #6055).
- Fixed a bug that caused a job to not be cleaned up when the job lease expires, if glexec is in use. (Ticket #6058).
- Fixed a problem, found while testing on Ubuntu 16, where the negotiator would crash at the end of the negotiation cycle. (Ticket #6064).
- Updated the default configuration in the Debian and Ubuntu packages to look for the Ganglia shared libraries in the proper place. (Ticket #5939).
- Updated the Enterprise Linux 7 RPM to require the proper SELinux utilities for its post-install script. (Ticket #6081).

Version 8.4.10

Release Notes:

- HTCondor version 8.4.10 released on December 13, 2016.

New Features:

- None.

Bugs Fixed:

- Added additional SELinux type enforcement rules on Enterprise Linux 7 for the *condor_shared_port* daemon and Linux tuning script. The RPM post install script makes the HTCondor SELinux domains permissive. (Ticket #5449). (Ticket #5560). (Ticket #5835).

- Fixed a performance problem in the *condor_schedd* when `RequestCpus` was an expression. Added a new parameter `SCHEDD_SLOT_WEIGHT`, which may be needed if `SLOT_WEIGHT` is not the default value of "Cpus", and refers to expressions in the job ad. (Ticket #5996).
- When transferring a job's sandbox, the permissions on sub-directories are now preserved in the same that they are for regular files. Previously, the permissions were modified by HTCondor daemon's umask, and directories transferred from a Windows machine to a UNIX machine had no permissions enabled. (Ticket #5948).
- Fixed bug in the `HOLD_IF_CPUS_EXCEEDED` configuration template metaknob. (Ticket #5933).
- Fixed a bug in the `LIMIT_JOB_RUNTIMES` configuration template metaknob so that it works in the face of a non-default MaxJobRuntime. (Ticket #5961).
- Fixed a bug that made it so a restart of the *condor_schedd* was required in order to change `REMOVE_SIGNIFICANT_ATTRIBUTES` to remove an attribute that the *condor_schedd*s had already marked as significant. (Ticket #5983).
- When creating a Google Compute Engine instance, instruct the server to delete the auto-created disk image when the instance is removed. (Ticket #5999).
- Fixed a bug in our support for Google Compute Engine which would cause authorization tokens to be renewed five minutes after the deadline instead of five minutes before. Naturally, this led to ten minutes of interrupted service for jobs (or workloads) which lasted longer than the initial valid duration of the tokens. (Ticket #6009).
- Fixed a bug that would cause the *condor_schedd* to crash when a condor cron job was scheduled to start during the one hour gap in daylight savings time when the clocks are moved backwards one hour. (Ticket #5995).
- Fixed a bug that would cause a core dump when running the condor_history command against a remote schedd. The results would be returned correctly, but a core file would appear in the log directory after the command exited. (Ticket #5956).
- In the Python bindings, fixed bugs in the LogReader and EventIterator classes that could cause invocations of poll() to return prematurely with no event. (Ticket #5920).
- Fixed a bug in *condor_dagman* that caused a file named '' (two single quotes) to be created if the `DAGMAN_SUPPRESS_JOB_LOGS` configuration macro was set to `True`. (Ticket #5941).
- Fixed a bug that can cause the *condor_starter* to crash if the connection to the *condor_shadow* is lost during file transfer. (Ticket #5972).
- Fixed a bug that allowed a user to bypass the `MAX_JOBS_PER_OWNER` limit by specifying an accounting group or nice_user in their submit file. (Ticket #5946).
- When there are no GPUs on a machine, *condor_gpu_discovery* would write to stderr in addition to its normal output, this made it hard to use the **-config** option as intended. *condor_gpu_discovery* has been changed so that it will never write to stderr when the **-config** option is specified, instead it will write error messages as configuration comments to stdout. (Ticket #5989).
- Removed obsolete ControlGroup option from HTCondor's systemd service unit configuration file. (Ticket #5997).
- Compiled benchmarking programs as a Position Independent Executable. Position Independent Executables are a requirement for entry into Debian 9. (Ticket #5994).

- Fixed a denial of service vulnerability when using the *condor_credd* on the Windows platform. (Ticket #5984).
- Fixed a bug where the **-pool** argument would be ignored by *condor_ssh_to_job* under certain circumstances. (Ticket #5919).

Version 8.4.9

Release Notes:

- HTCondor version 8.4.9 released on September 29, 2016.

New Features:

- Increased the maximum number of unique attributes that can be set by the *condor_chirp* command **set_job_attr_delayed** from 50 to 100, and added the configuration knob `CHIRP_DELAYED_UPDATE_MAX_ATTRS`. See section 3.5.11 for more information. (Ticket #5891).

Bugs Fixed:

- Fixed a bug where if the *condor_startd* crashed while running a Docker universe job, the job would be left running and not removed when the *condor_startd* restarted. The *condor_startd* now removes any orphaned Docker universe jobs on restart. (Ticket #5858).
- Fixed a bug that printed spurious locking-related warnings to the StarterLog when running Docker universe jobs. (Ticket #5876).
- The Job Router and HTCondor-C now properly send a RESCHEDULE command to the *condor_schedd* after submitting a job. (Ticket #5903).
- Fixed bugs in the Job Router that could cause a routed job to be aborted if the `UPDATE_JOB_INFO` hook printed attributes to be set in the job ad. (Ticket #5899).
- The Job Router now uses the correct name for the configuration parameter for the JOB_FINALIZE hook. Previously, the Job Router used the name JOB_EXIT, counter to what was documented. (Ticket #5802).
- Updated systemd configuration to start HTCondor after NIS has started. (Ticket #5814).
- Updated systemd configuration to start HTCondor after local LDAP name service daemon has started. (Ticket #5836).
- Updated systemd configuration to attempt restart of HTCondor daemons after 1 minute. (Ticket #5836).
- In the RPM packages, move the systemd tmpfiles configuration file to the recommended directory (/usr/lib/tmpfiles.d). (Ticket #5896).
- Fixed a bug introduced in 8.4.5 that caused configuration variables starting with STARTD. or STARTER. to be ignored. (Ticket #5861).

- Fixed a typo in the desired value of 'rmem_max' in the Linux kernel tuning script. Improved logging of Linux kernel tuning script by including the name of the file (not) being changed. (Ticket #5829).
- Fixed a bug that could cause the *condor_master* to crash after restarting the *condor_shared_port* daemon. (Ticket #5801).
- Fixed a bug that could cause the wrong dynamic slots to be preempted for a match when `ALLOW_PSLOT_PREEMPTION` is set to `True`. (Ticket #5748).
- Fixed a bug in *cream_gahp* that caused it to delegate RFC-format X.509 proxies incorrectly to the CREAM service. (Ticket #5773).
- Fixed a bug where the Windows version information was set to a single value for multiple programs. This resulted in crash boxes for most of the HTCondor tools being reported as a crash of *condor_gpu_discovery* (Ticket #5795).
- Fixed a bug whereby the *condor_collector* process would exit with an error several times per hour if the configuration knob `NO_DNS` is set to True. (Ticket #5762).
- Fixed monitoring of memory and CPU usage of running jobs on Mac OS X. This monitoring didn't work for a personal installation of HTCondor. With Mac OS X 10.11 and above, this monitoring resulted in a flood of errors messages to the system logs for a root-based installation. (Ticket #5777).
- Fixed a bug when attempting to authenticate using multiple methods wherein if a method failed, the remaining methods were not always attempted. (Ticket #5674).
- Fixed a bug where *condor_userprio* may fail to display the correct priority factor value for a user associated with a group. (Ticket #5848).
- Fixed a bug that can cause the *condor_procd* to crash. Fixed a bug that prevented other daemons from talking to the *condor_procd* when it is restarted after a crash. (Ticket #5863).
- If the *condor_procd* crashes, the *condor_master* now tries to restart it several times. Previously only one restart attempt was done. (Ticket #3655).
- Fixed a bug that resulted in the *condor_starter* crashing when attempting to run a BOINC backfill job. (Ticket #5862).
- Fixed a bug in the configuration language where an if defined test would reject a valid variable name when it had both an underscore and a digit. (Ticket #5914).
- Fixed a bug that caused the condor_ssh_to_job command to fail when using the HTCondor RPM installation. (Ticket #5591).

Version 8.4.8

Release Notes:

- HTCondor version 8.4.8 released on July 5, 2016.

New Features:

- None.

Bugs Fixed:

- Fixed a memory leak in the condor_q client code that impacted users of the Python API call htcondor.Schedd().query(). (Ticket #5727).
- Fixed a bug that caused file transfers to fail when using Bosco. (Ticket #5710).
- Fixed a bug that could cause the *condor_schedd* to crash when using `SCHEDD_CRON_JOBLIST`. (Ticket #5715).
- The *condor_schedd* now rejects job submissions when the job owner doesn't have a user account on the machine. Previously, the *condor_schedd* would accept such jobs and then fail to run them. (Ticket #5734).
- Fixed a bug introduced in the 8.4.7 release that resulted in the remote *condor_history* command failing unless the **-limit** argument is used. (Ticket #5735).
- Fixed a bug in *condor_history* that caused it to treat all unrecognized arguments as user names (Ticket #5706).
- The high-availability daemon now properly detects changes to the `HAD_LIST` when reconfigured. (Ticket #5753).
- The high-availability daemon now properly internalizes the `HAD_LIST` when reconfigured. (Ticket #5754).
- Fixed a bug that caused the *condor_master* to stop responding after it restarted a child daemon when shared port is enabled on Windows. This bug could also result in a hang on shutdown. (Ticket #5713).
- Fixed a bug that could cause *condor_status* or *condor_q* to crash when the **-xml** option is used. (Ticket #5718).
- Fixed a bug introduced in the 8.4.7 release that resulted in a parse error from *condor_submit* when `JobAdInformationAttrs` was set in the configuration variable `SUBMIT_ATTRS`. (Ticket #5720).

Version 8.4.7

Release Notes:

- HTCondor version 8.4.7 released on June 6, 2016.

New Features:

- Docker universe jobs now drop all Linux capabilities by default. The new knob DOCKER_DROP_ALL_CAPABILITIES, which defaults to true can be set to false to revert to the old behavior. (Ticket #5679).

- Added configuration variable MAX_TIME_SKIP to control how much system clock skip is allowed before the HTCondor daemons restart. See Section 3.5.3 for more information.
- On Linux, HTCondor appropriately tunes kernel parameters root_maxkeys and root_maxbytes to prevent *condor_master* startup failures on older Linux kernels. (Ticket #5671).
- The configuration variable SUBMIT_ATTRS now understands the +Attr syntax that *condor_submit* uses to inject attributes directly into the job ClassAd. (Ticket #5694).
- The *condor_submit* variable job_lease_duration can now be an expression. (Ticket #5694).

Bugs Fixed:

- All $function macro substitutions in in configuration files will now correctly handle variables with subsystem and localname prefixes as well as self references. In particular VAR = $F(VAR) now substitutes correctly rather than hanging forever. (Ticket #5565).
- Fixed a bug in Docker universe where the job would not run with the correct group id. (Ticket #5649).
- Fixed a performance problem in the *condor_schedd* that could cause it to become unresponsive for several minutes after the set of significant attributes for negotiation changes. (Ticket #5648).
- Fixed a bug where the python bindings ClassAd parser would fail to detect whether old or new format ClassAds were present in a stream, even though the ClassAd format was specified in advance. (Ticket #5643).
- Fixed a bug where some floating point values would have an extra .0 appended to the end when printed (e.g. 2E40.0). These values could not be read properly by normal number parsing functions. (Ticket #5682).
- When using GRIDMANAGER_SELECTION_EXPR, grid ads from different *condor_gridmanager* instances will no longer overwrite each other in the *condor_collector*. (Ticket #5683).
- In addition to logging to the file KERNEL_TUNING_LOG, the default LINUX_KERNEL_TUNING_SCRIPT now also logs to syslog and /etc/systcl.d/99-htcondor.conf. (Ticket #5489).
- Fixed a bug on *condor_history* that could result in truncation of the job id field. (Ticket #5527).
- On Windows, configuring HTCondor to restrict the range of outbound port numbers may cause substantial delays when using the command-line tools. Since we now know that it's not free to do so, LOWPORT and HIGHPORT no longer restrict the port numbers of outbound connections on Windows. If you still require this functionality, use OUT_LOWPORT and OUT_HIGHPORT. (Ticket #4711).
- Fixed a bug that would cause *condor_submit* to create extra, incorrectly named output and error files when $$ substitution is used as part of the filenames. (Ticket #2720).
- Fixed a bug that would cause the *condor_history_helper* to be invoked using the wrong name on Windows (Ticket #5656).
- Fixed a bug that would sometimes cause configuration variables with a subsystem prefix to be ignored. (Ticket #5310).
- Fixed a bug that could cause HAD to fail if a machine has an IPv6 address. (Ticket #5659).

- Fixed a bugs in *condor_history* when fetching history from a remote *condor_schedd*. The bugs caused complete failure when the remote *condor_schedd* was running Windows, and would corrupt some string values when the remote *condor_schedd* was any other operating system. (Ticket #5701).

Version 8.4.6

Release Notes:

- HTCondor version 8.4.6 released on April 21, 2016.

New Features:

- *condor_advertise* **-multiple** now tolerates multiple blank lines in the input file. It no longer quits parsing on the first first blank line that does not follow a valid ClassAd. (Ticket #5147).

Bugs Fixed:

- Fixed bug where when partitionable slots were enabled in the *condor_startd*, a job would be unable to start running on that machine in some cases. (Ticket #5626).
- Fixed a bug that would cause the *condor_startd* to crash when `ALLOW_PSLOT_PREEMPTION` was enabled. (Ticket #5586).
- Fixed a bug introduced in version 8.3 that removed the attribute `REMOTE_GROUP_RESOURCES_IN_USE` from the job ad in the negotiator. (Ticket #5593).
- Fixed a bug where HTCondor would regard as invalid text representations of IPv6 addresses which were the longest possible. This bug typically manifested as a failure to contact hosts which were advertising IPv6 addresses of this sort. (Ticket #5585).
- Fixed a memory leak in the *condor_negotiator* when `ALLOW_PSLOT_PREEMPTION` was enabled. (Ticket #5571).
- Fixed a bug where after a *condor_schedd* restart the submitter attribute `WEIGHTED_JOBS_RUNNING` would be incorrectly computed. (Ticket #5637).
- Fixed a bug when using `CLAIM_PARTITIONABLE_LEFTOVERS` and flocking. Machines from a remote pool could be treated as if they were in the local pool. As a result, the `RemotePool` attribute would not be set in the ads of jobs running on these machines, and the `FlockedJobs` and `RunningJobs` attributes of submitter ads would have incorrect values. (Ticket #5577).
- Fixed a bug that could cause a job's supplemental groups to be set incorrectly when `SOFT_UID_DOMAIN` is set to `True`. (Ticket #5603).
- Fixed a bug that caused supplemental groups to be set incorrectly when executing file transfer plugins and various hooks. (Ticket #5600).

- Fixed a bug that resulted in Windows 10 being reported as WindowsUnknown in the `OPSYSNAME` attribute of the *condor_startd* ClassAd. (Ticket #5575).
- Fixed a typo in the `LIMIT_JOB_RUNTIMES` policy configuration template that prevented the policy from working as intended. (Ticket #5307).

Version 8.4.5

Release Notes:

- HTCondor version 8.4.5 released on March 22, 2016.

New Features:

- The default for `DAGMAN_LOG_ON_NFS_IS_ERROR` has been changed from `True` to `False`. This is the result of changes in the 8.3 series that mean that file locking is no longer required on user logs. (Ticket #5516).

Bugs Fixed:

- Fixed a bug where HTCondor would unconditionally retry non-successful DNS lookups of the local system's hostname; this could cause delays of up to sixty seconds when using command-line tools on systems whose hostname was not in DNS. We no longer retry on errors at all, and only retry failures which are temporary. (Ticket #5553).
- Fixed a bug that would cause *condor_schedd*s flocking to remote pools to not send no jobs, or fewer jobs than possible to the remote pool. This was a result of not correctly setting the submitter attribute `WeightedJobsRunning` for flocked pools. (Ticket #5539).
- Accounting group names that contain spaces are now rejected by *condor_submit* and ignored by the *condor_negotiator*. Previously, submitting a job with an accounting group name that contained a space would cause the *condor_negotiator* to fail at startup. (Ticket #5538).
- Fixed a bug whereby per-job history files (enabled by the configuration setting `PER_JOB_HISTORY_DIR`) may briefly appear to be empty or incomplete. (Ticket #5562).
- Fixed a bug whereby ClassAds written into history files may contain the same attribute multiple times. (Ticket #5548).
- Fixed a bug that caused DAGMan to not work correctly with some local universe node jobs. (This bug was introduced in version 8.3.0.) (Ticket #5299).
- Fixed a bug that resulted in jobs managed by the *condor_job_router* not reporting memory and disk usage of the job correctly. (Ticket #5552).

- Reworked a bug fix from the 8.4.3 release that was designed to allow for more than 100 dynamic slots to be a bit more generous in allocating Disk to those slots. Now, those slots are less prone to fail to match subsequent jobs. (Ticket #5535).
- Fixed a bug in the randomization of ports within the LOWPORT to HIGHPORT range that would sometimes generate ports outside of this range on Windows. (Ticket #5555).
- Fixed a bug in *condor_off* **-peaceful** that could result in never sending the "off" command to machines when at least one of the machines could not be contacted when sending the previous "peaceful" command. (Ticket #5504).
- When cgroups are in use, limit the amount of file system cache in the kernel to prevent the OOM killer from killing jobs that use a large amount of file system cache. (Ticket #5500).

Version 8.4.4

Release Notes:

- HTCondor version 8.4.4 released on February 4, 2016.

New Features:

- None.

Bugs Fixed:

- Fixed a bug that caused the *condor_collector* to crash if `CONDOR_DEVELOPERS_COLLECTOR` failed to resolve. (Ticket #5492).
- Fixed a bug that caused Condor-C jobs to fail when `JobLeaseDuration` was set to less than one hour (3600 seconds). The remote job would be aborted due to the job lease not being renewed. (Ticket #5446).
- Fixed a bug that could cause HTCondor to misreport an EC2 job as running when it had in fact been purged from the service. Fixed bugs that could cause a running EC2 job to be misreported as idle. HTCondor also no longer sends EC2 services superfluous queries. (This may increase the latency of HTCondor job status updates.) (Ticket #4568).
- The grid manager now aborts if the GAHP hangs, which we detect by the absence of a response after `GRIDMANAGER_GAHP_RESPONSE_TIMEOUT` seconds. The EC2 GAHP now performs many fewer memory allocations in the course of normal operations, which improves stability on some systems. (Ticket #5442).
- Fixed a bug that caused the *condor_master* to fail if a shared port daemon address file written by a version of HTCondor prior to 8.4.0 is present. (Ticket #5488).
- Fixed a bug that caused updates to the job attribute `TimerRemove` to not be respected while the job was being managed by the *condor_shadow*, *condor_gridmanager*, or the Job Router. (Ticket #5470).

- Fixed a bug where the job policy expression of a job could appear in one of the `Reason` attributes of another job. (Ticket #5466).
- Fixed a bug, that occurred on the Windows platform, that would cause the *condor_shadow* to hang while trying to delete old shadow logs when the value of `MAX_NUM_SHADOW_LOG` was larger than the default value of 1. This bug would also sometimes result in the *condor_schedd* hanging. (Ticket #5499).

Version 8.4.3

Release Notes:

- HTCondor version 8.4.3 released on December 16, 2015.

New Features:

- None.

Bugs Fixed:

- Fixed a bug that caused the **-append** option to be handled too late to apply to the first Queue statement in a *condor_submit* file. (Ticket #5414).
- Fixed a bug that prevented running more than 100 slots on a single *condor_startd* with partitionable slots. (Ticket #5398).
- Fixed a bug which caused **ec2_iam_profile_name** not to work for Spot instances. (Ticket #5410).
- Fixed a bug where the cgroup VM limit would not be set for sizes over 2 Gibibytes. (Ticket #5434).
- Fixed bugs that prevented the HTCondor daemons from working promptly at startup when the *condor_shared_port* daemon was in use on Windows platforms. (Ticket #5283). (Ticket #5430). (Ticket #5431). (Ticket #5432). (Ticket #5433).
- Added SELinux type enforcement rules to allow the *condor_schedd* to use *sendmail* on Enterprise Linux 7 platforms. (Ticket #5418).
- Fixed a bug where HTCondor service would not start if the `condor_master.pid` file was empty on Linux platforms. (Ticket #5427).

Version 8.4.2

Release Notes:

- HTCondor version 8.4.2 released on November 17, 2015.

New Features:

- *condor_history* no longer reports an error when run on a system that does not have a history file. This change was made because the history file is not created until after the first job runs. So, users were always seeing an error message on a fresh installation of HTCondor. (Ticket #5374).

Bugs Fixed:

- Fixed a bug introduced in 8.4.1 that could cause the *condor_schedd* to exit. This affected remote submit, HTCondor-CE, and HTCondor-C. (Ticket #4522).
- The `TCP_FORWARDING_HOST` is now honored by HTCondor client programs. (Ticket #5339).
- Fixed a problem where Standard Universe jobs could not restart from a checkpoint in the Enterprise Linux 6 RPM distribution. (Ticket #5382). (Ticket #5383).
- Fixed bugs in the function of the DAGMan `DAGMAN_MAX_JOBS_IDLE`/`-maxidle` throttle, especially for node jobs that create multiple procs. (Ticket #5333).
- Fixed a problem where the RPMs would claim to publicly provide Globus shared libraries that are in a private location. (Ticket #5349).
- Added a default `request_memory` for *condor_submit* -interactive of 512 megabytes. Formerly, the default was one, which is insufficient in environments that strictly enforce memory usage. (Ticket #5344).
- Fixed a problem were the `condor_classad` RPM would claim to provide a replacement for the `classad` RPM in EPEL. (Ticket #5400).
- HTCondor now applies the configuration settings `GRIDMANAGER_GAHP_CALL_TIMEOUT` and `GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT` when running grid universe jobs for EC2 or Google Compute Engine. (Ticket #5300).
- Fixed a crash in the *condor_schedd* that happened when the schedd was under load and being shutdown in the fast mode. (Ticket #5371).
- Added a timeout to the *condor_fetchlog* command so that it will not hang forever waiting for a unresponsive daemon. (Ticket #5325).
- Fixed a problem that prevented HTCondor from building on some 64-bit Linux platforms such as Arm64. This was reported by Debian maintainers as their Bug 804386. (Ticket #5380).
- Fixed a problem where the platform string was incorrect in the RPM packages. (Ticket #5384).

Known Issues:

- The DAGMan workflow log file is not correctly written for local universe DAG node jobs that have no log file specified in the submit file, which causes DAGMan to wait forever, thinking the jobs have not completed. Note that this problem can be worked around by specifying *any* log file for the job, even `log = /dev/null`. (This bug is a regression that was introduced some time since version 8.2.4.) (Ticket #5299).

- DAG node retries do not work correctly with DAG node submit files that create more than one proc in the resulting cluster (such nodes cause DAGMan to hang if the retry is activated). We believe that this bug has existed since DAGMan first supported multi-proc node jobs. (Ticket #5350).

Version 8.4.1

Release Notes:

- HTCondor version 8.4.1 released on October 27, 2015.

Known Issues:

- Remote submit to an 8.4.1 *condor_schedd* is broken if file transfer is used. This also means HTCondor-CE and HTCondor-C are broken. This bug will be fixed in version 8.4.2. (Ticket #4522).
- `TCP_FORWARDING_HOST` is disregarded by HTCondor clients starting in version 8.3.6. This bug will be fixed in version 8.4.2 and 8.5.1. (Ticket #5339).

New Features:

- Added support to allow an admin to always volume mount certain directories into docker universe containers running on a host. (Ticket #5308).
- Added four policy metaknobs to simplify configuring a policy to either preempt or hold jobs that use more memory or CPU cores than provisioned in the slot. See the `POLICY` category of metaknobs in section 3.4.2 for additional information. (Ticket #5250).
- Added configuration variables and documentation so that we uniformly prefer `<var>_ATTRS` over `<var>_EXPRS` but support both. This includes `STARTD_ATTRS`, `STARTD_JOB_ATTRS` and `SUBMIT_ATTRS` which are often used by HTCondor sites which customize the configuration. These configuration variables are now exclusively for use by HTCondor administrators; The former default values for these variables have been moved into other configuration which is reserved for use by HTCondor developers. This is done to prevent administrators from accidentally removing the necessary defaults. A warning about use of `STARTD_EXPRS` has been disabled unless `STARTD_ATTRS` or `SLOT_TYPE_<n>_STARTD_ATTRS` is also used, since the use all three of these at the same time is not supported. (Ticket #5326).
- When *condor_reconfig* and *condor_restart* are run as root they will check to see if the condor user has read access to all of the configuration files before sending the command. This is done to prevent aborting the daemons accidentally by sending reconfig after the admin creates a new config file and forgets to give the condor user read access to that file. (Ticket #4506).
- Added the **-natural** sort option to *condor_status* to sort the slots in numerical order rather than alphabetical order. (Ticket #5131).

Bugs Fixed:

- When cgroups are enabled, and CGROUP_MEMORY_POLICY is soft, HTCondor now also sets the hard limit to the virtual memory limit of the job, if there is one. (Ticket #5280).
- If cgroups are enabled, and a job goes over the memory limit, the cgroup OOM killer fires, and the job is put on hold. HTCondor now updates the job's memory usage statistics with the most up to date usage, instead of relying on the previous snapshot. (Ticket #5341).
- Fixed a bug where the *condor_kbdd* could not accurately measure the keyboard idle time. This daemon now works correctly on Linux systems whose X server support the MIT screen saver extension. (Ticket #5265).
- Fixed a bug which prevented SOAP submissions. (Ticket #5260).
- The parameter `STARTD_HISTORY` is now set to record the job histories per startd, in the log directory of the execute machine. These can be read with the *condor_history* command. Previously the default was not to record these. (Ticket #5257).
- The parameter `SCHEDD_USE_SLOT_WEIGHT` now defaults to true, so that SLOT_WEIGHT can be used with hierarchical group quotas and partitionable slots. (Ticket #5256).
- Fixed bug whereby occasionally the command-line tools would emit debug messages to stderr with text "I am: hostname: . . . ". (Ticket #5276).
- Fixed a bug that prevented node retries from working on DAG nodes that are DAG-level NOOP nodes. (This bug has existed at least since the 8.2 series.) (Ticket #5277).
- Fixed a problem when the HTCondor executables were not compiled with `RPATH` enabled on Enterprise Linux 6 platforms. `RPATH` is used to load Globus and other libraries from the condor-externals RPM. (Ticket #5294).
- The job attribute `JobCurrentStartTransferOutputDate` is now properly reported in the job ad. (Ticket #5298).
- Fixed configuration parameter `NETWORK_HOSTNAME`, which was broken starting with version 8.3.2. (Ticket #5288).
- Fixed a bug that could cause the Job Router to crash when invoking a transformation hook. (Ticket #5224).
- Fixed several memory leaks in the *nordugrid_gahp*. (Ticket #5322).
- Improved the *batch_gahp* to better handle batch systems that reuse job IDs. (Ticket #5062).
- When the *batch_gahp* rejects a request because it is overloaded, the *condor_gridmanager* now reduces the rate of requests and retries the rejected request later. (Ticket #5253).

  item The *condor_had* and *condor_replication* daemons now work properly when Shared Port is enabled. They still require their own dedicated ports. (Ticket #5301).
- Fixed a bug that cause *condor_mips* to report numbers about 40 percent lower than it should on Linux platforms. (Ticket #5261).
- Fixed a bug in *condor_install* that would cause it to configure HTCondor to advertise the public IP addresses to the collector even when using localhost or 127.0.0.1 for a personal HTCondor. (Ticket #5286).

- Fixed a bug in *condor_q* that caused slices in the Queue statement to be treated as part of the arguments filename when the slice was longer than 8 characters. (Ticket #5273).
- Added SELinux type enforcement rules to allow the *condor_schedd* to be able to access user files in NFS mounted file systems. (Ticket #5343).

Version 8.4.0

Release Notes:

- HTCondor version 8.4.0 released on September 14, 2015.

New Features:

- None.

Bugs Fixed:

- Fixed a bug introduced in HTCondor version 8.3.7 that caused the *condor_shared_port* daemon to leak file descriptors. Also made HTCondor work better when some HTCondor daemons are using shared port, but the *condor_master* is not. (Ticket #5259).
- The *condor_starter* lowers the OOM (out of memory) score of jobs so the OOM killer is more likely to chose an HTCondor job rather than an HTCondor daemon or other user process. (Ticket #5249).
- Job submission fails if X.509 certificates are advertised with EC2 grid universe jobs. Therefore EC2 grid universe jobs no longer advertise their access keys. (Ticket #5252).

Chapter 11

Command Reference Manual (man pages)

bosco_cluster

Manage and configure the clusters to be accessed.

Synopsis

bosco_cluster [**−h** || **−−help**]

bosco_cluster [**−l** || **−−list**] [**−a** || **−−add <host>** [**schedd**]] [**−r** || **−−remove <host>**] [**−s** || **−−status <host>**] [**−t** || **−−test <host>**]

Description

bosco_cluster is part of the Bosco system for accessing high throughput computing resources from a local desktop. For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/

bosco_cluster enables management and configuration of the computing resources the Bosco tools access; these are called clusters.

A **<host>** is of the form `user@fqdn.example.com`.

Options

—help Print usage information and exit.

—list List all installed clusters.

—remove <*host*> Remove an already installed cluster, where the cluster is identified by <*host*>.

—add <*host*> *[scheduler]* Install and add a cluster defined by <*host*>. The optional *scheduler* specifies the scheduler on the cluster. Valid values are `pbs`, `lsf`, `condor`, `sge` or `slurm`. If not given, the default will be `pbs`.

—status <*host*> Query and print the status of an already installed cluster, where the cluster is identified by <*host*>.

—test <*host*> Attempt to submit a test job to an already installed cluster, where the cluster is identified by <*host*>.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

bosco_findplatform

Synopsis

bosco_findplatform [**−h** || **−−help**]

bosco_findplatform [**−u** || **−−url**] [**−b** || **−−bit**] [**−f** || **−−full**] [**−−force=<platformstring>**] [**−i** || **−−install <installoptions>**]

Description

bosco_findplatform is part of the Bosco system for accessing high throughput computing resources from a local desktop.

This command is not meant to be executed on the command line by users.

For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/

Options

—help Print usage information and exit.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

bosco_install

Synopsis

bosco_install [−−**help**] | [−−**usage**]

bosco_install [−−**install[=<path/to/release_dir>]**] [−−**prefix=<path>**] [−−**install-dir=<path>**] [−−**local-dir=<path>**] [−−**make-personal-condor**] [−−**bosco**] [−−**type=<[submit][,execute][,manager]>**] [−−**central-manager=<host>**] [−−**credd**] [−−**owner=<username>**] [−−**maybe-daemon-owner**] [−−**install-log=<file>**] [−−**overwrite**] [−−**env-scripts-dir=<dir>**] [−−**no-env-scripts**] [−−**ignore-missing-libs**] [−−**force**] [−−**backup**] [−−**verbose**]

Description

bosco_install is part of the Bosco system for accessing high throughput computing resources from a local desktop. For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/

bosco_install is linked to *condor_install*. The command

```
bosco_install
```

becomes

```
condor_install --bosco
```

Please see the *condor_install* man page for details of the command line options.

A Personal HTCondor specialized for Bosco is installed, permitting central manager tasks and job submission.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

bosco_ssh_start

Synopsis

bosco_ssh_start

Description

bosco_ssh_start is part of the Bosco system for accessing high throughput computing resources from a local desktop.

This command is not meant to be executed on the command line by users.

For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

bosco_start

start up the Personal HTCondor installation specific to Bosco

Synopsis

bosco_start

Description

bosco_start is part of the Bosco system for accessing high throughput computing resources from a local desktop. For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/

After installation, *bosco_start* invokes the daemons of the Personal HTCondor installation specific to the Bosco implementation.

There are no command line arguments to this script.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

bosco_stop

Shut down HTCondor daemons in a Bosco installation.

Synopsis

bosco_stop

Description

bosco_stop is part of the Bosco system for accessing high throughput computing resources from a local desktop. For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/.

bosco_stop shuts down the HTCondor daemons that are installed and running as part of the Personal HTCondor. It is the equivalent of *condor_off*.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

bosco_uninstall

uninstall a Bosco installation

Synopsis

bosco_uninstall

bosco_uninstall is part of the Bosco system for accessing high throughput computing resources from a local desktop. For detailed information, please see the Bosco web site: https://osg-bosco.github.io/docs/.

bosco_uninstall removes the Bosco software, but leaves files in the `.bosco` and `.ssh` directories.

There are no command line arguments to this script.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_advertise

Send a ClassAd to the *condor_collector* daemon

Synopsis

condor_advertise [**-help** | **-version**]

condor_advertise [**-pool** *centralmanagerhostname[:portname]*] [**-debug**] [**-tcp**] [**-udp**] [**-multiple**] *update-command* [*classad-filename*]

Description

condor_advertise sends one or more ClassAds 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(s) should be read. If *classad-filename* is omitted or is the dash character ('-'), then the ClassAd(s) are read from standard input.

When **-multiple** is specified, multiple ClassAds may be published. Publishing many ClassAds in a single invocation of *condor_advertise* is more efficient than invoking *condor_advertise* once per ClassAd. The ClassAds are expected to be separated by one or more blank lines. When **-multiple** is not specified, blank lines are ignored (for backward compatibility). It is best not to rely on blank lines being ignored, as this may change in the future.

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_SRVR_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

-debug Print debugging information as the command executes.

-multiple Send more than one ClassAd, where the boundary between ClassAds is one or more blank lines.

-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. Used by default if `UPDATE_COLLECTOR_WITH_TCP` is true.

-udp Use UDP for communication.

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 HTCondor 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.

Attributes are defined in Appendix A of the HTCondor manual.

For those administrators who do need *condor_advertise*, the following attributes may be included:

`DaemonStartTime`

`UpdateSequenceNumber`

If both of the above are included, the *condor_collector* will automatically include the following attributes:

`UpdatesTotal`

`UpdatesLost`

`UpdatesSequenced`

`UpdatesHistory` Affected by `COLLECTOR_DAEMON_HISTORY_SIZE`.

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. Success means that all ClassAds were successfully sent to all *condor_collector* daemons. When there are multiple ClassAds or multiple *condor_collector* daemons, it is possible that some but not all publications succeed; in this case, the exit status is 1, indicating failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_check_userlogs

Check job event log files for errors

Synopsis

condor_check_userlogs *UserLogFile1* [*UserLogFile2 . . . UserLogFileN*]

Description

condor_check_userlogs is a program for checking a job event log or a set of job event 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_checkpoint

send a checkpoint command to jobs running on specified hosts

Synopsis

condor_checkpoint [**-help** | **-version**]

condor_checkpoint [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-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 HTCondor 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 HTCondor 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 *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_chirp

Access files or job ClassAd from an executing job

Synopsis

***condor_chirp* <Chirp-Command>**

Description

condor_chirp is not intended for use as a command-line tool. It is most often invoked by an HTCondor job, while the job is 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.

When invoked by an HTCondor job, the command-line arguments describe the operation to be performed. Each of these arguments is described below within the section on Chirp Commands. Descriptions using the terms *local* and *remote* are given from the point of view of the executing job.

If the input file name for **put** or **write** 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 ClassAd. To do this, place

```
+WantIOProxy = true
```

in the submit description file of the job.

condor_chirp only works for jobs run in the vanilla, parallel and java universes.

Chirp Commands

fetch *RemoteFileName LocalFileName* Copy the *RemoteFileName* from the submit machine to the execute machine, naming it *LocalFileName*.

put [-mode *mode*] [-perm *UnixPerm*] *LocalFileName RemoteFileName* Copy the *LocalFileName* from the execute machine to the submit machine, naming it *RemoteFileName*. The optional **-perm** *UnixPerm* argument describes the file access permissions in a Unix format; 660 is an example Unix format.

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.

remove *RemoteFileName* Remove the *RemoteFileName* file from the submit machine.

get_job_attr *JobAttributeName* Prints the named job ClassAd attribute to standard output.

set_job_attr *JobAttributeName AttributeValue* Sets the named job ClassAd attribute with the given attribute value.

get_job_attr_delayed *JobAttributeName* Prints the named job ClassAd attribute to standard output, potentially reading the cached value from a recent set_job_attr_delayed.

set_job_attr_delayed *JobAttributeName AttributeValue* Sets the named job ClassAd attribute with the given attribute value, but does not immediately synchronize the value with the submit side. It can take 15 minutes before the synchronization occurs. This has much less overhead than the non delayed version. With this option, jobs do *not* need ClassAd attribute `WantIOProxy` set. With this option, job attribute names are restricted to begin with the case sensitive substring `Chirp`.

ulog *Message* Appends *Message* to the job event log.

read [-offset *offset*] [-stride *length skip*] *RemoteFileName Length* Read *Length* bytes from *RemoteFileName*. Optionally, implement a stride by starting the read at *offset* and reading *length* bytes with a stride of *skip* bytes.

write [-offset *offset*] [-stride *length skip*] *RemoteFileName LocalFileName* [*numbytes*] Write the contents of *LocalFileName* to *RemoteFileName*. Optionally, start writing to the remote file at *offset* and write *length* bytes with a stride of *skip* bytes. If the optional *numbytes* follows *LocalFileName*, then the write will halt after *numbytes* input bytes have been written. Otherwise, the entire contents of *LocalFileName* will be written.

rmdir [-r] *RemotePath* Delete the directory specified by *RemotePath*. If the optional **-r** is specified, recursively delete the entire directory.

getdir [-l] *RemotePath* List the contents of the directory specified by *RemotePath*. If *-l* is specified, list all metadata as well.

whoami Get the user's current identity.

whoareyou *RemoteHost* Get the identity of *RemoteHost*.

link [-s] *OldRemotePath NewRemotePath* Create a hard link from *OldRemotePath* to *NewRemotePath*. If the optional *-s* is specified, create a symbolic link instead.

readlink *RemoteFileName* Read the contents of the file defined by the symbolic link *RemoteFileName*.

stat *RemotePath* Get metadata for *RemotePath*. Examines the target, if it is a symbolic link.

lstat *RemotePath* Get metadata for *RemotePath*. Examines the file, if it is a symbolic link.

statfs *RemotePath* Get file system metadata for *RemotePath*.

access *RemotePath Mode* Check access permissions for *RemotePath*. *Mode* is one or more of the characters `r`, `w`, `x`, or `f`, representing read, write, execute, and existence, respectively.

chmod *RemotePath UnixPerm* Change the permissions of *RemotePath* to *UnixPerm*. *UnixPerm* describes the file access permissions in a Unix format; 660 is an example Unix format.

chown *RemotePath UID GID* Change the ownership of *RemotePath* to *UID* and *GID*. Changes the target of *RemotePath*, if it is a symbolic link.

chown *RemotePath UID GID* Change the ownership of *RemotePath* to *UID* and *GID*. Changes the link, if *RemotePath* is a symbolic link.

truncate *RemoteFileName Length* Truncates *RemoteFileName* to *Length* bytes.

utime *RemotePath AccessTime ModifyTime* Change the access to *AccessTime* and modification time to *ModifyTime* of *RemotePath*.

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 called `RemoteFile` on the submit machine, run

```
echo foo | condor_chirp put -mode wa - RemoteFile
```

To append the message "Hello World" to the job event 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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"
```

The HTCondor manual has 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_compile

create a relinked executable for use as a standard universe job

Synopsis

condor_compile *cc* | *CC* | *gcc* | *f77* | *g++* | *ld* | *make* | ...

Description

Use *condor_compile* to relink a program with the HTCondor libraries for submission as a standard universe job. The HTCondor libraries provide the program with additional support, such as the capability to produce checkpoints, which facilitate the 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, then the program must use another universe, such as vanilla. Source or object code may not be available if there is only an executable binary, or if a shell script is to be executed as an HTCondor job.

To use *condor_compile*, issue the command *condor_compile* with command line arguments that form the normally entered command to compile or link the application. Resulting executables will have the HTCondor libraries linked in. For example,

```
condor_compile cc -O -o myprogram.condor file1.c file2.c ...
```

will produce the binary `myprogram.condor`, which is relinked for HTCondor, capable of checkpoint/migration/remote system calls, and ready to submit as a standard universe job.

If the HTCondor 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 HTCondor 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)

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)
```

NOTE: If you explicitly call *ld* when you normally create your binary, instead use:

```
condor_compile ld <ld arguments and options>
```

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_config_val

Query or set a given HTCondor configuration variable

Synopsis

condor_config_val <*help option*>

condor_config_val [**<location options>**] <*edit option*>

condor_config_val [**<location options>**] [**<view options>**] *vars*

condor_config_val **use** *category*[*:template_name*] [**-expand**]

Description

condor_config_val can be used to quickly see what the current HTCondor configuration is on any given machine. Given a space separated set of configuration variables with the *vars* argument, *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. Variables and values may instead be queried from a daemon specified using a **location option**.

Raw output of *condor_config_val* displays the string used to define the configuration variable. This is what is on the right hand side of the equals sign (=) in a configuration file for a variable. The default output is an *expanded* one. Expanded output recursively replaces any macros within the raw definition of a variable with the macro's raw definition.

Each daemon remembers settings made by a successful invocation of *condor_config_val*. The configuration *file* is not modified.

condor_config_val can be used to persistently set or unset configuration variables for a specific daemon on a given machine using a *-set* or *-unset* **edit option**. Persistent settings remain when the daemon is restarted. Configuration variables for a specific daemon on a given machine may be set or unset for the time period that the daemon continues to run using a *-rset* or *-runset* **edit option**. These runtime settings will override persistent settings until the daemon is restarted. Any changes made will not take effect until *condor_reconfig* is invoked.

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 3.8.1 on page 395 for more details on security settings. Further, security considerations require proper settings of configuration variables `SETTABLE_ATTRS_<PERMISSION-LEVEL>` (see 3.5.3), `ENABLE_PERSISTENT_CONFIG` (see 3.5.3), and `HOSTALLOW...` (see 3.5.3) in order to use *condor_config_val* to change any configuration variable.

It is generally wise to test a new configuration on a single machine to ensure that no syntax or other errors in the configuration have been made before the reconfiguration of many machines. Having bad syntax or invalid configuration settings is a fatal error for HTCondor daemons, and they will exit. It is far better to discover such a problem on a single

machine than to cause all the HTCondor daemons in the pool to exit. *condor_config_val* can help with this type of testing.

Options

-help (help option) Print usage information and exit.

-version (help option) Print the HTCondor version information and exit.

-set *"var = value"* (edit option) Sets one or more persistent configuration file variables. The new value remains if the daemon is restarted. One or more variables can be set; the syntax requires double quote marks to identify the pairing of variable name to value, and to permit spaces.

-unset *var* (edit option) Each of the persistent configuration variables listed reverts to its previous value.

-rset *"var = value"* (edit option) Sets one or more configuration file variables. The new value remains as long as the daemon continues running. One or more variables can be set; the syntax requires double quote marks to identify the pairing of variable name to value, and to permit spaces.

-runset *var* (edit option) Each of the configuration variables listed reverts to its previous value as long as the daemon continues running.

-dump (view option) Display the raw value of all *vars* listed. If no *vars* are listed, then print all configuration variables and their values. The **-expand**, **-default**, and **-evaluate** options take precedence over this **-dump** option, such that the output will not be raw.

-default (view option) Default values are displayed.

-expand (view option) Expanded values are displayed. This is the default.

-raw (view option) Raw values are displayed.

-verbose (view option) Display configuration file name and line number where the variable is set, along with the raw, expanded, and default values of the variable.

-debug[:<opts>] (view option) Send output to `stderr`, overriding a set value of `TOOL_DEBUG`.

-evaluate (view option) Applied only when a **location option** specifies a daemon. The value of the requested parameter will be evaluated with respect to the ClassAd of that daemon.

-used (view option) Applied only when a **location option** specifies a daemon. Modifies which variables are displayed to only those used by the specified daemon.

-unused (view option) Applied only when a **location option** specifies a daemon. Modifies which variables are displayed to only those *not* used by the specified daemon.

-config (view option) Applied only when the configuration is read from files (the default), and *not* when applied to a specific daemon. Display the current configuration file that set the variable.

-writeconfig[:upgrade] ***filename*** (view option) For the configuration read from files (the default), write to file *filename* all configuration variables. Values that are the same as internal, compile-time defaults will be preceded by the comment character. If the **:upgrade** *o*ption is specified, then values that are the same as the internal, compile-time defaults are omitted. Variables are in the same order as the they were read from the original configuration files.

-mixedcase (view option) Applied only when the configuration is read from files (the default), and *not* when applied to a specific daemon. Print variable names with the same letter case used in the variable's definition.

-local-name <*name*> (view option) Applied only when the configuration is read from files (the default), and *not* when applied to a specific daemon. Inspect the values of attributes that use local names, which is useful to distinguish which daemon when there is more than one of the particular daemon running.

-subsystem <*daemon*> (view option) Applied only when the configuration is read from files (the default), and *not* when applied to a specific daemon. Specifies the subsystem or daemon name to query, with a default value of the `TOOL` subsystem.

-address <*ip:port*> (location option) Connect to the given IP address and port number.

-pool ***centralmanagerhostname[:portnumber]*** (location option) Use the given central manager and an optional port number to find daemons.

-name <*machine_name*> (location option) Query the specified machine's *condor_master* daemon for its configuration. Does not function together with any of the options: **-dump**, **-config**, or **-verbose**.

-master | **-schedd** | **-startd** | **-collector** | **-negotiator** (location option) The specific daemon to query.

use *category*[*:set name*] [-expand] Display information about configuration templates (see 3.4). Specifying only a *category* will list the *template_names* available for that category. Specifying a *category* and a *template_name* will display the definition of that configuration template. Adding the **-expand** option will display the expanded definition (with macro substitutions). (**-expand** has no effect if a *template_name* is not specified.) Note that there is no dash before **use** and that spaces are not allowed next to the colon character separating *category* and *template_name*.

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

Here is a set of examples to show a sequence of operations using *condor_config_val*. To request the *condor_schedd* daemon on host perdita to display the value of the MAX_JOBS_RUNNING configuration variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

To request the *condor_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 is value before initial set of the variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

Getting a list of template_names for the **role** configuration template category:

```
% condor_config_val use role
use ROLE accepts
  CentralManager
  Execute
  Personal
  Submit
```

Getting the definition of **role:personal** configuration template:

```
% condor_config_val use role:personal
use ROLE:Personal is
    CONDOR_HOST=127.0.0.1
COLLECTOR_HOST=$(CONDOR_HOST):0
DAEMON_LIST=MASTER COLLECTOR NEGOTIATOR STARTD SCHEDD
RunBenchmarks=0
```

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_configure

Configure or install HTCondor

Synopsis

condor_configure **or** ***condor_install*** [−−**help**] [−−**usage**]

condor_configure or ***condor_install*** [−−**install[=<path/to/release>]**] [−−**install-dir=<path>**] [−−**prefix=<path>**] [−−**local-dir=<path>**] [−−**make-personal-condor**] [−−**bosco**] [−−**type = < submit, execute, manager >**] [−−**central-manager = < hostname>**] [−−**owner = < ownername >**] [−−**maybe-daemon-owner**] [−−**install-log = < file >**] [−−**overwrite**] [−−**ignore-missing-libs**] [−−**force**] [−−**no-env-scripts**] [−−**env-scripts-dir = < directory >**] [−−**backup**] [−−**credd**] [−−**verbose**]

Description

condor_configure and *condor_install* refer to a single script that installs and/or configures HTCondor on Unix machines. As the names imply, *condor_install* is intended to perform a HTCondor 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 HTCondor 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 HTCondor is running. To effect changes while HTCondor 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 HTCondor. This list includes: `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src`.

Options

—help Print help screen and exit

—usage Print short usage and exit

—install[=<path/to/release>] Perform installation, assuming that the current working directory contains the release directory, if the optional =<path/to/release> is not specified. Without further options, the configuration is that of a Personal HTCondor, 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 HTCondor 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 HTCondor 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 HTCondor, a fully-functional, one-machine pool.

—bosco Installs and configures Bosco, a personal HTCondor that submits jobs to remote batch systems.

—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 HTCondor, the machine fulfills all three of these roles.

—central-manager=<hostname> Instructs the current HTCondor installation to use the specified machine as the central manager. This modifies the configuration variable `COLLECTOR_HOST` to point to the given host name. The central manager machine's HTCondor configuration needs to be independently configured to act as a manager using the option **–type=manager**.

—owner=<ownername> Set configuration such that HTCondor 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 HTCondor daemons start up as the specified effective user. The section on security within the HTCondor manual discusses UIDs in HTCondor. 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.

—maybe-daemon-owner If **–owner** is not specified and no appropriate user can be found to run Condor, then this option will allow the daemon user to be selected. This option is rarely needed by users but can be useful for scripts that invoke condor_configure to install Condor.

—install-log=<file> Save information about the installation in the specified file. This is normally only needed when condor_configure is called by a higher-level script, not when invoked by a person.

—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 HTCondor 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 HTCondor 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 HTCondor programs in it. In this case, *condor_install* with 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 HTCondor 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 HTCondor installation. This option causes *condor_configure* to generate these scripts in the specified directory.

—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 HTCondor on the machine (machine1@cs.wisc.edu) to be the pool's central manager. On machine1, within the directory that contains the unzipped HTCondor distribution directories:

```
% condor_install --type=submit,execute,manager
```

This will allow the machine to submit and execute HTCondor 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 HTCondor 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 HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_continue

continue suspended jobs from the HTCondor queue

Synopsis

condor_continue [**-help** | **-version**]

condor_continue [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] *cluster* | *cluster.process* | *user* | ***-constraint*** *expression* | *-all*

Description

condor_continue continues one or more suspended jobs from the HTCondor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The job(s) to be continued are identified by one of the 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 continue 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 Continue all jobs in the specified cluster

cluster.process Continue the specific job in the cluster

user Continue jobs belonging to specified user

-constraint *expression* Continue all jobs which match the job ClassAd expression constraint

-all Continue all the jobs in the queue

Exit Status

condor_continue will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To continue all jobs except for a specific user:

```
% condor_continue -constraint 'Owner =!= "foo"'
```

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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.

The *condor_convert_history* command can also be used to reconstruct the new format in a history file that has been corrupted or concantenated with another history file.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_dagman

meta scheduler of the jobs submitted as the nodes of a DAG or DAGs

Synopsis

condor_dagman *-f -t -l .* **-help**

condor_dagman **-version**

condor_dagman *-f -l .* **-csdversion** *version_string* [**-debug** *level*] [**-maxidle** *numberOfProcs*] [**-maxjobs** *numberOfJobs*] [**-maxpre** *NumberOfPreScripts*] [**-maxpost** *NumberOfPostScripts*] [**-noeventchecks**] [**-allowlogerror**] [**-usedagdir**] **-lockfile** *filename* [**-waitfordebug**] [**-autorescue** *0|1*] [**-dorescuefrom** *number*] [**-allowversionmismatch**] [**-DumpRescue**] [**-verbose**] [**-force**] [**-notification** *value*] [**-suppress_notification**] [**-dont_suppress_notification**] [**-dagman** *DagmanExecutable*] [**-outfile_dir** *directory*] [**-update_submit**] [**-import_env**] [**-priority** *number*] [**-dont_use_default_node_log**] [**-DontAlwaysRunPost**] [**-AlwaysRunPost**] [**-DoRecovery**] **-dag** *dag_file* [***-dag*** *dag_file_2 . . .* ***-dag*** *dag_file_n*]

Description

condor_dagman is a meta scheduler for the HTCondor 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 an HTCondor job. *condor_dagman* uses log files to coordinate the further submission of the jobs within the DAG.

All command line arguments to the *DaemonCore* library functions work for *condor_dagman*. When invoked from the command line, *condor_dagman* requires the arguments *-f -l .* to appear first on the command line, to be processed by *DaemonCore*. The **csdversion** must also be specified; at start up, *condor_dagman* checks for a version mismatch with the *condor_submit_dag* version in this argument. The *-t* argument must also be present for the **-help** option, such that output is sent to the terminal.

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** *a*rguments. 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; output DAG input file lines as they are parsed
- level = 7; internal debugging output; rarely used; output DAG input file lines as they are parsed

Options

-help Display usage information and exit.

-version Display version information and exit.

-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.

-maxidle ***NumberOfProcs*** Sets the maximum number of idle procs allowed before *condor_dagman* stops submitting more node jobs. Note that for this argument, each individual proc within a cluster counts as a towards the limit, which is inconsistent with **-maxjobs** . Once idle procs start to run, *condor_dagman* will resume submitting jobs once the number of idle procs falls below the specified limit. *NumberOfProcs* is a non-negative integer. If this option is omitted, the number of idle procs is limited by the configuration variable `DAGMAN_MAX_JOBS_IDLE` (see 3.5.23), which defaults to 1000. To disable this limit, set *NumberOfProcs* to 0. Note that submit description files that queue multiple procs can cause the *NumberOfProcs* limit to be exceeded. Setting `queue 5000` in the submit description file, where *-maxidle* is set to 250 will result in a cluster of 5000 new procs being submitted to the *condor_schedd*, not 250. In this case, *condor_dagman* will resume submitting jobs when the number of idle procs falls below 250.

-maxjobs ***NumberOfClusters*** Sets the maximum number of clusters within the DAG that will be submitted to HTCondor at one time. Note that for this argument, each cluster counts as one job, no matter how many individual procs are in the cluster. *NumberOfClusters* is a non-negative integer. If this option is omitted, the number of clusters is limited by the configuration variable `DAGMAN_MAX_JOBS_SUBMITTED` (see 3.5.23), which defaults to 0 (unlimited).

-maxpre ***NumberOfPreScripts*** Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPreScripts* is a non-negative integer. If this option is omitted, the number of PRE scripts is limited by the configuration variable `DAGMAN_MAX_PRE_SCRIPTS` (see 3.5.23), which defaults to 20.

-maxpost *NumberOfPostScripts* Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPostScripts* is a non-negative integer. If this option is omitted, the number of POST scripts is limited by the configuration variable `DAGMAN_MAX_POST_SCRIPTS` (see 3.5.23), which defaults to 20.

-noeventchecks This argument is no longer used; it is now ignored. Its functionality is now implemented by the `DAGMAN_ALLOW_EVENTS` configuration variable.

-allowlogerror As of verson 8.5.5 this argument is no longer supported, and setting it will generate a warning.

-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 nonexistent 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. (Note that, starting with version 7.4.0, *condor_dagman* no longer requires an exact version match between itself and the `.condor.sub` file. Instead, a "minimum compatible version" is defined, and any `.condor.sub` file of that version or newer is accepted.)

-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. The Rescue DAG file is produced whether or not there are parse errors reading the original DAG input file. The name of the file differs if there was a parse error.

-verbose (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) Cause *condor_submit_dag* to give verbose error messages.

-force (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) 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 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 is in effect, any existing old-style rescue DAGs will be deleted, and the original DAG will be run. See the HTCondor manual section on Rescue DAGs for more information.

-notification *value* This argument is only included to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs. Sets the e-mail notification for DAGMan itself. This information will be used within the HTCondor submit description file for DAGMan. This file is produced by *condor_submit_dag*. The **notification** option is described in the *condor_submit* manual page.

-suppress_notification Causes jobs submitted by *condor_dagman* to not send email notification for events. The same effect can be achieved by setting the configuration variable `DAGMAN_SUPPRESS_NOTIFICATION` to `True`. This command line option is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself. This flag is generally superfluous, as `DAGMAN_SUPPRESS_NOTIFICATION` defaults to `True`.

-dont_suppress_notification Causes jobs submitted by *condor_dagman* to defer to content within the submit description file when deciding to send email notification for events. The same effect can be achieved by setting the configuration variable `DAGMAN_SUPPRESS_NOTIFICATION` to `False`. This command line flag is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself. If both **-dont_suppress_notification** and **-suppress_notification** are specified within the same command line, the last argument is used.

-dagman *DagmanExecutable* (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) 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.

-outfile_dir *directory* (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) 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.

-update_submit (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) 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.

-import_env (This argument is included only to be passed to *condor_submit_dag* if lazy submit file generation is used for nested DAGs.) This optional argument causes *condor_submit_dag* to import the current environment into the **environment** command of the `.condor.sub` file it generates.

-priority *number* Sets the minimum job priority of node jobs submitted and running under this *condor_dagman* job.

-dont_use_default_node_log This option is disabled as of HTCondor version 8.3.1. Tells *condor_dagman* to use the file specified by the job ClassAd attribute `UserLog` to monitor job status. If this command line argument is used, then the job event log file cannot be defined with a macro.

-DontAlwaysRunPost This option causes *condor_dagman* to not run the POST script of a node if the PRE script fails. (This was the default behavior prior to HTCondor version 7.7.2, and is again the default behavior from version 8.5.4 onwards.)

-AlwaysRunPost This option causes *condor_dagman* to always run the POST script of a node, even if the PRE script fails. (This was the default behavior for HTCondor version 7.7.2 through version 8.5.3.)

-DoRecovery Causes *condor_dagman* to start in recovery mode. This means that it reads the relevant job user log(s) and catches up to the given DAG's previous state before submitting any new jobs.

-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 an HTCondor job by running condor_submit_dag. See the condor_submit_dag manual page 938 for examples.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_dagman_metrics_reporter

Report the statistics of a DAGMan run to a central HTTP server

Synopsis

condor_dagman_metrics_reporter [**-s**] [**-u** *URL*] [**-t** *maxtime*] **-f** */path/to/metrics/file*

Description

condor_dagman_metrics_reporter anonymously reports metrics from a DAGMan workflow to a central server. The reporting of workflow metrics is only enabled for DAGMan workflows run under Pegasus; metrics reporting has been requested by Pegasus' funding sources: see http://pegasus.isi.edu/wms/docs/latest/funding_citing_usage.php#usage_statistics and https://confluence.pegasus.isi.edu/display/pegasus/DAGMan+Metrics+Reporting for the requirements to collect this data.

The data sent to the server is in JSON format. Here is an example of what is sent:


```
{
    "client":"condor_dagman",
    "version":"8.1.0",
    "planner":"/lfs1/devel/Pegasus/pegasus/bin/pegasus-plan",
    "planner_version":"4.3.0cvs",
    "type":"metrics",
    "wf_uuid":"htcondor-test-job_dagman_metrics-A-subdag",
    "root_wf_uuid":"htcondor-test-job_dagman_metrics-A",
    "start_time":1375313459.603,
    "end_time":1375313491.498,
    "duration":31.895,
    "exitcode":1,
    "dagman_id":"26",
    "parent_dagman_id":"11",
    "rescue_dag_number":0,
    "jobs":4,
    "jobs_failed":1,
    "jobs_succeeded":3,
    "dag_jobs":0,
    "dag_jobs_failed":0,
    "dag_jobs_succeeded":0,
    "total_jobs":4,
    "total_jobs_run":4,
    "total_job_time":0.000,
```

```
    "dag_status":2
}
```

Metrics are sent only if the *condor_dagman* process has `PEGASUS_METRICS` set to `True` in its environment, and the `CONDOR_DEVELOPERS` configuration variable does *not* have the value `NONE`.

Ordinarily, this program will be run by *condor_dagman*, and users do not need to interact with it. This program uses the following environment variables:

PEGASUS_USER_METRICS_DEFAULT_SERVER The URL of the default server to which to send the data. It defaults to `http://metrics.pegasus.isi.edu/metrics`. It can be overridden at the command line with the **-u** option.

PEGASUS_USER_METRICS_SERVER A comma separated list of URLs of servers that will receive the data, in addition to the default server.

The *-f* argument specifies the metrics file to be sent to the HTTP server.

Options

-s Sleep for a random number of seconds between 1 and 10, before attempting to send data. This option is used to space out the reporting from any sub-DAGs when a DAG is removed.

-u *URL* Overrides setting of the environment variable `PEGASUS_USER_METRICS_DEFAULT_SERVER`. This option is unused by *condor_dagman*; it is for testing by developers.

-t *maxtime* A maximum time in seconds that defaults to 100 seconds, setting a limit on the amount of time this program will wait for communication from the server. A setting of zero will result in a single attempt per server. *condor_dagman* retrieves this value from the `DAGMAN_PEGASUS_REPORT_TIMEOUT` configuration variable.

-f *metrics_file* The name of the file containing the metrics values to be reported.

Exit Status

condor_dagman_metrics_reporter will exit with a status value of 0 (zero) upon success, and it will exit with a value of 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_drain

Control draining of an execute machine

Synopsis

condor_drain [**-help**]

condor_drain [**-debug**] [**-pool** *pool-name*] [**-graceful** | **-quick** | **-fast**] [**-resume-on-completion**] [**-check** *expr*] *machine-name*

condor_drain [**-debug**] [**-pool** *pool-name*] **-cancel** [**-request-id** *id*] *machine-name*

Description

condor_drain is an administrative command used to control the draining of all slots on an execute machine. When a machine is draining, it will not accept any new jobs. Which machine to drain is specified by the argument *machine-name*, and will be the same as the machine ClassAd attribute `Machine`.

How currently running jobs are treated depends on the draining schedule that is chosen with a command-line option:

-graceful Initiate a graceful eviction of the job. This means all promises that have been made to the job are honored, including `MaxJobRetirementTime`. The eviction of jobs is coordinated to reduce idle time. This means that if one slot has a job with a long retirement time and the other slots have jobs with shorter retirement times, the effective retirement time for all of the jobs is the longer one. If no draining schedule is specified, **-graceful** is chosen by default.

-quick `MaxJobRetirementTime` is not honored. Eviction of jobs is immediately initiated. Jobs are given time to shut down and produce checkpoints, according to the usual policy, that is, given by `MachineMaxVacateTime`.

-fast Jobs are immediately hard-killed, with no chance to gracefully shut down or produce a checkpoint.

Once draining is complete, the machine will enter the Drained/Idle state. To resume normal operation (negotiation) at that time or any previous time during draining, the **-cancel** option may be used. The **-resume-on-completion** option results in automatic resumption of normal operation once draining has completed, and may be used when initiating draining. This is useful for forcing a machine with a partitionable slots to join all of the resources back together into one machine, facilitating de-fragmentation and whole machine negotiation.

Options

-help Display brief usage information and exit.

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-pool *pool-name* Specify an alternate HTCondor pool, if the default one is not desired.

-graceful (the default) Honor the maximum vacate and retirement time policy.

-quick Honor the maximum vacate time, but not the retirement time policy.

-fast Honor neither the maximum vacate time policy nor the retirement time policy.

-resume-on-completion When done draining, resume normal operation, such that potentially the whole machine could be claimed.

-check *expr* Abort draining, if `expr` is not true for all slots to be drained.

-cancel Cancel a prior draining request, to permit the *condor_negotiator* to use the machine again.

-request-id *id* Specify a specific draining request to cancel, where *id* is given by the `DrainingRequestId` machine ClassAd attribute.

Exit Status

condor_drain will exit with a non-zero status value if it fails and zero status if it succeeds.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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 HTCondor 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 `ADMINISTRATOR` level of access.

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.

The *subsystem* argument is any value `$(SUBSYSTEM)` that has a defined configuration variable of `$(SUBSYSTEM)_LOG`, or any of

- `NEGOTIATOR_MATCH`
- `HISTORY`
- `STARTD_HISTORY`

A value for the optional *.extension* to the *subsystem* argument is typically one of the three strings:

1. `.old`
2. `.slot<X>`
3. `.slot<X>.old`

Within these strings, `<X>` is substituted with the slot number.

A *subsystem* argument of `STARTD_HISTORY` fetches all *condor_startd* history by concatenating all instances of log files resulting from rotation.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_findhost

find machine(s) in the pool that can be used with minimal impact on currently running HTCondor 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 an HTCondor pool of machines for the best machine or machines that will have the minimum impact on running HTCondor 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 HTCondor pool. The expression

```
(Interactive =?= TRUE )
```

will let *condor_findhost* know that it can claim a machine even if HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_gather_info

Gather information about an HTCondor installation and a queued job

Synopsis

condor_gather_info [−−**jobid** *ClusterId.ProcId*] [−−**scratch** */path/to/directory*]

Description

condor_gather_info is a Linux-only tool that will collect and output information about the machine it is run upon, about the HTCondor installation local to the machine, and optionally about a specified HTCondor job. The information gathered by this tool is most often used as a debugging aid for the developers of HTCondor.

Without the −−**jobid** option, information about the local machine and its HTCondor installation is gathered and placed into the file called `condor-profile.txt`, in the current working directory. The information gathered is under the category of Identity.

With the −−**jobid** option, additional information is gathered about the job given in the command line argument and identified by its `ClusterId` and `ProcId` ClassAd attributes. The information includes both categories, Identity and Job information. As the quantity of information can be extensive, this information is placed into a compressed tar file. The file is placed into the current working directory, and it is named using the format

```
cgi-<username>-jid<ClusterId>.<ProcId>-<year>-<month>-<day>-<hour>_<minute>_<second>-<TZ>.tar.gz
```

All values within <> are substituted with current values. The building of this potentially large tar file can require a fair amount of temporary space. If the −−**scratch** option is specified, it identifies a directory in which to build the tar file. If the −−**scratch** option is *not* specified, then the directory will be `/tmp/cgi-<PID>`, where the process ID is that of the *condor_gather_info* executable.

The information gathered by this tool:

1. Identity
    - User name who generated the report
    - Script location and machine name
    - Date of report creation
    - `uname -a`
    - Contents of `/etc/issue`
    - Contents of `/etc/redhat-release`
    - Contents of `/etc/debian_version`
    - Contents of `$(LOG)/MasterLog`

- Contents of `$(LOG)/ShadowLog`
- Contents of `$(LOG)/SchedLog`
- Output of `ps -auxww -forest`
- Output of `df -h`
- Output of `iptables -L`
- Output of `ls 'condor_config_val LOG'`
- Output of `ldd 'condor_config_val SBIN'/condor_schedd`
- Contents of `/etc/hosts`
- Contents of `/etc/nsswitch.conf`
- Output of `ulimit -a`
- Output of `uptime`
- Output of `free`
- Network interface configuration (`ifconfig`)
- HTCondor version
- Location of HTCondor configuration files
- HTCondor configuration variables
  - All variables and values
  - Definition locations for each configuration variable

2. Job Information

   - Output of `condor_q jobid`
   - Output of `condor_q -l jobid`
   - Output of `condor_q -analyze jobid`
   - Job event log, if it exists
     - Only events pertaining to the job ID
   - If *condor_gather_info* has the proper permissions, it runs *condor_fetchlog* on the machine where the job most recently ran, and includes the contents of the logs from the *condor_master*, *condor_startd*, and *condor_starter*.

Options

—jobid *<ClusterId.ProcId>* Data mine information about this HTCondor job from the local HTCondor installation and *condor_schedd*.

—scratch *<path/to/directory>* A path to temporary space needed when building the output tar file. Defaults to `/tmp/cgi-<PID>`, where `<PID>` is replaced by the process ID of *condor_gather_info*.

Files

- `condor-profile.txt` The Identity portion of the information gathered when *condor_gather_info* is run without arguments.
- `cgi-<username>-jid<cluster>.<proc>-<year>-<month>-<day>-<hour>_<minute>_<second>-<T` The output file which contains all of the information produced by this tool.

Exit Status

condor_gather_info will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_gpu_discovery

Output GPU-related ClassAd attributes

Synopsis

***condor_gpu_discovery* -help**

condor_gpu_discovery [**<options>**]

Description

condor_gpu_discovery runs discovery software to determine the host's GPU capabilities, which are output as ClassAd attributes.

This tool is not available for MAC OS platforms.

With no command line options, the single ClassAd attribute `DetectedGPUs` is printed. If the value is 0, no GPUs were detected. If one or more GPUS were detected, the value is a string, presented as a comma and space separated list of the GPUs discovered, where each is given a name further used as the *prefix string* in other attribute names. Where there is more than one GPU of a particular type, the *prefix string* includes an integer value numbering the device; these integer values monotonically increase from 0. For example, a discovery of two GPUs may output

```
DetectedGPUs="CUDA0, CUDA1"
```

Further command line options use `"CUDA"` either with or without one of the integer values 0 or 1 as the *prefix string* in attribute names.

Options

-help Print usage information and exit.

-properties In addition to the `DetectedGPUs` attribute, display standard CUDA attributes. Each of these attribute names will have a *prefix string* at the beginning of its name. For a host with more than one of the same GPU type, those attribute values that are the same across all of the GPUs will not have an integer value in the *prefix string*. The attributes are `Capability`, `DeviceName`, `DriverVersion`, `ECCEnabled`, `GlobalMemoryMb`, and `RuntimeVersion`. The displayed standard Open CL attributes are `DeviceName`, `ECCEnabled`, `OpenCLVersion`, and `GlobalMemoryMb`.

-extra Display the additional attributes of Each of these attribute names will have a *prefix string* at the beginning of its name. `ClockMhz`, `ComputeUnits`, and `CoresPerCU` for a CUDA device, and `ClockMhz` and

`ComputeUnits` for an OCL device.

-dynamic Display attributes of NVIDIA devices that change values as the GPU is working. Each of these attribute names will have a *prefix string* at the beginning of its name. These are `FanSpeedPct`, `BoardTempC`, `DieTempC`, `EccErrorsSingleBit`, and `EccErrorsDoubleBit`.

-mixed When displaying attribute values, assume that the machine has a heterogeneous set of GPUs, so always include the integer value in the *prefix string*.

-device <*N*> Display properties only for GPU device <*N*>, where <*N*> is the integer value defined for the *prefix string*. Note that the attribute names in this output will *not* contain the value for <*N*>.

-tag *string* Set the resource tag portion of the intended machine ClassAd attribute `Detected<ResourceTag>` to be *string*. If this option is not specified, the resource tag is `"GPUs"`, resulting in attribute name `DetectedGPUs`.

-prefix *str* When naming attributes, use *str* as the *prefix string*. When this option is not specified, the *prefix string* is either `CUDA` or `OCL`.

-simulate:D,N For testing purposes, assume that N devices of type D were detected. No discovery software is invoked. If D is 0, it refers to GeForce GT 330, and a default value for N is 1. If D is 1, it refers to GeForce GTX 480, and a default value for N is 2.

-opencl Prefer detection via OpenCL rather than CUDA. Without this option, CUDA detection software is invoked first, and no further Open CL software is invoked if CUDA devices are detected.

-cuda Do only CUDA detection.

-nvcuda For Windows platforms only, use a CUDA driver rather than the CUDA run time.

-config Output in the syntax of HTCondor configuration, instead of ClassAd language. An additional attribute is produced `NUM_DETECTED_GPUs` which is set to the number of GPUs detected.

-verbose For interactive use of the tool, output extra information to show detection while in progress.

-diagnostic Show diagnostic information, to aid in tool development.

Exit Status

condor_gpu_discovery will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_history

View log of HTCondor jobs completed to date

Synopsis

condor_history [**-help**]

condor_history [**-name** *name*] [**-pool** *centralmanagerhostname[:portnumber]*] [**-backwards**] [**-forwards**] [**-constraint** *expr*] [**-file** *filename*] [**-userlog** *filename*] [**-format** *formatString AttributeName*] [**-autoformat[:jlhVr,tng]** *attr1 [attr2 ...]*] [**-l** | **-long** | **-xml** | **-json**] [**-match** | **-limit** *number*] [**cluster** | **cluster.process** | **owner**]

Description

condor_history displays a summary of all HTCondor jobs listed in the specified history files. If no history files are specified with the **-file** option, the local history file as specified in HTCondor's configuration file (`$(SPOOL)/history` by default) is read. The default listing summarizes in reverse 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, given as the job ClassAd attribute `RemoteWallClockTime`.

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 772 for details on converting history files to the new format.

Options

-help Display usage information and exit.

-name *name* Query the named *condor_schedd* daemon.

-pool *centralmanagerhostname[:portnumber]* Use the *centralmanagerhostname* as the central manager to locate *condor_schedd* daemons. The default is the `COLLECTOR_HOST`, as specified in the configuration.

-backwards List jobs in reverse chronological order. The job most recently added to the history file is first. This is the default ordering.

-forwards List jobs in chronological order. The job most recently added to the history file is last. At least 4 characters must be given to distinguish this option from the **-file** and **-format** options.

-constraint *expr* Display jobs that satisfy the expression.

-attributes *attrs* Display only the given attributes when the **-long** *o*ption is used.

-since *jobid or expr* Stop scanning when the given jobid is found or when the expression becomes true.

-local Read from local history files even if there is a `SCHEDD_HOST` configured.

-file *filename* Use the specified file instead of the default history file.

-userlog *filename* Display jobs, with job information coming from a job event log, instead of from the default history file. A job event log does not contain all of the job information, so some fields in the normal output of *condor_history* will be blank.

-format *formatString*AttributeName Display jobs with a custom format. See the *condor_q* man page **-format** option for details.

-autoformat[:jlhVr,tng] *attr1 [attr2 ...]* or -af[:jlhVr,tng] *attr1 [attr2 ...]* (output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

j print the job ID as the first field,
l label each field,
h print column headings before the first line of output,
V use %V rather than %v for formatting (string values are quoted),
r print "raw", or unevaluated values,
, add a comma character after each field,
t add a tab character before each field instead of the default space character,
n add a newline character after each field,
g add a newline character between ClassAds, and suppress spaces before each field.
Use **-af:h** to get tabular values with headings.
Use **-af:lrng** to get -long equivalent format.
The newline and comma characters may *not* be used together. The **l** and **h** characters may *not* be used together.

-l or -long Display job ClassAds in long format.

-limit *Number* Limit the number of jobs displayed to *Number*. Same option as **-match**.

-match *Number* Limit the number of jobs displayed to *Number*. Same option as **-limit**.

-xml Display job ClassAds in XML format. The XML format is fully defined in the reference manual, obtained from the ClassAds web page, with a link at http://htcondor.org/classad/classad.html.

-json Display job ClassAds in JSON format.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_hold

put jobs in the queue into the hold state

Synopsis

condor_hold [**-help** | **-version**]

condor_hold [**-debug**] [**-reason** *reasonstring*] [**-subcode** *number*] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] *cluster...* | *cluster.process...* | *user...* | **-constraint** *expression ...*

condor_hold [**-debug**] [**-reason** *reasonstring*] [**-subcode** *number*] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] **-all**

Description

condor_hold places jobs from the HTCondor 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`.

-reason *reasonstring* Sets the job ClassAd attribute `HoldReason` to the value given by *reasonstring*. *reasonstring* will be delimited by double quote marks on the command line, if it contains space characters.

-subcode *number* Sets the job ClassAd attribute `HoldReasonSubCode` to the integer value given by *number*.

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

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_install

Configure or install HTCondor

Synopsis

condor_configure **or** ***condor_install*** [−−**help**] [−−**usage**]

condor_configure or ***condor_install*** [−−**install[=<path/to/release>]**] [−−**install-dir=<path>**] [−−**prefix=<path>**] [−−**local-dir=<path>**] [−−**make-personal-condor**] [−−**bosco**] [−−**type = < submit, execute, manager >**] [−−**central-manager = < hostname>**] [−−**owner = < ownername >**] [−−**maybe-daemon-owner**] [−−**install-log = < file >**] [−−**overwrite**] [−−**ignore-missing-libs**] [−−**force**] [−−**no-env-scripts**] [−−**env-scripts-dir = < directory >**] [−−**backup**] [−−**credd**] [−−**verbose**]

Description

condor_configure and *condor_install* refer to a single script that installs and/or configures HTCondor on Unix machines. As the names imply, *condor_install* is intended to perform a HTCondor 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 HTCondor 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 HTCondor is running. To effect changes while HTCondor 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 HTCondor. This list includes: `bin`, `etc`, `examples`, `include`, `lib`, `libexec`, `man`, `sbin`, `sql` and `src`.

Options

—help Print help screen and exit

—usage Print short usage 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 HTCondor, 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 HTCondor 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 HTCondor 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 HTCondor, a fully-functional, one-machine pool.

—bosco Installs and configures Bosco, a personal HTCondor that submits jobs to remote batch systems.

—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 HTCondor, the machine fulfills all three of these roles.

—central-manager=<hostname> Instructs the current HTCondor installation to use the specified machine as the central manager. This modifies the configuration variable `COLLECTOR_HOST` to point to the given host name. The central manager machine's HTCondor configuration needs to be independently configured to act as a manager using the option **–type=manager**.

—owner=<ownername> Set configuration such that HTCondor 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 HTCondor daemons start up as the specified effective user. 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.

—maybe-daemon-owner If **–owner** is not specified and no appropriate user can be found to run Condor, then this option will allow the daemon user to be selected. This option is rarely needed by users but can be useful for scripts that invoke condor_configure to install Condor.

—install-log=<file> Save information about the installation in the specified file. This is normally only needed when condor_configure is called by a higher-level script, not when invoked by a person.

—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 HTCondor 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 HTCondor 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 HTCondor programs in it. In this case, *condor_install* with 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 HTCondor 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 HTCondor installation. This option causes *condor_configure* to generate these scripts in the specified directory.

—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 HTCondor on the machine (machine1@cs.wisc.edu) to be the pool's central manager. On machine1, within the directory that contains the unzipped HTCondor distribution directories:

```
% condor_install --type=submit,execute,manager
```

This will allow the machine to submit and execute HTCondor 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 HTCondor 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 HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_job_router_info

Discover and display information related to job routing

Synopsis

condor_job_router_info [**-help** | **-version**]

condor_job_router_info **-config**

condor_job_router_info **-match-jobs -jobads** ***filename*** [**-ignore-prior-routing**]

Description

condor_job_router_info displays information about job routing. The information will be either the available, configured routes or the routes for specified jobs.

Options

-help Display usage information and exit.

-version Display HTCondor version information and exit.

-config Display configured routes.

-match-jobs For each job listed in the file specified by the **-jobads** option, display the first route found.

-ignore-prior-routing For each job, remove any existing routing ClassAd attributes, and set attribute `JobStatus` to the Idle state before finding the first route.

-jobads ***filename*** Read job ClassAds from file *filename*. If *filename* is `-`, then read from `stdin`.

Exit Status

condor_job_router_info will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_master

The master HTCondor Daemon

Synopsis

condor_master

Description

This daemon is responsible for keeping all the rest of the HTCondor 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 HTCondor 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 HTCondor pool, regardless of what functions each machine are performing. Additionally, on Linux platforms, if you start the *condor_master* as root, it will tune (but never decrease) certain kernel parameters important to HTCondor's performance.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_off

Shutdown HTCondor daemons

Synopsis

condor_off [**-help** | **-version**]

condor_off [**-graceful** | **-fast** | **-peaceful** | **-force-graceful**] [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname*]

Description

condor_off shuts down a set of the HTCondor 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 HTCondor daemons if need be. To restart HTCondor running on a machine, see the *condor_on* command.

With the **-daemon** *master* option, *condor_off* will shut down all daemons including the *condor_master*. Specification using the **-daemon** option will shut down only the specified daemon.

For security reasons of authentication and authorization, this command requires `ADMINISTRATOR` level of access.

Options

-help Display usage information

-version Display version information

-graceful Gracefully shutdown daemons (the default)

-fast Quickly shutdown daemons. A minimum of the first two characters of this option must be specified, to distinguish it from the **-force-graceful** command.

-peaceful Wait indefinitely for jobs to finish

-force-graceful Force a graceful shutdown, even after issuing a **-peaceful** command. A minimum of the first two characters of this option must be specified, to distinguish it from the **-fast** command.

-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 *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-daemon *daemonname* 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 -daemon 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_on

Start up HTCondor daemons

Synopsis

condor_on [**-help** | **-version**]

condor_on [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname*]

Description

condor_on starts up a set of the HTCondor 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 **-daemon** *master* option will tell the *condor_master* to start up the HTCondor daemons specified in the configuration variable `DAEMON_LIST`. If a daemon other than the *condor_master* is specified with the **-daemon** option, *condor_on* starts up only that daemon.

This command cannot be used to start up the *condor_master* daemon.

For security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

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 ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-daemon *daemonname* Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

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 -daemon 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 -daemon schedd
```

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_ping

Attempt a security negotiation to determine if it succeeds

Synopsis

condor_ping [**-help** | **-version**]

condor_ping [**-debug**] [**-address** <*a.b.c.d:port*>] [**-pool** *host name*] [**-name** *daemon name*] [**-type** *subsystem*] [**-config** *filename*] [**-quiet | -table | -verbose**] *token* [*token [. . .]*]

Description

condor_ping attempts a security negotiation to discover whether the configuration is set such that the negotiation succeeds. The target of the negotiation is defined by one or a combination of the **address**, **pool**, **name**, or **type** options. If no target is specified, the default target is the *condor_schedd* daemon on the local machine.

One or more *token*s may be listed, thereby specifying one or more authorization level to impersonate in security negotiation. A token is the value `ALL`, an authorization level, a command name, or the integer value of a command. The many command names and their associated integer values will more likely be used by experts, and they are defined in the file `condor_includes/condor_commands.h`.

An authorization level may be one of the following strings. If `ALL` is listed, then negotiation is attempted for each of these possible authorization levels.

READ

WRITE

ADMINISTRATOR

SOAP

CONFIG

OWNER

DAEMON

NEGOTIATOR

ADVERTISE_MASTER

ADVERTISE_STARTD

ADVERTISE_SCHEDD

CLIENT

Options

-help Display usage information

-version Display version information

-debug Print extra debugging information as the command executes.

-config *filename* Attempt the negotiation based on the contents of the configuration file contents in file *filename*.

-address *<a.b.c.d:port>* Target the given IP address with the negotiation attempt.

-pool *hostname* Target the given *host* with the negotiation attempt. May be combined with specifications defined by **name** and **type** options.

-name *daemonname* Target the daemon given by *daemonname* with the negotiation attempt.

-type *subsystem* Target the daemon identified by *subsystem*, one of the values of the predefined `$(SUBSYSTEM)` macro.

-quiet Set exit status only; no output displayed.

-table Output is displayed with one result per line, in a table format.

-verbose Display all available output.

Examples

The example Unix command

```
condor_ping  -address "<127.0.0.1:9618>" -table READ WRITE DAEMON
```

places double quote marks around the sinful string to prevent the less than and the greater than characters from causing redirect of input and output. The given IP address is targeted with 3 attempts to negotiate: one at the `READ` authorization level, one at the `WRITE` authorization level, and one at the `DAEMON` authorization level.

Exit Status

condor_ping will exit with the status value of the negotiation it attempted, where 0 (zero) indicates success, and 1 (one) indicates failure. If multiple security negotiations were attempted, the exit status will be the logical OR of all values.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_pool_job_report

generate report about all jobs that have run in the last 24 hours on all execute hosts

Synopsis

condor_pool_job_report

Description

condor_pool_job_report is a Linux-only tool that is designed to be run nightly using *cron*. It is intended to be run on the central manager, or another machine that has administrative permissions, and is able to fetch the *condor_startd* history logs from all of the *condor_startd* daemons in the pool. After fetching these logs, *condor_pool_job_report* then generates a report about job run times and mails it to administrators, as defined by configuration variable `CONDOR_ADMIN`.

Exit Status

condor_pool_job_report will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_power

send packet intended to wake a machine from a low power state

Synopsis

condor_power [**-h**]

condor_power [**-d**] [**-i**] [**-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.

condor_power relies on the router receiving the WOL packet to correctly broadcast the request. Since routers are often configured to ignore requests to broadcast messages on a different subnet than the sender, the send of a WOL packet to a machine on a different subnet may fail.

Options

-h Print usage information and exit.

-d Enable debugging messages.

-i Read a ClassAd that is piped in through standard input.

-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 a mask for an IPv4 address. Without this option, a global broadcast will be sent.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_preen

remove extraneous files from HTCondor directories

Synopsis

condor_preen [**-mail**] [**-remove**] [**-verbose**] [**-debug**]

Description

condor_preen examines the directories belonging to HTCondor, and removes extraneous files and directories which may be left over from HTCondor 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 HTCondor 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 variables `VALID_SPOOL_FILES` and `SYSTEM_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.

-debug Print extra debugging information as the command executes.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_prio

change priority of jobs in the HTCondor queue

Synopsis

condor_prio **-p** *priority* | **+** *value* | **-** *value* [**-n** *schedd_name*] *cluster* | *cluster.process* | *username* | *-a*

condor_prio **-p** *priority* | **+** *value* | **-** *value* [**-pool** *pool_name* **-n** *schedd_name*]*cluster* | *cluster.process* | *username* | *-a*

Description

condor_prio changes the priority of one or more jobs in the HTCondor queue. If the job identification is given by *cluster.process*, *condor_prio* attempts to change the priority of the single job with job ClassAd attributes `ClusterId` and `ProcId`. If described by *cluster*, *condor_prio* attempts to change the priority of all processes with the given `ClusterId` job ClassAd attribute. If *username* is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. For *-a*, *condor_prio* attempts to change priority of all jobs in the queue.

The user must set a new priority with the **-p** option, or specify a priority adjustment. The priority of a job can be any integer, with higher numbers corresponding to greater priority. For adjustment of the current priority, **+** *value* increases the priority by the amount given with *value*. **-** *value* decreases the priority by the amount given with *value*.

Only the owner of a job or the super user can change the priority.

The priority changed by *condor_prio* is only used when comparing to the priority jobs owned by the same user and submitted from the same machine.

Options

-n ***schedd_name*** Change priority of jobs queued at the specified *condor_schedd* in the local pool.

-pool ***pool_name*** **-n** ***schedd_name*** Change priority of jobs queued at the specified *condor_schedd* in the specified pool.

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_procd

Track and manage process families

Synopsis

***condor_procd* -h**

***condor_procd* -A** *address-file* [**options**]

Description

condor_procd tracks and manages process families on behalf of the HTCondor daemons. It may track families of PIDs via relationships such as: direct parent/child, environment variables, UID, and supplementary group IDs. Management of the PID families include

- registering new families or new members of existing families
- getting usage information
- signaling families for operations such as suspension, continuing, or killing the family
- getting a snapshot of the tree of families

In a regular HTCondor installation, this program is not intended to be used or executed by any human.

The required argument, **-A** *address-file*, is the path and file name of the address file which is the named pipe that clients must use to speak with the *condor_procd*.

Options

-h Print out usage information and exit.

-D Wait for the debugger. Initially sleep 30 seconds before beginning normal function.

-C *principal* The *principal* is the UID of the owner of the named pipe that clients must use to speak to the *condor_procd*.

-L *log-file* A file the *condor_procd* will use to write logging information.

-E When specified, another tool such as the *procd_ctl* tool must allocate the GID associated with a process. When this option is *not* specified, the *condor_procd* will allocate the GID itself.

-P *PID* If not specified, the *condor_procd* will use the *condor_procd*'s parent, which may not be PID 1 on Unix, as the parent of the *condor_procd* and the root of the tracking family. When not specified, if the *condor_procd*'s parent PID dies, the *condor_procd* exits. When specified, the *condor_procd* will track this *PID* family in question and not also exit if the PID exits.

-S *seconds* The maximum number of seconds the *condor_procd* will wait between taking snapshots of the tree of families. Different clients to the *condor_procd* can specify different snapshot times. The quickest snapshot time is the one performed by the *condor_procd*. When this option is not specified, a default value of 60 seconds is used.

-G *min-gid max-gid* If the **-E** option is *not* specified, then track process families using a self-allocated, free GID out of the inclusive range specified by *min-gid* and *max-gid*. This means that if a new process shows up using a previously known GID, the new process will automatically associate into the process family assigned that GID. If the **-E** option *is* specified, then instead of self-allocating the GID, the *procd_ctl* tool must be used to associate the GID with the PID root of the family. The associated GID must still be in the range specified. This is a Linux-only feature.

-K *windows-softkill-binary* This is the path and executable name of the *condor_softkill.exe* binary. It is used to send softkill signals to process families. This is a Windows-only feature.

-I *glexec-kill-path glexec-path* Specifies, with *glexec-kill-path*, the path and executable name of a binary used to send a signal to a PID. The *glexec* binary, specified by *glexec-path*, executes the program specified with *glexec-kill-path* under the right privileges to send the signal.

General Remarks

This program may be used in a stand alone mode, independent of HTCondor, to track process families. The programs *procd_ctl* and *gidd_alloc* are used with the *condor_procd* in stand alone mode to interact with the daemon and to inquire about certain state of running processes on the machine, respectively.

Exit Status

condor_procd will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_q

Display information about jobs in queue

Synopsis

condor_q [**-help [Universe | State]**]

condor_q [**-debug**] [*general options*] [*restriction list*] [*output options*] [*analyze options*]

Description

condor_q displays information about jobs in the HTCondor job queue. By default, *condor_q* queries the local job queue, but this behavior may be modified by specifying one of the general options.

As of version 8.5.2, *condor_q* defaults to querying only the current user's jobs. This default is overridden when the restriction list has usernames and/or job ids, when the -submitter or -allusers arguments are specified, or when the current user is a queue superuser. It can also be overridden by setting the `CONDOR_Q_ONLY_MY_JOBS` configuration macro to `False`.

As of version 8.5.6, *condor_q* defaults to batch-mode output (see -batch in the Options section below). The old behavior can be obtained by specifying -nobatch on the command line. To change the default back to its pre-8.5.6 value, set the new configuration variable `CONDOR_Q_DASH_BATCH_IS_DEFAULT` to `False`.

Batches of jobs

As of version 8.5.6, *condor_q* defaults to displaying information about batches of jobs, rather than individual jobs. The intention is that this will be a more useful, and user-friendly, format for users with large numbers of jobs in the queue. Ideally, users will specify meaningful batch names for their jobs, to make it easier to keep track of related jobs.

(For information about specifying batch names for your jobs, see the *condor_submit* (11) and *condor_submit_dag* (11) man pages.)

A batch of jobs is defined as follows:

- An entire workflow (a DAG or hierarchy of nested DAGs) (note that *condor_dagman* now specifies a default batch name for all jobs in a given workflow)
- All jobs in a single cluster
- All jobs submitted by a single user that have the same executable specified in their submit file (unless submitted with different batch names)

- All jobs submitted by a single user that have the same batch name specified in their submit file or on the *condor_submit* or *condor_submit_dag* command line.

Output

There are many output options that modify the output generated by *condor_q*. The effects of these options, and the meanings of the various output data, are described below.

Output options

If the **-long** option is specified, *condor_q* displays a long description of the queried jobs by printing the entire job ClassAd for all jobs matching the restrictions, if any. Individual attributes of the job ClassAd can be displayed by means of the **-format** option, which displays attributes with a `printf(3)` format, or with the **-autoformat** option. Multiple **-format** options may be specified in the option list to display several attributes of the job.

For most output options (except as specified), the last line of *condor_q* output contains a summary of the queue: the total number of jobs, and the number of jobs in the completed, removed, idle, running, held and suspended states.

If no output options are specified, *condor_q* now defaults to batch mode, and displays the following columns of information, with one line of output per batch of jobs:

```
OWNER, BATCH_NAME, SUBMITTED, DONE, RUN, IDLE, [HOLD,] TOTAL, JOB_IDS
```

Note that the HOLD column is only shown if there are held jobs in the output or if there are *no* jobs in the output.

If the **-nobatch** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, SUBMITTED, RUN_TIME, ST, PRI, SIZE, CMD
```

If the **-dag** option is specified (in conjunction with **-nobatch**), *condor_q* displays the following columns of information, with one line of output per job; the owner is shown only for top-level jobs, and for all other jobs (including sub-DAGs) the node name is shown:

```
ID, OWNER/NODENAME, SUBMITTED, RUN_TIME, ST, PRI, SIZE, CMD
```

If the **-run** option is specified (in conjunction with **-nobatch**), *condor_q* displays the following columns of information, with one line of output per running job:

```
ID, OWNER, SUBMITTED, RUN_TIME, HOST(S)
```

Also note that the **-run** option disables output of the totals line.

If the **-grid** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, STATUS, GRID->MANAGER, HOST, GRID_JOB_ID
```

If the **-goodput** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, SUBMITTED, RUN_TIME, GOODPUT, CPU_UTIL, Mb/s
```

If the **-io** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, RUNS, ST, INPUT, OUTPUT, RATE, MISC
```

If the **-cputime** option is specified (in conjunction with **-nobatch**), *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, SUBMITTED, CPU_TIME, ST, PRI, SIZE, CMD
```

If the **-hold** option is specified, *condor_q* displays the following columns of information, with one line of output per job:

```
ID, OWNER, HELD_SINCE, HOLD_REASON
```

If the **-totals** option is specified, *condor_q* displays only one line of output no matter how many jobs and batches of jobs are in the queue. That line of output contains the total number of jobs, and the number of jobs in the completed, removed, idle, running, held and suspended states.

Output data

The available output data are as follows:

ID (Non-batch mode only) The cluster/process id of the HTCondor job.

OWNER The owner of the job or batch of jobs.

OWNER/NODENAME (**-dag** only) The owner of a job or the DAG node name of the job.

BATCH_NAME (Batch mode only) The batch name of the job or batch of jobs.

SUBMITTED The month, day, hour, and minute the job was submitted to the queue.

DONE (Batch mode only) The number of job procs that are done, but still in the queue.

RUN (Batch mode only) The number of job procs that are running.

IDLE (Batch mode only) The number of job procs that are in the queue but idle.

HOLD (Batch mode only) The number of job procs that are in the queue but held.

TOTAL (Batch mode only) The total number of job procs in the queue, unless the batch is a DAG, in which case this is the total number of clusters in the queue. Note: for non-DAG batches, the TOTAL column contains correct values only in version 8.5.7 and later.

JOB_IDS (Batch mode only) The range of job IDs belonging to the batch.

RUN_TIME (Non-batch mode only) Wall-clock time accumulated by the job to date in days, hours, minutes, and seconds.

ST (Non-batch mode only) Current status of the job, which varies somewhat according to the job universe and the timing of updates. H = on hold, R = running, I = idle (waiting for a machine to execute on), C = completed, X = removed, S = suspended (execution of a running job temporarily suspended on execute node), < = transferring input (or queued to do so), and > = transferring output (or queued to do so).

PRI (Non-batch mode only) User specified priority of the job, displayed as an integer, with higher numbers corresponding to better priority.

SIZE (Non-batch mode only) The peak amount of memory in Mbytes consumed by the job; note this value is only refreshed periodically. The actual value reported is taken from the job ClassAd attribute `MemoryUsage` if this attribute is defined, and from job attribute `ImageSize` otherwise.

CMD (Non-batch mode only) The name of the executable.

HOST(S) (**-run** only) The host where the job is running.

STATUS (**-grid** only) The state that HTCondor 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

GRID->MANAGER (**-grid** only) A guess at what remote batch system is running the job. It is a guess, because HTCondor 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 (**-grid** only) The host to which the job was submitted.

GRID_JOB_ID (**-grid** only) (More information needed here.)

GOODPUT (**-goodput** only) 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 (**-goodput** only) 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 (**-goodput** only) The network usage of this job, in Megabits per second of run-time.

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.

INPUT (**-io** only) For standard universe, FileReadBytes; otherwise, BytesRecvd.

OUTPUT (**-io** only) For standard universe, FileWriteBytes; otherwise, BytesSent.

RATE (**-io** only) For standard universe, FileReadBytes+FileWriteBytes; otherwise, BytesRecvd+BytesSent.

MISC (**-io** only) JobUniverse.

CPU_TIME (**-cputime** only) 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.)

HELD_SINCE (**-hold** only) Month, day, hour and minute at which the job was held.

HOLD_REASON (**-hold** only) The hold reason for the job.

Analyze

The **-analyze** or **-better-analyze** options can 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 can vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_REQUIREMENTS` expression. If the analyze option **-verbose** is specified along with the **-analyze** option, the reason for failure is displayed on a per machine basis. **-better-analyze** differs from **-analyze** in that it will do matchmaking analysis on jobs even if they are currently running, or if the reason they are not running is not due to matchmaking. **-better-analyze** also produces more thorough analysis of complex Requirements and shows the values of relevant job ClassAd attributes. When only a single machine is being analyzed via **-machine** or **-mconstraint**, the values of relevant attributes of the machine ClassAd are also displayed.

Restrictions

To restrict the display to jobs of interest, a list of zero or more restriction options may be supplied. Each restriction may be one of:

- ***cluster.process***, which matches jobs which belong to the specified cluster and have the specified process number;
- ***cluster*** (without a *process*), which matches all jobs belonging to the specified cluster;
- ***owner***, which matches all jobs owned by the specified owner;
- **-constraint *expression***, which matches all jobs that satisfy the specified ClassAd expression;
- **-allusers**, which overrides the default restriction of only matching jobs submitted by the current user.

If *cluster* or *cluster.process* is specified, and the job matching that restriction is a *condor_dagman* job, information for all jobs of that DAG is displayed in batch mode (in non-batch mode, only the *condor_dagman* job itself is displayed).

If no *owner* restrictions are present, 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.

Options

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-batch (output option) Show a single line of progress information for a batch of jobs, where a batch is defined as follows:

- An entire workflow (a DAG or hierarchy of nested DAGs)
- All jobs in a single cluster
- All jobs submitted by a single user that have the same executable specified in their submit file
- All jobs submitted by a single user that have the same batch name specified in their submit file or on the *condor_submit* or *condor_submit_dag* command line.

Also change the output columns as noted above.

Note that, as of version 8.5.6, **-batch** is the default, unless the `CONDOR_Q_DASH_BATCH_IS_DEFAULT` configuration variable is set to `False`.

-nobatch (output option) Show a line for each job (turn off the **-batch** option).

-global (general option) Queries all job queues in the pool.

-submitter *submitter* (general option) List jobs of a specific submitter in the entire pool, not just for a single *condor_schedd*.

-name *name* (general option) Query only the job queue of the named *condor_schedd* daemon.

-pool *centralmanagerhostname[:portnumber]* (general option) Use the *centralmanagerhostname* as the central manager to locate *condor_schedd* daemons. The default is the `COLLECTOR_HOST`, as specified in the configuration.

-jobads *file* (general option) 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* **-long** is used with the header stripped out.

-userlog *file* (general option) Display jobs, with job information coming from a job event log, instead of from the real ClassAds from the *condor_schedd* daemon. This is most useful for automated testing of the status of jobs known to be in the given job event log, because it reduces the load on the *condor_schedd*. A job event log does not contain all of the job information, so some fields in the normal output of *condor_q* will be blank.

-autocluster (output option) Output *condor_schedd* daemon auto cluster information. For each auto cluster, output the unique ID of the auto cluster along with the number of jobs in that auto cluster. This option is intended to be used together with the **-long** option to output the ClassAds representing auto clusters. The ClassAds can then be used to identify or classify the demand for sets of machine resources, which will be useful in the on-demand creation of execute nodes for glidein services.

-cputime (output option) 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. Note that this option has no effect unless used in conjunction with **-nobatch**.

-currentrun (output option) 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.

-dag (output option) Display DAG node jobs under their DAGMan instance. Child nodes are listed using indentation to show the structure of the DAG. Note that this option has no effect unless used in conjunction with **-nobatch**.

-expert (output option) Display shorter error messages.

-grid (output option) Get information only about jobs submitted to grid resources described as **gt2** or **gt5**.

-goodput (output option) Display job goodput statistics.

-help [Universe | State] (output option) Print usage info, and, optionally, additionally print job universes or job states.

-hold (output option) 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.

-limit *Number* (output option) Limit the number of items output to *Number*.

-io (output option) Display job input/output summaries.

-long (output option) Display entire job ClassAds in long format (one attribute per line).

-run (output option) Get information about running jobs. Note that this option has no effect unless used in conjunction with **-nobatch**.

-stream-results (output option) Display results as jobs are fetched from the job queue rather than storing results in memory until all jobs have been fetched. This can reduce memory consumption when fetching large numbers of jobs, but if *condor_q* is paused while displaying results, this could result in a timeout in communication with *condor_schedd*.

-totals (output option) Display only the totals.

-version (output option) Print the HTCondor version and exit.

-wide (output option) If this option is specified, and the command portion of the output would cause the output to extend beyond 80 columns, display beyond the 80 columns.

-xml (output option) Display entire job ClassAds in XML format. The XML format is fully defined in the reference manual, obtained from the ClassAds web page, with a link at http://htcondor.org/classad/classad.html.

-json (output option) Display entire job ClassAds in JSON format.

-attributes *Attr1[,Attr2 . . .]* (output option) Explicitly list the attributes, by name in a comma separated list, which should be displayed when using the **-xml**, **-json** or **-long** options. Limiting the number of attributes increases the efficiency of the query.

-format ***fmt attr*** (output 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 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. %r prints the unevaluated, or raw values. 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`. %v identifies the type of the attribute, and then prints the value in an appropriate format. %V identifies the type of the attribute, and then prints the value in an appropriate format as it would appear in the **-long** format. As an example, strings used with %V will have quote marks. 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.

-autoformat[:jlhVr,tng] ***attr1 [attr2 ...]*** **or -af[:jlhVr,tng]** ***attr1 [attr2 ...]*** (output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings. This output option does *not* work in conjunction with any of the options **-run**, **-currentrun**, **-hold**, **-grid**, **-goodput**, or **-io**.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

j print the job ID as the first field,

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may *not* be used together. The **l** and **h** characters may *not* be used together.

-analyze[:<qual>] (analyze option) Perform a matchmaking analysis on why the requested jobs are not running. First a simple analysis determines if the job is not running due to not being in a runnable state. If the job is in a runnable state, then this option is equivalent to **-better-analyze**. **<qual>** is a comma separated list containing one or more of

priority to consider user priority during the analysis

summary to show a one line summary for each job or machine

reverse to analyze machines, rather than jobs

-better-analyze[:<qual>] (analyze option) Perform a more detailed matchmaking analysis to determine how many resources are available to run the requested jobs. This option is never meaningful for Scheduler universe jobs and only meaningful for grid universe jobs doing matchmaking. **<qual>** is a comma separated list containing one or more of

priority to consider user priority during the analysis

summary to show a one line summary for each job or machine

reverse to analyze machines, rather than jobs

-machine *name* (analyze option) When doing matchmaking analysis, analyze only machine ClassAds that have slot or machine names that match the given name.

-mconstraint *expression* (analyze option) When doing matchmaking analysis, match only machine ClassAds which match the ClassAd expression constraint.

-slotads *file* (analyze option) When doing matchmaking analysis, use the machine ClassAds 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* **-long** is used.

-userprios *file* (analyze option) When doing matchmaking analysis with priority, read user priorities from the file rather than the ones from the *condor_negotiator* daemon. This is most useful for debugging purposes or to speed up analysis in situations where the *condor_negotiator* daemon is slow to respond to *condor_userprio* requests. The file should be in the format produced by *condor_userprio* **-long**.

-nouserprios (analyze option) Do not consider user priority during the analysis.

-reverse-analyze (analyze option) Analyze machine requirements against jobs.

-verbose (analyze option) When doing analysis, show progress and include the names of specific machines in the output.

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 HTCondor 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.

Options **-goodput**, **-cputime**, and **-io** are most useful for standard universe jobs, since they rely on values computed when a job produces a checkpoint.

It is possible to to hold jobs that are in the X state. To avoid this it is best to construct a **-constraint** *expression* that option contains `JobStatus != 3` if the user wishes to avoid this condition.

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 analysis in summary format:

```
$ condor_q -analyze:summary

-- Submitter: submit-1.chtc.wisc.edu : <192.168.100.43:9618?sock=11794_95bb_3> :
 submit-1.chtc.wisc.edu
Analyzing matches for 5979 slots
            Autocluster  Matches    Machine     Running  Serving
 JobId     Members/Idle  Reqmnts  Rejects Job  Users Job Other User Avail Owner
---------- ------------ -------- ------------ ---------- ---------- ----- -----
25764522.0  7/0             5910        820   7/10       5046        34   smith
25764682.0  9/0             2172        603   9/9        1531        29   smith
```

```
25765082.0  18/0            2172        603   18/9       1531        29   smith
25765900.0  1/0             2172        603   1/9        1531        29   smith
```

An example that shows summary information by machine:

```
$ condor_q -ana:sum,rev

-- Submitter: s-1.chtc.wisc.edu : <192.168.100.43:9618?sock=11794_95bb_3> : s-1.chtc.wisc.edu
Analyzing matches for 2885 jobs
                              Slot  Slot's Req    Job's Req     Both
Name                          Type  Matches Job  Matches Slot    Match %
------------------------      ---- ------------  ------------ ----------
slot1@INFO.wisc.edu           Stat         2729  0                 0.00
slot2@INFO.wisc.edu           Stat         2729  0                 0.00
slot1@aci-001.chtc.wisc.edu   Part            0  2793              0.00
slot1_1@a-001.chtc.wisc.edu   Dyn          2644  2792             91.37
slot1_2@a-001.chtc.wisc.edu   Dyn          2623  2601             85.10
slot1_3@a-001.chtc.wisc.edu   Dyn          2644  2632             85.82
slot1_4@a-001.chtc.wisc.edu   Dyn          2644  2792             91.37
slot1@a-002.chtc.wisc.edu     Part            0  2633              0.00
slot1_10@a-002.chtc.wisc.edu  Den          2623  2601             85.10
```

An example with two independent DAGs in the queue:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:35169?...
OWNER  BATCH_NAME    SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
wenger DAG: 3696    2/12 11:55      _     10       _     10 3698.0 ... 3707.0
wenger DAG: 3697    2/12 11:55      1      1       1     10 3709.0 ... 3710.0

14 jobs; 0 completed, 0 removed, 1 idle, 13 running, 0 held, 0 suspended
```

Note that the "13 running" in the last line is two more than the total of the RUN column, because the two *condor_dagman* jobs themselves are counted in the last line but not the RUN column.

Also note that the "completed" value in the last line does not correspond to the total of the DONE column, because the "completed" value in the last line only counts jobs that are completed but still in the queue, whereas the DONE column counts jobs that are no longer in the queue.

Here's an example with a held job, illustrating the addition of the HOLD column to the output:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
OWNER  BATCH_NAME        SUBMITTED   DONE   RUN    IDLE   HOLD  TOTAL JOB_IDS
wenger CMD: /bin/slee   9/13 16:25      _      3       _      1      4 599.0 ...

4 jobs; 0 completed, 0 removed, 0 idle, 3 running, 1 held, 0 suspended
```

Here are some examples with a nested-DAG workflow in the queue, which is one of the most complicated cases. The workflow consists of a top-level DAG with nodes NodeA and NodeB, each with two two-proc clusters; and a sub-DAG SubZ with nodes NodeSA and NodeSB, each with two two-proc clusters.

First of all, non-batch mode with all of the node jobs in the queue:

```
$ condor_q -nobatch

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger          9/13 16:05   0+00:00:13 R  0    2.4 condor_dagman -p 0
 592.0   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 60
 592.1   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 300
 593.0   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 60
 593.1   wenger          9/13 16:05   0+00:00:07 R  0    0.0 sleep 300
 594.0   wenger          9/13 16:05   0+00:00:07 R  0    2.4 condor_dagman -p 0
 595.0   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 60
 595.1   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 300
 596.0   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 60
 596.1   wenger          9/13 16:05   0+00:00:01 R  0    0.0 sleep 300

10 jobs; 0 completed, 0 removed, 0 idle, 10 running, 0 held, 0 suspended
```

Now non-batch mode with the **-dag** option (unfortunately, *condor_q* doesn't do a good job of grouping procs in the same cluster together):

```
$ condor_q -nobatch -dag

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER/NODENAME      SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger             9/13 16:05   0+00:00:27 R  0    2.4 condor_dagman -
 592.0    |-NodeA           9/13 16:05   0+00:00:21 R  0    0.0 sleep 60
 593.0    |-NodeB           9/13 16:05   0+00:00:21 R  0    0.0 sleep 60
 594.0    |-SubZ            9/13 16:05   0+00:00:21 R  0    2.4 condor_dagman -
 595.0     |-NodeSA         9/13 16:05   0+00:00:15 R  0    0.0 sleep 60
 596.0     |-NodeSB         9/13 16:05   0+00:00:15 R  0    0.0 sleep 60
 592.1    |-NodeA           9/13 16:05   0+00:00:21 R  0    0.0 sleep 300
 593.1    |-NodeB           9/13 16:05   0+00:00:21 R  0    0.0 sleep 300
 595.1     |-NodeSA         9/13 16:05   0+00:00:15 R  0    0.0 sleep 300
 596.1     |-NodeSB         9/13 16:05   0+00:00:15 R  0    0.0 sleep 300

10 jobs; 0 completed, 0 removed, 0 idle, 10 running, 0 held, 0 suspended
```

Now, finally, the non-batch (default) mode:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
OWNER  BATCH_NAME     SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
wenger ex1.dag+591   9/13 16:05      _      8      _      5 592.0 ... 596.1

10 jobs; 0 completed, 0 removed, 0 idle, 10 running, 0 held, 0 suspended
```

There are several things about this output that may be slightly confusing:

- The TOTAL column is less than the RUN column. This is because, for DAG node jobs, their contribution to the TOTAL column is the number of clusters, not the number of procs (but their contribution to the RUN column is the number of procs). So the four DAG nodes (8 procs) contribute 4, and the sub-DAG contributes 1, to the TOTAL column. (But, somewhat confusingly, the sub-DAG job is *not* counted in the RUN column.)
- The sum of the RUN and IDLE columns (8) is less than the 10 jobs listed in the totals line at the bottom. This is because the top-level DAG and sub-DAG jobs are not counted in the RUN column, but they are counted in the totals line.

Now here is non-batch mode after proc 0 of each node job has finished:

```
$ condor_q -nobatch

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger          9/13 16:05   0+00:01:19 R  0    2.4 condor_dagman -p 0
 592.1   wenger          9/13 16:05   0+00:01:13 R  0    0.0 sleep 300
 593.1   wenger          9/13 16:05   0+00:01:13 R  0    0.0 sleep 300
 594.0   wenger          9/13 16:05   0+00:01:13 R  0    2.4 condor_dagman -p 0
 595.1   wenger          9/13 16:05   0+00:01:07 R  0    0.0 sleep 300
 596.1   wenger          9/13 16:05   0+00:01:07 R  0    0.0 sleep 300

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

The same state also with the **-dag** option:

```
$ condor_q -nobatch -dag

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
 ID      OWNER/NODENAME      SUBMITTED     RUN_TIME ST PRI SIZE CMD
 591.0   wenger            9/13 16:05   0+00:01:30 R  0    2.4 condor_dagman -
 592.1    |-NodeA           9/13 16:05   0+00:01:24 R  0    0.0 sleep 300
 593.1    |-NodeB           9/13 16:05   0+00:01:24 R  0    0.0 sleep 300
 594.0    |-SubZ            9/13 16:05   0+00:01:24 R  0    2.4 condor_dagman -
 595.1     |-NodeSA         9/13 16:05   0+00:01:18 R  0    0.0 sleep 300
 596.1     |-NodeSB         9/13 16:05   0+00:01:18 R  0    0.0 sleep 300

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

And, finally, that state in batch (default) mode:

```
$ condor_q

-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9619?...
OWNER  BATCH_NAME     SUBMITTED   DONE   RUN    IDLE  TOTAL JOB_IDS
wenger ex1.dag+591   9/13 16:05      _      4       _      5 592.1 ... 596.1

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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 HTCondor 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. Multiple attribute value pairs may be listed on the same command line.

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`.

condor_qedit validates both attribute names and attribute values, checking for correct ClassAd syntax. An error message is printed, and no attribute is set or changed if the name or value is invalid.

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") && (D
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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_qsub

Queue jobs that use PBS/SGE-style submission

Synopsis

condor_qsub [−−**version**]

condor_qsub [**Specific options**] [**Directory options**] [**Environmental options**] [**File options**] [**Notification options**] [**Resource options**] [**Status options**] [**Submission options**] *commandfile*

Description

condor_qsub submits an HTCondor job. This job is specified in a PBS/Torque style or an SGE style. *condor_qsub* permits the submission of dependent jobs without the need to specify the full dependency graph at submission time. Doing things this way is neither as efficient as HTCondor's DAGMan, nor as functional as SGE's *qsub* or *qalter*. *condor_qsub* serves as a minimal translator to be able to use software originally written to interact with PBS, Torque, and SGE in an HTCondor pool.

condor_qsub attempts to behave like *qsub*. Less than half of the *qsub* functionality is implemented. Option descriptions describe the differences between the behavior of *qsub* and *condor_qsub*. *qsub* options not listed here are not supported. Some concepts present in PBS and SGE do not apply to HTCondor, and so these options are not implemented.

For a full listing of *qsub* options, please see

POSIX : http://pubs.opengroup.org/onlinepubs/9699919799/utilities/qsub.html

SGE : http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html

PBS/Torque : http://docs.adaptivecomputing.com/torque/4-1-3/Content/topics/commands/qsub.htm

condor_qsub accepts either command line options or the single file, *commandfile*, that contains all of the commands.

condor_qsub does the opposite of job submission within the **grid** universe **batch** grid type, which takes HTCondor jobs submitted with HTCondor syntax and submits them to PBS, SGE, or LSF.

Options

-a *date_time* (Submission option) Specify a deferred execution date and time. The PBS/Torque syntax of *date_time* is a string in the form *[[[[CC]YY]MM]DD]hhmm[.SS]*. The portions of this string which are optional are *CC*, *YY*, *MM*, *DD*, and *SS*. For SGE, *MM* and *DD* are *not* optional. For PBS, *MM* and *DD* are optional. *condor_qsub* follows the PBS style.

-A *account_string* (Status option) Uses group accounting where the string *account_string* is the accounting group associated with this job. Unlike SGE, there is no default group of `"sge"`.

-b *y|n* (Submission option) Using the SGE definition of its *-b* option, a value of *y* causes *condor_qsub* to *not* parse the file for additional *condor_qsub* commands. The default value is *n*. If the command line argument **-f** *filename* is also specified, it negates a value of *y*.

-c *checkpoint_option* (Submission option) For standard universe jobs only, controls the how HTCondor produces checkpoints. *checkpoint_options* may be one of

n or N Do not produce checkpoints.

s or S Do not produce periodic checkpoints. A job will only produce a checkpoint when the job is evicted.

More options may be implemented in the future.

—condor-keep-files (Specific option) Directs HTCondor to *not* remove temporary files generated by *condor_qsub*, such as HTCondor submit files and sentinel jobs. These temporary files may be important for debugging.

-cwd (Directory option) Specifies the initial directory in which the job will run to be the current directory from which the job was submitted. This sets **initialdir** for *condor_submit*.

-d *path* or -wd *path* (Directory option) Specifies the initial directory in which the job will run to be *path*. This sets **initialdir** for *condor_submit*.

-e *filename* (File option) Specifies the *condor_submit* command **error**, the file where `stderr` is written. If not specified, set to the default name of `<commandfile>.e<ClusterId>`, where `<commandfile>` is the *condor_qsub* argument, and `<ClusterId>` is the job attribute `ClusterId` assigned for the job.

—f *qsub_file* (Specific option) Parse *qsub_file* to search for and set additional *condor_submit* commands. Within the file, commands will appear as `#PBS` or `#SGE`. *condor_qsub* will parse the batch file listed as *qsub_file*.

-h (Status option) Placed submitted job directly into the hold state.

—help (Specific option) Print usage information and exit.

-hold_jid *<jid>* (Status option) Submits a job in the hold state. This job is released only when a previously submitted job, identified by its cluster ID as *<jid>*, exits successfully. Successful completion is defined as not exiting with exit code 100. In implementation, there are three jobs that define this SGE feature. The first job is the previously submitted job. The second job is the newly submitted one that is waiting for the first to finish successfully. The third job is what SGE calls a *sentinel* job; this is an HTCondor local

universe job that watches the history for the first job's exit code. This third job will exit once it has seen the exit code and, for a successful termination of the first job, run *condor_release* on the second job. If the first job is an array job, the second job will only be released after all individual jobs of the first job have completed.

-i ***[hostname:]filename*** (File option) Specifies the *condor_submit* command **input**, the file from which `stdin` is read.

-j ***characters*** (File option) Acceptable characters for this option are `e`, `o`, and `n`. The only sequence that is relevant is `eo`; it specifies that both standard output and standard error are to be sent to the same file. The file will be the one specified by the **-o** option, if both the **-o** and **-e** options exist. The file will be the one specified by the **-e** option, if only the **-e** option is provided. If neither the **-o** nor the **-e** options are provided, the file will be the default used for the **-o** option.

-l ***resource_spec*** (Resource option) Specifies requirements for the job, such as the amount of RAM and the number of CPUs. Only PBS-style resource requests are supported. *resource_spec* is a comma separated list of key/value pairs. Each pair is of the form `resource_name=value`. `resource_name` and `value` may be

`resource_name`	`value`	Description
arch	string	Sets `Arch` machine attribute. Enclose in double quotes.
file	size	Disk space requested.
host	string	Host machine on which the job must run.
mem	size	Amount of memory requested.
nodes	`{<node_count> | <hostname>} [:ppn=<ppn>] [:gpus=<gpu>] [:<property> [:<property>] ...] [+ ...]`	Number and/or properties of nodes to be used. For examples, please see http://docs.adaptivecomputing.com/torque/4-1-3/Content/topics/2-jobs/requestingRes.htm#qsub A size value is an
opsys	string	Sets `OpSys` machine attribute. Enclose in double quotes.
procs	integer	Number of CPUs requested.

integer specified in bytes, following the PBS/Torque default. Append `Kb`, `Mb`, `Gb`, or `Tb` to specify the value in powers of two quantities greater than bytes.

-m ***a|e|n*** (Notification option) Identify when HTCondor sends notification e-mail. If *a*, send e-mail when the job terminates abnormally. If *e*, send e-mail when the job terminates. If *n*, never send e-mail.

-M ***e-mail_address*** (Notification option) Sets the destination address for HTCondor e-mail.

-o ***filename*** (File option) Specifies the *condor_submit* command **output**, the file where `stdout` is written. If not specified, set to the default name of `<commandfile>.o<ClusterId>`, where `<commandfile>` is the

condor_qsub argument, and `<ClusterId>` is the job attribute `ClusterId` assigned for the job.

-p *integer* (Status option) Sets the **priority** submit command for the job, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority.

—print (Specific option) Send to `stdout` the contents of the HTCondor submit description file that *condor_qsub* generates.

-r *y|n* (Status option) The default value of *y* implements the default HTCondor policy of assuming that jobs that do not complete are placed back in the queue to be run again. When *n*, job submission is restricted to only running the job if the job ClassAd attribute `NumJobStarts` is currently 0. This identifies the job as not re-runnable, limiting it to start once.

-S *shell* (Submission option) Specifies the path and executable name of a shell. Alters the HTCondor submit description file produced, such that the executable becomes a wrapper script. Within the submit description file will be `executable = <shell>` and `arguments = <commandfile>`.

-t *start [-stop:step]* (Submission option) Queues a set of nearly identical jobs. The SGE-style syntax is supported. *start*, *stop*, and *step* are all integers. *start* is the starting index of the jobs, *stop* is the ending index (inclusive) of the jobs, and *step* is the step size through the indices. Note that using more than one processor or node in a job will not work with this option.

—test (Specific option) With the intention of testing a potential job submission, parse files and commands to generate error output. Produces, but then removes the HTCondor submit description file. Never submits the job, even if no errors are encountered.

-v *variable list* (Environmental option) Used to set the submit command **environment** for the job. *variable list* is as that defined for the submit command. Note that the syntax needed is specialized to deal with quote marks and white space characters.

-V (Environmental option) Sets `getenv = True` in the submit description file.

-W *attr_name=attr_value[,attr_name=attr_value...]* (File option) PBS/Torque supports a number of attributes. However, *condor_qsub* only supports the names *stagein* and *stageout* for *attr_name*. The format of *attr_value* for *stagein* and *stageout* is `local_file@hostname:remote_file[,...]` and we strip it to `remote_file[,...]`. HTCondor's file transfer mechanism is then used if needed.

—version (Specific option) Print version information for the *condor_qsub* program and exit. Note that *condor_qsub* has its own version numbers which are separate from those of HTCondor.

Exit Status

condor_qsub will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure to submit a job.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_reconfig

Reconfigure HTCondor daemons

Synopsis

condor_reconfig [**-help** | **-version**]

condor_reconfig [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname*]

Description

condor_reconfig reconfigures all of the HTCondor daemons in accordance with the current status of the HTCondor 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. Other configuration variables that can only be changed if the HTCondor daemons are restarted are listed in the HTCondor manual in the section on configuration. 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 **-daemon** *master* option will cause the reconfiguration of the *condor_master* daemon and all the child processes of the *condor_master*.

For security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

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 ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-daemon *daemonname* 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 -daemon 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_release

release held jobs in the HTCondor 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 HTCondor 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

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_reschedule

Update scheduling information to the central manager

Synopsis

condor_reschedule [**-help** | **-version**]

condor_reschedule [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-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

-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 *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_restart

Restart a set of HTCondor daemons

Synopsis

condor_restart [**-help** | **-version**]

condor_restart [**-debug**] [**-graceful** | **-fast** | **-peaceful**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**] [**-daemon** *daemonname*]

Description

condor_restart restarts a set of HTCondor daemons on a set of machines. The daemons will be put into a consistent state, killed, and then invoked 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 **-daemon** *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 reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

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`.

-graceful Gracefully shutdown daemons (the default) before restarting them

-fast Quickly shutdown daemons before restarting them

-peaceful Wait indefinitely for jobs to finish before shutting down daemons, prior to restarting them

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint ***expression*** Apply this command only to machines matching the given ClassAd *expression*

-all Send the command to all machines in the pool

-daemon ***daemonname*** Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

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 -daemon 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_rm

remove jobs from the HTCondor 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 HTCondor 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 HTCondor 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 have not yet been stopped or removed.

Examples

For a user to remove all their jobs that are not currently running:

```
% condor_rm -constraint 'JobStatus =!= 2'
```

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_rmdir

Windows-only no-fail deletion of directories

Synopsis

condor_rmdir [**/HELP** | **/?**]

condor_rmdir @*filename*

condor_rmdir [**/VERBOSE**] [**/DIAGNOSTIC**] [**/PATH:<path>**] [**/S**] [**/C**] [**/Q**] [**/NODEL**] *directory*

Description

condor_rmdir can delete a specified *directory*, and will not fail if the directory contains files that have ACLs that deny the SYSTEM process delete access, unlike the built-in Windows *rmdir* command.

The directory to be removed together with other command line arguments may be specified within a file named *filename*, prefixing this argument with an `@` character.

The *condor_rmdir.exe* executable is is intended to be used by HTCondor with the **/S /C** options, which cause it to recurse into subdirectories and continue on errors.

Options

/HELP Print usage information.

/? Print usage information.

/VERBOSE Print detailed output.

/DIAGNOSTIC Print out the internal flow of control information.

/PATH:<path> Remove the directory given by **<path>**.

/S Include subdirectories in those removed.

/C Continue even if access is denied.

/Q Print error output only.

/NODEL Do not remove directories. ACLs may still be changed.

Exit Status

condor_rmdir will exit with a status value of 0 (zero) upon success, and it will exit with the standard HRESULT error code upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_router_q

Display information about routed jobs in the queue

Synopsis

condor_router_q [**-S**] [**-R**] [**-I**] [**-H**] [**-route** *name*] [**-idle**] [**-held**] [**-constraint** *X*] [**condor_q options**]

Description

condor_router_q displays information about jobs managed by the *condor_job_router* that are in the HTCondor job queue. The functionality of this tool is that of *condor_q*, with additional options specialized for routed jobs. Therefore, any of the options for *condor_q* may also be used with *condor_router_q*.

Options

-S Summarize the state of the jobs on each route.

-R Summarize the running jobs on each route.

-I Summarize the idle jobs on each route.

-H Summarize the held jobs on each route.

-route *name* Display only the jobs on the route identified by *name*.

-idle Display only the idle jobs.

-held Display only the held jobs.

-constraint *X* Display only the jobs matching constraint *X*.

Exit Status

condor_router_q will exit with a status of 0 (zero) upon success, and non-zero otherwise.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_router_rm

Remove jobs being managed by the HTCondor Job Router

Synopsis

condor_router_rm [*router_rm options*] [*condor_rm options*]

Description

condor_router_rm is a script that provides additional features above those offered by *condor_rm*, for removing jobs being managed by the HTCondor Job Router.

The options that may be supplied to *condor_router_rm* belong to two groups:

- **router_rm options** provide the additional features
- **condor_rm options** are those options already offered by *condor_rm*. See the *condor_rm* manual page for specification of these options.

Options

-constraint ***X*** (router_rm option) Remove jobs matching the constraint specified by *X*

-held (router_rm option) Remove only jobs in the hold state

-idle (router_rm option) Remove only idle jobs

-route ***name*** (router_rm option) Remove only jobs on specified route

Exit Status

condor_router_rm will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_run

Submit a shell command-line as an HTCondor job

Synopsis

condor_run [**-u** *universe*] [**-a** *submitcmd*] *"shell command"*

Description

condor_run bundles a shell command line into an HTCondor job and submits the job. The *condor_run* command waits for the HTCondor job to complete, writes the job's output to the terminal, and exits with the exit status of the HTCondor 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 HTCondor 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 HTCondor 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 HTCondor job.

condor_run removes the HTCondor job from the queue and deletes its temporary files, if *condor_run* is killed before the HTCondor 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.

-a *submitcmd* Add the specified submit command to the implied submit description file for the job. To include spaces within *submitcmd*, enclose the submit command in double quote marks. And, to include double quote marks within *submitcmd*, enclose the submit command in single quote marks.

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 HTCondor job's log file; it is monitored by *condor_run*, to determine when the job exits.

.condor_out.<pid> The output of the HTCondor job before it is output to the terminal.

.condor_error.<pid> Any error messages for the HTCondor 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 HTCondor job may remain in the queue.

General Remarks

condor_run is intended for submitting simple shell command lines to HTCondor. 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. HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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**] [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-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 security reasons of authentication and authorization, this command requires ADMINISTRATOR level of access.

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 ***hostname*** Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_ssh_to_job

create an ssh session to a running job

Synopsis

condor_ssh_to_job [**-help**]

condor_ssh_to_job [**-debug**] [**-name** *schedd-name*] [**-pool** *pool-name*] [**-ssh** *ssh-command*] [**-keygen-options** *ssh-keygen-options*] [**-shells** *shell1,shell2,...*] [**-auto-retry**] [**-remove-on-interrupt**] *cluster* | *cluster.process* | *cluster.process.node* [*remote-command*]

Description

condor_ssh_to_job creates an *ssh* session to a running job. The job is specified with the argument. If only the job *cluster* id is given, then the job *process* id defaults to the value 0.

condor_ssh_to_job is available in Unix HTCondor distributions, and works with two kinds of jobs: those in the vanilla, vm, java, local, or parallel universes, and those jobs in the grid universe which use EC2 resources. It will not work with other grid universe jobs.

For jobs in the vanilla, vm, java, local, or parallel universes, the user must be the owner of the job or must be a queue super user, and both the *condor_schedd* and *condor_starter* daemons must allow *condor_ssh_to_job* access. If no *remote-command* is specified, an interactive shell is created. An alternate *ssh* program such as *sftp* may be specified, using the **-ssh** option, for uploading and downloading files.

The remote command or shell runs with the same user id as the running job, and it is initialized with the same working directory. The environment is initialized to be the same as that of the job, plus any changes made by the shell setup scripts and any environment variables passed by the *ssh* client. In addition, the environment variable `_CONDOR_JOB_PIDS` is defined. It is a space-separated list of PIDs associated with the job. At a minimum, the list will contain the PID of the process started when the job was launched, and it will be the first item in the list. It may contain additional PIDs of other processes that the job has created.

The *ssh* session and all processes it creates are treated by HTCondor as though they are processes belonging to the job. If the slot is preempted or suspended, the *ssh* session is killed or suspended along with the job. If the job exits before the *ssh* session finishes, the slot remains in the Claimed Busy state and is treated as though not all job processes have exited until all *ssh* sessions are closed. Multiple *ssh* sessions may be created to the same job at the same time. Resource consumption of the *sshd* process and all processes spawned by it are monitored by the *condor_starter* as though these processes belong to the job, so any policies such as `PREEMPT` that enforce a limit on resource consumption also take into account resources consumed by the *ssh* session.

condor_ssh_to_job stores ssh keys in temporary files within a newly created and uniquely named directory. The newly created directory will be within the directory defined by the environment variable `TMPDIR`. When the ssh session is finished, this directory and the ssh keys contained within it are removed.

See the HTCondor administrator's manual section on configuration for details of the configuration variables related to *condor_ssh_to_job*.

An *ssh* session works by first authenticating and authorizing a secure connection between *condor_ssh_to_job* and the *condor_starter* daemon, using HTCondor protocols. The *condor_starter* generates an ssh key pair and sends it securely to *condor_ssh_to_job*. Then the *condor_starter* spawns *sshd* in inetd mode with its stdin and stdout attached to the TCP connection from *condor_ssh_to_job*. *condor_ssh_to_job* acts as a proxy for the *ssh* client to communicate with *sshd*, using the existing connection authorized by HTCondor. *At no point is sshd listening on the network for connections or running with any privileges other than that of the user identity running the job.* If CCB is being used to enable connectivity to the execute node from outside of a firewall or private network, *condor_ssh_to_job* is able to make use of CCB in order to form the *ssh* connection.

The login shell of the user id running the job is used to run the requested command, *sshd* subsystem, or interactive shell. This is hard-coded behavior in *OpenSSH* and cannot be overridden by configuration. This means that *condor_ssh_to_job* access is effectively disabled if the login shell disables access, as in the example programs */bin/true* and */sbin/nologin*.

condor_ssh_to_job is intended to work with *OpenSSH* as installed in typical environments. It does not work on Windows platforms. If the *ssh* programs are installed in non-standard locations, then the paths to these programs will need to be customized within the HTCondor configuration. Versions of *ssh* other than *OpenSSH* may work, but they will likely require additional configuration of command-line arguments, changes to the *sshd* configuration template file, and possibly modification of the `$(LIBEXEC)/condor_ssh_to_job_sshd_setup` script used by the *condor_starter* to set up *sshd*.

For jobs in the grid universe which use EC2 resources, a request that HTCondor have the EC2 service create a new key pair for the job by specifying **ec2_keypair_file** causes *condor_ssh_to_job* to attempt to connect to the corresponding instance via *ssh*. This attempts invokes *ssh* directly, bypassing the HTCondor networking layer. It supplies *ssh* with the public DNS name of the instance and the name of the file with the new key pair's private key. For the connection to succeed, the instance must have started an *ssh* server, and its security group(s) must allow connections on port 22. Conventionally, images will allow logins using the key pair on a single specific account. Because *ssh* defaults to logging in as the current user, the **-l <username>** option or its equivalent for other versions of *ssh* will be needed as part of the *remote-command* argument. Although the **-X** option does not apply to EC2 jobs, adding **-X** or **-Y** to the *remote-command* argument can duplicate the effect.

Options

-help Display brief usage information and exit.

-debug Causes debugging information to be sent to `stderr`, based on the value of the configuration variable `TOOL_DEBUG`.

-name *schedd-name* Specify an alternate *condor_schedd*, if the default (local) one is not desired.

-pool *pool-name* Specify an alternate HTCondor pool, if the default one is not desired. Does not apply to EC2 jobs.

-ssh ***ssh-command*** Specify an alternate *ssh* program to run in place of *ssh*, for example *sftp* or *scp*. Additional arguments are specified as *ssh-command*. Since the arguments are delimited by spaces, place double quote marks around the whole command, to prevent the shell from splitting it into multiple arguments to *condor_ssh_to_job*. If any arguments must contain spaces, enclose them within single quotes. Does not apply to EC2 jobs.

-keygen-options ***ssh-keygen-options*** Specify additional arguments to the *ssh_keygen* program, for creating the ssh key that is used for the duration of the session. For example, a different number of bits could be used, or a different key type than the default. Does not apply to EC2 jobs.

-shells ***shell1,shell2,...*** Specify a comma-separated list of shells to attempt to launch. If the first shell does not exist on the remote machine, then the following ones in the list will be tried. If none of the specified shells can be found, */bin/sh* is used by default. If this option is not specified, it defaults to the environment variable `SHELL` from within the *condor_ssh_to_job* environment. Does not apply to EC2 jobs.

-auto-retry Specifies that if the job is not yet running, *condor_ssh_to_job* should keep trying periodically until it succeeds or encounters some other error.

-remove-on-interrupt If specified, attempt to remove the job from the queue if *condor_ssh_to_job* is interrupted via a CTRL-c or otherwise terminated abnormally.

-X Enable X11 forwarding. Does not apply to EC2 jobs.

-x Disable X11 forwarding.

Examples

```
% condor_ssh_to_job 32.0
Welcome to slot2@tonic.cs.wisc.edu!
Your condor job is running with pid(s) 65881.
% gdb -p 65881
(gdb) where
...
% logout
Connection to condor-job.tonic.cs.wisc.edu closed.
```

To upload or download files interactively with *sftp*:

```
% condor_ssh_to_job -ssh sftp 32.0
Connecting to condor-job.tonic.cs.wisc.edu...
sftp> ls
...
sftp> get outputfile.dat
```

This example shows downloading a file from the job with *scp*. The string "remote" is used in place of a host name in this example. It is not necessary to insert the correct remote host name, or even a valid one, because the connection to the job is created automatically. Therefore, the placeholder string "remote" is perfectly fine.

```
% condor_ssh_to_job -ssh scp 32 remote:outputfile.dat .
```

This example uses *condor_ssh_to_job* to accomplish the task of running *rsync* to synchronize a local file with a remote file in the job's working directory. Job id 32.0 is used in place of a host name in this example. This causes *rsync* to insert the expected job id in the arguments to *condor_ssh_to_job*.

```
% rsync -v -e "condor_ssh_to_job" 32.0:outputfile.dat .
```

Note that *condor_ssh_to_job* was added to HTCondor in version 7.3. If one uses *condor_ssh_to_job* to connect to a job on an execute machine running a version of HTCondor older than the 7.3 series, the command will fail with the error message

```
Failed to send CREATE_JOB_OWNER_SEC_SESSION to starter
```

Exit Status

condor_ssh_to_job will exit with a non-zero status value if it fails to set up an ssh session. If it succeeds, it will exit with the status value of the remote command or shell.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_sos

Issue a command that will be serviced with a higher priority

Synopsis

condor_sos [**-help** | **-version**]

condor_sos [**-debug**] [**-timeoutmult** *value*] *condor_command*

Description

condor_sos sends the *condor_command* in such a way that the command is serviced ahead of other waiting commands. It appears to have a higher priority than other waiting commands.

condor_sos is intended to give administrators a way to query the *condor_schedd* and *condor_collector* daemons when they are under such a heavy load that they are not responsive.

There must be a special command port configured, in order for a command to be serviced with priority. The *condor_schedd* and *condor_collector* always have the special command port. Other daemons require configuration by setting configuration variable `<SUBSYS>_SUPER_ADDRESS_FILE`.

Options

-help Display usage information

-version Display version information

-debug Print extra debugging information as the command executes.

-timeoutmult *value* Multiply any timeouts set for the command by the integer *value*.

Examples

The example command

```
condor_sos -timeoutmult 5 condor_hold -all
```

causes the `condor_hold -all` command to be handled by the *condor_schedd* with priority over any other commands that the *condor_schedd* has waiting to be serviced. It also extends any set timeouts by a factor of 5.

Exit Status

condor_sos will exit with the value 1 on error and with the exit value of the invoked command when the command is successfully invoked.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_stats

Display historical information about the HTCondor pool

Synopsis

condor_stats [**-f** *filename*] [**-orgformat**] [**-pool** *centralmanagerhostname[:portnumber]*] [**time-range**] *query-type*

Description

condor_stats displays historic information about an HTCondor 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, owner, shutdown, delete, backfill, and drained 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 MiB received, MiB sent, average receive bandwidth (in KiB/sec), and average send bandwidth (in KiB/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 *resoursequery*, *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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_status

Display status of the HTCondor 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 HTCondor 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 *display option* may be specified. Any number of *custom options* 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 ClassAd query without performing the query.

-absent (Query option) Query for and display only absent resources.

-ads *filename* (Query option) Read the set of ClassAds in the file specified by *filename*, instead of querying the *condor_collector*.

-any (Query option) Query all ClassAds and display their type, target type, and name.

-avail (Query option) Query *condor_startd* ClassAds and identify resources which are available.

-ckptsrvr (Query option) Query *condor_ckpt_server* ClassAds and display checkpoint server attributes.

-claimed (Query option) Query *condor_startd* ClassAds 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.

-collector (Query option) Query *condor_collector* ClassAds and display attributes.

-defrag (Query option) Query *condor_defrag* ClassAds.

-direct *hostname* (Query option) Go directly to the given host name to get the ClassAds to display. By default, returns the *condor_startd* ClassAd. If **-schedd** is also given, return the *condor_schedd* ClassAd on that host.

-java (Query option) Display only Java-capable resources.

-license (Query option) Display license attributes.

-master (Query option) Query *condor_master* ClassAds and display daemon master attributes.

-negotiator (Query option) Query *condor_negotiator* ClassAds 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.

-run (Query option) Display information about machines currently running jobs.

-schedd (Query option) Query *condor_schedd* ClassAds and display attributes.

-server (Query option) Query *condor_startd* ClassAds and display resource attributes.

-startd (Query option) Query *condor_startd* ClassAds.

-state (Query option) Query *condor_startd* ClassAds and display resource state information.

-statistics *WhichStatistics* (Query option) Can only be used if the **-direct** option has been specified. Identifies which Statistics attributes to include in the ClassAd. *WhichStatistics* is specified using the same syntax as defined for `STATISTICS_TO_PUBLISH`. A definition is in the HTCondor Administrator's manual section on configuration.

-storage (Query option) Display attributes of machines with network storage resources.

-submitters (Query option) Query ClassAds 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.

-offline (Query option) Query *condor_startd* ClassAds, and display, for each machine with at least one offline universe, which universes are offline for it.

-attributes *Attr1[,Attr2 . . .]* (Display option) Explicitly list the attributes in a comma separated list which should be displayed when using the **-xml**, **-json** or **-long** options. Limiting the number of attributes increases the efficiency of the query.

-expert (Display option) Display shortened error messages.

-long (Display option) Display entire ClassAds. Implies that totals will not be displayed.

-sort *expr* (Display option) Change the display order to be based on ascending values of an evaluated expression given by *expr*. Evaluated expressions of a string type are in a case insensitive alphabetical order. If multiple **-sort** arguments appear on the command line, the primary sort will be on the leftmost one within the command line, and it is numbered 0. A secondary sort will be based on the second expression, and it is numbered 1. For

informational or debugging purposes, the ClassAd output to be displayed will appear as if the ClassAd had two additional attributes. `CondorStatusSortKeyExpr<N>` is the expression, where `<N>` is replaced by the number of the sort. `CondorStatusSortKey<N>` gives the result of evaluating the sort expression that is numbered `<N>`.

-total (Display option) Display totals only.

-xml (Display option) Display entire ClassAds, in XML format. The XML format is fully defined in the reference manual, obtained from the ClassAds web page, with a link at http://htcondor.org/classad/classad.html.

-json (Display option) Display entire ClassAds in JSON format.

-constraint *const* (Custom option) Add constraint expression.

-compact (Custom option) Show compact form, rolling up slots into a single line.

-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. %r prints the unevaluated, or raw values. 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`. %v identifies the type of the attribute, and then prints the value in an appropriate format. %V identifies the type of the attribute, and then prints the value in an appropriate format as it would appear in the **-long** format. As an example, strings used with %V will have quote marks. 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.

-autoformat[:lhVr,tng] *attr1 [attr2 ...]* or -af[:lhVr,tng] *attr1 [attr2 ...]* (Output option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings. This output option does *not* work in conjunction with the **-run** option.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may *not* be used together. The **l** and **h** characters may *not* be used together.

-target ***filename*** (Custom option) Where evaluation requires a target ClassAd to evaluate against, file *filename* contains the target ClassAd.

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 HTCondor 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 HTCondor 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, HTCondor 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

The Unix command to use the constraint option to see all machines with the `OpSys` of `"LINUX"`:

```
% condor_status -constraint OpSys==\"LINUX\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

The Windows command to do the same thing:

```
>condor_status -constraint " OpSys==""LINUX"" "
```

Note that quotation marks are used to delimit the single argument which is the expression, and the quotation marks that identify the string must be escaped by using a set of two double quote marks without any intervening spaces.

To see all machines that are currently in the Idle state, the Unix command is

```
% condor_status -constraint State==\"Idle\"
```

To see all machines that are bench marked to have a MIPS rating of more than 750, the Unix command is

```
% condor_status -constraint 'Mips>750'
```

-cod option example

The **-cod** option displays the status of COD claims within a given HTCondor 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
```

-format option example To display the name and memory attributes of each job ClassAd in a format that is easily parsable by other tools:

```
% condor_status -format "%s " Name -format "%d\n" Memory
```

To do the same with the **autoformat** option, run

```
% condor_status -autoformat Name Memory
```

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_store_cred

securely stash a 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

condor_store_cred stores passwords in a secure manner. There are two separate uses of *condor_store_cred*:

1. A shared pool password is needed in order to implement the `PASSWORD` authentication method. *condor_store_cred* using the **-c** option deals with the password for the implied `condor_pool@$(UID_DOMAIN)` user name.

   On a Unix machine, *condor_store_cred* with the **-f** option is used to set the pool password, as needed when used with the `PASSWORD` authentication method. The pool password is placed in a file specified by the `SEC_PASSWORD_FILE` configuration variable.

2. In order to submit a job from a Windows platform machine, or to execute a job on a Windows platform machine utilizing the **run_as_owner** functionality, *condor_store_cred* stores the password of a user/domain pair securely in the Windows registry. Using this stored password, HTCondor may act on behalf of the submitting user to access files, such as writing output or log files. HTCondor is able to run jobs with the user ID of the submitting user. The password is stored in the same manner as the system does when setting or changing account passwords.

Passwords are stashed in a persistent manner; they are maintained across system reboots.

The *add* argument on the Windows platform stores the 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* argument deletes the current password, if it exists.

The *query* reports whether the password is stored or not.

Options

-c Operations refer to the pool password, as used in the `PASSWORD` authentication method.

-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 *password*, rather than prompting the user to enter a password.

-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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_submit

Queue jobs for execution under HTCondor

Synopsis

condor_submit [**-terse**] [**-verbose**] [**-unused**] [**-name** *schedd_name*] [**-remote** *schedd_name*] [**-addr** <*ip:port*>] [**-pool** *pool_name*] [**-disable**] [**-password** *passphrase*] [**-debug**] [**-append** *command* **...**][**-batch-name** *batch_name*] [**-spool**] [**-dump** *filename*] [**-interactive**] [**-dry-run**] [**-maxjobs** *number-of-jobs*] [**-single-cluster**] [**-stm** *method*] [**<submit-variable>=<value>**] [*submit description file*] [**-queue** *queue_arguments*]

Description

condor_submit is the program for submitting jobs for execution under HTCondor. *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 HTCondor 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 HTCondor to start the next job in a cluster than for HTCondor 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.

The *submit description file* argument is the path and file name of the submit description file. If this optional argument is missing or is the dash character (−), then the commands are taken from standard input. If − is specified for the *submit description file*, **-verbose** is implied; this can be overridden by specifying **-terse**.

Note that submission of jobs from a Windows machine requires a stashed password to allow HTCondor to impersonate the user submitting the job. To stash a password, use the *condor_store_cred* command. See the manual page for details.

For lengthy lines within the submit description file, the backslash (\) is a line continuation character. Placing the backslash at the end of a line causes the current line's command to be continued with the next line of the file. Submit description files may contain comments. A comment is any line beginning with a pound character (#).

Options

-terse Terse output - display JobId ranges only.

-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 non default 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**.

-addr ***<ip:port>*** Submit to the *condor_schedd* at the IP address and port given by the *sinful string* argument *<ip:port>*.

-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 when submitting a job for read permissions on all input files, such as those defined by commands **input** and **transfer_input_files**, as well as write permission to output files, such as a log file defined by **log** and output files defined with **output** or **transfer_output_files**.

-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 `TOOL_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. If the *command* specifies a **queue** command, as in the example

```
condor_submit mysubmitfile -append "queue input in A, B, C"
```

then the entire **-append** command line option and its arguments are converted to

```
condor_submit mysubmitfile -queue input in A, B, C
```

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.

-batch-name ***batch_name*** Set the batch name for this submit. The batch name is displayed by *condor_q* **-batch**. It is intended for use by users to give meaningful names to their jobs and to influence how *condor_q* groups jobs for display. Use of this argument takes precedence over a batch name specified in the submit description file itself.

-spool Spool all required input files, job event 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*.

-interactive Indicates that the user wants to run an interactive shell on an execute machine in the pool. This is equivalent to creating a submit description file of a vanilla universe sleep job, and then running *condor_ssh_to_job* by hand. Without any additional arguments, *condor_submit* with the -interactive flag creates a dummy vanilla universe job that sleeps, submits it to the local scheduler, waits for the job to run, and then launches *condor_ssh_to_job* to run a shell. If the user would like to run the shell on a machine that matches a particular **requirements** expression, the submit description file is specified, and it will contain the expression. Note that all policy expressions specified in the submit description file are honored, but any **executable** or **universe** commands are overwritten to be sleep and vanilla. The job ClassAd attribute `InteractiveJob` is set to `True` to identify interactive jobs for *condor_startd* policy usage.

-dry-run ***file*** Parse the submit description file, sending the resulting job ClassAd to the file given by *file*, but do not submit the job(s). This permits observation of the job specification, and it facilitates debugging the submit description file contents. If *file* is **-**, the output is written to `stdout`.

-maxjobs ***number-of-jobs*** If the total number of jobs specified by the submit description file is more than the integer value given by *number-of-jobs*, then no jobs are submitted for execution and an error message is generated. A 0 or negative value for the *number-of-jobs* causes no limit to be imposed.

-single-cluster If the jobs specified by the submit description file causes more than a single cluster value to be assigned, then no jobs are submitted for execution and an error message is generated.

-stm ***method*** Specify the method use to move a sandbox into HTCondor. *method* is one of **stm_use_schedd_only** or **stm_use_transferd**.

<submit-variable>=<value> Defines a submit command or submit variable with a value, and parses it as if it was placed at the beginning of the submit description file. The submit description file is not changed. To correctly parse the *condor_submit* command line, this option must be specified without white space characters before and after the equals sign (=), or the entire option must be surrounded by double quote marks.

-queue *queue_arguments* A command line specification of how many jobs to queue, which is only permitted if the submit description file does not have a **queue** command. The *queue_arguments* are the same as may be within a submit description file. The parsing of the *queue_arguments* finishes at the end of the line or when a dash character (−) is encountered. Therefore, its best placement within the command line will be at the end of the command line.

On a Unix command line, the shell expands file globs before parsing occurs.

Submit Description File Commands

Note: more information on submitting HTCondor jobs can be found here: 2.5.

As of version 8.5.6, the *condor_submit* language supports multi-line values in commands. The syntax is the same as the configuration language (see more details here: 3.3.5).

Each submit description file describes one or more clusters of jobs to be placed in the HTCondor 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 generally the last 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 at least one *executable* command and at least one *queue* command. All of the other commands have default actions.

Note that a submit file that contains more than one executable command will produce multiple clusters when submitted. This is not generally recommended, and is not allowed for submit files that are run as DAG node jobs by *condor_dagman*.

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 executable as part of 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, identified as the old syntax and the new syntax. The old syntax supports white space characters within arguments only in special circumstances; when used, the command line arguments are represented in the job ClassAd attribute `Args`. The new syntax supports uniform quoting of white space characters within arguments; when used, the command line arguments are represented in the job ClassAd attribute `Arguments`.

Old Syntax

In the old syntax, individual command line arguments are delimited (separated) by space characters. To allow a double quote mark in an argument, it is escaped with a backslash; that is, the two character sequence `\"` becomes a single double quote mark within an argument.

Further interpretation of the argument string differs depending on the operating system. On Windows, the entire argument string is passed verbatim (other than the backslash in front of double quote marks) to the Windows

application. Most Windows applications will allow spaces within an argument value by surrounding the argument with double quotes marks. 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'
```

New Syntax

Here are the rules for using the new syntax:

1. The entire string representing the command line arguments is surrounded by double quote marks. This permits the white space characters of spaces and tabs to potentially be embedded within a single argument. Putting the double quote mark within the arguments is accomplished by escaping it with another double quote mark.
2. The white space characters of spaces or tabs delimit arguments.
3. To embed white space characters of spaces or tabs within a single argument, surround the entire argument with single quote marks.
4. To insert a literal single quote mark, escape it within an argument already delimited by single quote marks by adding another single quote mark.

Example:

```
arguments = "3 simple arguments"
```

Produces:

```
argument 1: 3
argument 2: simple
argument 3: arguments
```

Another example:

```
arguments = "one 'two with spaces' 3"
```

Produces:

```
argument 1: one
argument 2: two with spaces
argument 3: 3
```

And yet another 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, the 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 HTCondor to capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). A path is given with respect to the file system of the machine on which the job is submitted. The file is written (by the job) in the remote scratch directory of the machine where the job is executed. When the job exits, the resulting file is transferred back to the machine where the job was submitted, and the path is utilized for file placement. 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 HTCondor 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> HTCondor 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`). A path is given with respect to the file system of the machine on which the job is submitted. The file is transferred before execution to the remote scratch directory of the machine where the job is executed. 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 HTCondor will write a log file of what is happening with this job cluster, called a job event log. For example, HTCondor 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, HTCondor does not create a log for this cluster. If a relative path is specified, it is relative to the current working directory as the job is submitted or the directory specified by submit command **initialdir** on the submit machine.

log_xml = <True | False> If **log_xml** is `True`, then the job event 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 job event log file, ensure that all of the jobs specify this option in the same way.

notification = <Always | Complete | Error | Never> Owners of HTCondor 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 owner will be notified when the job terminates. If defined by *Error*, the owner will only be notified if the job terminates abnormally, or if the job is placed on hold because of a failure, and not by user request. If defined by *Never* (the default), the owner will not receive e-mail, regardless to what happens to the job. The HTCondor User's manual documents statistics included in the e-mail.

notify_user = <email-address> Used to specify the e-mail address to use when HTCondor sends e-mail about a job. If not specified, HTCondor defaults to using the e-mail address defined by

```
job-owner@UID_DOMAIN
```

where the configuration variable `UID_DOMAIN` is specified by the HTCondor site administrator. If `UID_DOMAIN` has not been specified, HTCondor 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`). A path is given with respect to the file system of the machine on which the job is submitted. The file is written (by the job) in the remote scratch directory of the machine where the job is executed. When the job exits, the resulting file is transferred back to the machine where the job was submitted, and the path is utilized for file placement. 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> An HTCondor 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.

Note that the priority setting in an HTCondor submit file will be overridden by *condor_dagman* if the submit file is used for a node in a DAG, and the priority of the node within the DAG is non-zero (see 2.10.9 for more details).

queue [<int expr>] Places zero or more copies of the job into the HTCondor queue.

queue [**<int expr>**] [**<varname>**] **in** [**slice**] **<list of items>** Places zero or more copies of the job in the queue based on items in a **<list of items>**

queue [**<int expr>**] [**<varname>**] **matching** [**files | dirs**] [**slice**] **<list of items with file globbing>**] Places zero or more copies of the job in the queue based on files that match a **<list of items with file globbing>**

queue [**<int expr>**] [**<list of varnames>**] **from** [**slice**] **<file name> | <list of items>**] Places zero or more copies of the job in the queue based on lines from the submit file or from **<file name>**

The optional argument <*int expr*> specifies how many times to repeat the job submission for a given set of arguments. It may be an integer or an expression that evaluates to an integer, and it defaults to 1. All but the first form of this command are various ways of specifying a list of items. When these forms are used <*int expr*> jobs will be queued for each item in the list. The *in*, *matching* and *from* keyword indicates how the list will be specified.

- *in* The list of items is an explicit comma and/or space separated **<list of items>**. If the **<list of items>** begins with an open paren, and the close paren is not on the same line as the open, then the list continues until a line that begins with a close paren is read from the submit file.
- *matching* Each item in the **<list of items with file globbing>** will be matched against the names of files and directories relative to the current directory, the set of matching names is the resulting list of items.
  - *files* Only filenames will matched.
  - *dirs* Only directory names will be matched.
- *from* **<file name> | <list of items>** Each line from **<file name>** or **<list of items>** is a single item, this allows for multiple variables to be set for each item. Lines from **<file name>** or **<list of items>** will be split on comma and/or space until there are values for each of the variables specified in **<list of varnames>**. The last variable will contain the remainder of the line. When the **<list of items>** form is used, the list continues until the first line that begins with a close paren, and lines beginning with pound sign ('#') will be skipped. When using the **<file name>** form, if the **<file name>** ends with |, then it will be executed as a script whatever the script writes to `stdout` will be the list of items.

The optional argument <*varname*> or <*list of varnames*> is the name or names of of variables that will be set to the value of the current item when queuing the job. If no <*varname*> is specified the variable ITEM will be used.

Leading and trailing whitespace be trimmed. The optional argument <*slice*> is a python style slice selecting only some of the items in the list of items. Negative step values are not supported.

A submit file may contain more than one **queue** statement, and 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 | java | vm | parallel | docker> Specifies which HTCondor universe to use when running this job. The HTCondor universe specifies an HTCondor execution environment.

The **vanilla** universe is the default (except where the configuration variable `DEFAULT_UNIVERSE` defines it otherwise), and is an execution environment for jobs which do not use HTCondor's mechanisms for taking checkpoints; these are ones that have not been linked with the HTCondor libraries. Use the **vanilla** universe to submit shell scripts to HTCondor.

The **standard** universe tells HTCondor that this job has been re-linked via *condor_compile* with the HTCondor libraries and therefore supports taking checkpoints and remote system calls.

The **scheduler** universe is for a job that is to run on the machine where the job is submitted. This universe is intended for a job that acts as a metascheduler and will not be preempted.

The **local** universe is for a job that is to run on the machine where the job is submitted. This universe runs the job immediately and will not preempt the job.

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 **java** universe is for programs written to the Java Virtual Machine.

The **vm** universe facilitates the execution of a virtual machine.

The **parallel** universe is for parallel jobs (e.g. MPI) that require multiple machines in order to run.

The **docker** universe runs a docker container as an HTCondor job.

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. HTCondor will give the job the machine with the highest rank. For example,

```
request_memory = max({60, Target.TotalSlotMemory})
rank = Memory
```

asks HTCondor 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. The HTCondor User's Manual contains complete information on the syntax and available attributes that can be used in the ClassAd expression.

request_cpus = <num-cpus> A requested number of CPUs (cores). If not specified, the number requested will be 1. If specified, the expression

```
&& (RequestCpus <= Target.Cpus)
```

is appended to the **requirements** expression for the job.

For pools that enable dynamic *condor_startd* provisioning, specifies the minimum number of CPUs requested for this job, resulting in a dynamic slot being created with this many cores.

request_disk = <quantity> The requested amount of disk space in KiB requested for this job. If not specified, it will be set to the job ClassAd attribute `DiskUsage`. The expression

```
&& (RequestDisk <= Target.Disk)
```

is appended to the **requirements** expression for the job.

For pools that enable dynamic *condor_startd* provisioning, a dynamic slot will be created with at least this much disk space.

Characters may be appended to a numerical value to indicate units. `K` or `KB` indicates KiB, 2^{10} numbers of bytes. `M` or `MB` indicates MiB, 2^{20} numbers of bytes. `G` or `GB` indicates GiB, 2^{30} numbers of bytes. `T` or `TB` indicates TiB, 2^{40} numbers of bytes.

request_memory = <quantity> The required amount of memory in MiB that this job needs to avoid excessive swapping. If not specified and the submit command **vm_memory** is specified, then the value specified for **vm_memory** defines **request_memory**. If neither **request_memory** nor **vm_memory** is specified, the value is set by the configuration variable `JOB_DEFAULT_REQUESTMEMORY`. The actual amount of memory used by a job is represented by the job ClassAd attribute `MemoryUsage`.

For pools that enable dynamic *condor_startd* provisioning, a dynamic slot will be created with at least this much RAM.

The expression

```
&& (RequestMemory <= Target.Memory)
```

is appended to the **requirements** expression for the job.

Characters may be appended to a numerical value to indicate units. `K` or `KB` indicates KiB, 2^{10} numbers of bytes. `M` or `MB` indicates MiB, 2^{20} numbers of bytes. `G` or `GB` indicates GiB, 2^{30} numbers of bytes. `T` or `TB` indicates TiB, 2^{40} numbers of bytes.

request_<name> = <quantity> The required amount of the custom machine resource identified by `<name>` that this job needs. The custom machine resource is defined in the machine's configuration. Machines that have available GPUs will define `<name>` to be `GPUs`.

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 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, HTCondor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.

2. Cpus >= RequestCpus, if the job ClassAd attribute `RequestCpus` is defined.
3. Disk >= RequestDisk, if the job ClassAd attribute `RequestDisk` is defined. Otherwise, Disk >= DiskUsage is appended to the requirements. 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 HTCondor 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. HTCondor 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.
4. Memory >= RequestMemory, if the job ClassAd attribute `RequestMemory` is defined.
5. 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 829. Also, see the HTCondor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

FILE TRANSFER COMMANDS

dont_encrypt_input_files = < file1,file2,file... > A comma and/or space separated list of input files that are *not* to be network encrypted when transferred with the file transfer mechanism. Specification of files in this manner overrides configuration that would use encryption. Each input file must also be in the list given by **transfer_input_files**. When a path to an input file or directory is specified, this specifies the path to the file on the submit side. A single wild card character (*) may be used in each file name.

dont_encrypt_output_files = < file1,file2,file... > A comma and/or space separated list of output files that are *not* to be network encrypted when transferred back with the file transfer mechanism. Specification of files in this manner overrides configuration that would use encryption. The output file(s) must also either be in the list given by **transfer_output_files** or be discovered and to be transferred back with the file transfer mechanism. When a path to an output file or directory is specified, this specifies the path to the file on the execute side. A single wild card character (*) may be used in each file name.

encrypt_execute_directory = <True | False> Defaults to `False`. If set to `True`, HTCondor will encrypt the contents of the remote scratch directory of the machine where the job is executed. This encryption is transparent to the job itself, but ensures that files left behind on the local disk of the execute machine, perhaps due to a system crash, will remain private. In addition, *condor_submit* will append to the job's **requirements** expression

```
&& (TARGET.HasEncryptExecuteDirectory)
```

to ensure the job is matched to a machine that is capable of encrypting the contents of the execute directory. This support is limited to Windows platforms that use the NTFS file system and Linux platforms with the *ecryptfs-utils* package installed.

encrypt_input_files = < file1,file2,file... > A comma and/or space separated list of input files that are to be network encrypted when transferred with the file transfer mechanism. Specification of files in this manner overrides configuration that would *not* use encryption. Each input file must also be in the list given by **transfer_input_files**. When a path to an input file or directory is specified, this specifies the path to the file on the submit side. A single

wild card character (*) may be used in each file name. The method of encryption utilized will be as agreed upon in security negotiation; if that negotiation failed, then the file transfer mechanism must also fail for files to be network encrypted.

encrypt_output_files = < file1,file2,file... > A comma and/or space separated list of output files that are to be network encrypted when transferred back with the file transfer mechanism. Specification of files in this manner overrides configuration that would *not* use encryption. The output file(s) must also either be in the list given by **transfer_output_files** or be discovered and to be transferred back with the file transfer mechanism. When a path to an output file or directory is specified, this specifies the path to the file on the execute side. A single wild card character (*) may be used in each file name. The method of encryption utilized will be as agreed upon in security negotiation; if that negotiation failed, then the file transfer mechanism must also fail for files to be network encrypted.

max_transfer_input_mb = <ClassAd Integer Expression> This integer expression specifies the maximum allowed total size in MiB of the input files that are transferred for a job. This expression does *not* apply to grid universe, standard universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not defined, the value set by configuration variable `MAX_TRANSFER_INPUT_MB` is used. If the observed size of all input files at submit time is larger than the limit, the job will be immediately placed on hold with a `HoldReasonCode` value of 32. If the job passes this initial test, but the size of the input files increases or the limit decreases so that the limit is violated, the job will be placed on hold at the time when the file transfer is attempted.

max_transfer_output_mb = <ClassAd Integer Expression> This integer expression specifies the maximum allowed total size in MiB of the output files that are transferred for a job. This expression does *not* apply to grid universe, standard universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not set, the value set by configuration variable `MAX_TRANSFER_OUTPUT_MB` is used. If the total size of the job's output files to be transferred is larger than the limit, the job will be placed on hold with a `HoldReasonCode` value of 33. The output will be transferred up to the point when the limit is hit, so some files may be fully transferred, some partially, and some not at all.

output_destination = <destination-URL> When present, defines a URL that specifies both a plug-in and a destination for the transfer of the entire output sandbox or a subset of output files as specified by the submit command **transfer_output_files**. The plug-in does the transfer of files, and no files are sent back to the submit machine. The HTCondor Administrator's manual has full details.

should_transfer_files = <YES | NO | IF_NEEDED > The **should_transfer_files** setting is used to define if HTCondor 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 HTCondor to always transfer files for the job. *NO* disables HTCondor'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`, HTCondor will transfer the necessary files.

For more information about this and other settings related to transferring files, see the HTCondor User's manual section on the file transfer mechanism.

Note that **should_transfer_files** is not supported for jobs submitted to the grid universe.

skip_filechecks = <True | False> When `True`, file permission checks for the submitted job are disabled. When `False`, file permissions are checked; this is the behavior when this command is not present in the submit description file. File permissions are checked for read permissions on all input files, such as those defined by commands **input** and **transfer_input_files**, and for write permission to output files, such as a log file defined by **log** and output files defined with **output** or **transfer_output_files**.

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 `False`. 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 `False`. 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 and vanilla universes. If **transfer_executable** is set to `False`, then HTCondor looks for the executable on the remote machine, and does not transfer the executable over. This is useful for an already pre-staged executable; HTCondor behaves more like rsh. The default value is `True`.

transfer_input_files = < file1,file2,file... > A comma-delimited list of all the files and directories 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.

When a path to an input file or directory is specified, this specifies the path to the file on the submit side. The file is placed in the job's temporary scratch directory on the execute side, and it is named using the base name of the original path. For example, `/path/to/input_file` becomes `input_file` in the job's scratch directory.

A directory may be specified by appending the forward slash character (/) as a trailing path separator. This syntax is used for both Windows and Linux submit hosts. A directory example using a trailing path separator is `input_data/`. When a directory is specified with the trailing path separator, the contents of the directory are transferred, but the directory itself is not transferred. It is as if each of the items within the directory were listed in the transfer list. When there is no trailing path separator, the directory is transferred, its contents are transferred, and these contents are placed inside the transferred directory.

For grid universe jobs other than HTCondor-C, the transfer of directories is not currently supported.

Symbolic links to files are transferred as the files they point to. Transfer of symbolic links to directories is not currently supported.

For vanilla and vm universe jobs only, a file may be specified by giving a URL, instead of a file name. The implementation for URL transfers requires both configuration and available plug-in.

transfer_output_files = < file1,file2,file... > This command forms an explicit list of output files and directories to be transferred back from the temporary working directory on the execute machine to the submit machine. If there are multiple files, they must be delimited with commas. Setting **transfer_output_files** to the empty string (`""`) means that no files are to be transferred.

For HTCondor-C jobs and all other non-grid universe jobs, if **transfer_output_files** is not specified, HTCondor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job. Subdirectories are not scanned for output, so if output from subdirectories is desired, the output list must be explicitly specified. For grid universe jobs other than HTCondor-C, desired output files must also be explicitly listed. Another reason to explicitly list output files is for a job that creates many files, and the user wants only a subset transferred back.

For grid universe jobs other than with grid type **condor**, 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.

When a path to an output file or directory is specified, it specifies the path to the file on the execute side. As a destination on the submit side, the file is placed in the job's initial working directory, and it is named using the base name of the original path. For example, `path/to/output_file` becomes `output_file` in the job's initial working directory. The name and path of the file that is written on the submit side may be modified by using **transfer_output_remaps**. Note that this remap function only works with files but not with directories.

A directory may be specified using a trailing path separator. An example of a trailing path separator is the slash character on Unix platforms; a directory example using a trailing path separator is `input_data/`. When a directory is specified with a trailing path separator, the contents of the directory are transferred, but the directory itself is not transferred. It is as if each of the items within the directory were listed in the transfer list. When there is no trailing path separator, the directory is transferred, its contents are transferred, and these contents are placed inside the transferred directory.

For grid universe jobs other than HTCondor-C, the transfer of directories is not currently supported.

Symbolic links to files are transferred as the files they point to. Transfer of symbolic links to directories is not currently supported.

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. You cannot specify directories to be remapped.

when_to_transfer_output = < ON_EXIT | ON_EXIT_OR_EVICT > Setting **when_to_transfer_output** equal to *ON_EXIT* will cause HTCondor 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 HTCondor 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.

POLICY COMMANDS

max_retries = <integer> The maximum number of retries allowed for this job (must be non-negative). If the job fails (does not exit with the **success_exit_code** exit code) it will be retried up to **max_retries** times (unless retries are ceased because of the **retry_until** command). If **max_retries** is not defined, and either **retry_until** or **success_exit_code** is, the value of `DEFAULT_JOB_MAX_RETRIES` will be used for the maximum number of retries.

The combination of the **max_retries**, **retry_until**, and **success_exit_code** commands causes an appropriate `OnExitRemove` expression to be automatically generated. If retry command(s) and **on_exit_remove** are both defined, the `OnExitRemove` expression will be generated by OR'ing the expression specified in `OnExitRemove` and the expression generated by the retry commands.

retry_until <Integer | ClassAd Boolean Expression> An integer value or boolean expression that prevents further retries from taking place, even if **max_retries** have not been exhausted. If **retry_until** is an integer, the job exiting with that exit code will cause retries to cease. If **retry_until** is a ClassAd expression, the expression evaluating to `True` will cause retries to cease.

success_exit_code = <integer> The exit code that is considered successful for this job. Defaults to 0 if not defined.

Note: non-zero values of success_exit_code should generally not be used for DAG node jobs. At the present time, *condor_dagman* does not take into account the value of **success_exit_code**. This means that, if **success_exit_code** is set to a non-zero value, *condor_dagman* will consider the job failed when it actually succeeds. For single-proc DAG node jobs, this can be overcome by using a POST script that takes into account the value of **success_exit_code** (although this is not recommended). For multi-proc DAG node jobs, there is currently no way to overcome this limitation.

hold = <True | False> If **hold** is set to `True`, then the submitted job will be placed into the Hold state. Jobs in the Hold state will not run until released by *condor_release*. Defaults to `False`.

keep_claim_idle = <integer> An integer number of seconds that a job requests the *condor_schedd* to wait before releasing its claim after the job exits or after the job is removed.

The process by which the *condor_schedd* claims a *condor_startd* is somewhat time-consuming. To amortize this cost, the *condor_schedd* tries to reuse claims to run subsequent jobs, after a job using a claim is done. However, it can only do this if there is an idle job in the queue at the moment the previous job completes. Sometimes, and especially for the node jobs when using DAGMan, there is a subsequent job about to be submitted, but it has not yet arrived in the queue when the previous job completes. As a result, the *condor_schedd* releases the claim, and the next job must wait an entire negotiation cycle to start. When this submit command is defined with a non-negative integer, when the job exits, the *condor_schedd* tries as usual to reuse the claim. If it cannot, instead of releasing the claim, the *condor_schedd* keeps the claim until either the number of seconds given as a parameter, or a new job which matches that claim arrives, whichever comes first. The *condor_startd* in question will remain in the Claimed/Idle state, and the original job will be "charged" (in terms of priority) for the time in this state.

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 HTCondor 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`.

As an example, if the job is to be removed once the output is retrieved with *condor_transfer_data*, then use

```
leave_in_queue = (JobStatus == 4) && ((StageOutFinish =?= UNDEFINED) ||\
                 (StageOutFinish == 0))
```

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 HTCondor configuration variable `MAX_NEXT_JOB_START_DELAY`, which defaults to 10 minutes. This command does not apply to **scheduler** or **local** universe jobs.

This command has been historically used to implement a form of job start throttling from the job submitter's perspective. It was effective for the case of multiple job submission where the transfer of extremely large input data sets to the execute machine caused machine performance to suffer. This command is no longer useful, as throttling should be accomplished through configuration of the *condor_schedd* daemon.

on_exit_hold = <ClassAd Boolean Expression> The ClassAd expression is checked when the job exits, and if `True`, places the job into the Hold state. If `False` (the default value when not defined), 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 = (time() - 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.

Only job ClassAd attributes will be defined for use by this ClassAd expression. This expression is available for the vanilla, java, parallel, grid, local and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

on_exit_hold_reason = <ClassAd String Expression> When the job is placed on hold due to the **on_exit_hold** expression becoming `True`, this expression is evaluated to set the value of `HoldReason` in the job ClassAd. If this expression is `UNDEFINED` or produces an empty or invalid string, a default description is used.

on_exit_hold_subcode = <ClassAd Integer Expression> When the job is placed on hold due to the **on_exit_hold** expression becoming `True`, this expression is evaluated to set the value of `HoldReasonSubCode` in the job ClassAd. The default subcode is 0. The `HoldReasonCode` will be set to 3, which indicates that the job went on hold due to a job policy expression.

on_exit_remove = <ClassAd Boolean Expression> The ClassAd expression is checked when the job exits, and if `True` (the default value when undefined), 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 a job occasionally segfaults, but chances are that the job will finish successfully if the job is run again with the same data. The **on_exit_remove** expression can cause the job to run again with the following command. Assume that the signal identifier for the segmentation fault is 11 on the platform where the job will be running.

```
on_exit_remove = (ExitBySignal == False) || (ExitSignal != 11)
```

This expression lets the job leave the queue if the job was not killed by a signal or if it was killed by a signal other than 11, representing segmentation fault in this example. So, if the exited due to 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 the job should only leave the queue if it exited on its own with status 0, this **on_exit_remove** expression works well:

```
on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)
```

If the job was killed by a signal or exited with a non-zero exit status, HTCondor would leave the job in the queue to run again.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression.

periodic_hold = <ClassAd Boolean Expression> This expression is checked periodically when the job is not in the Held state. If it becomes `True`, the job will be placed on hold. If unspecified, the default value is `False`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

Only job ClassAd attributes will be defined for use by this ClassAd expression. Note that, by default, this expression is only checked once every 60 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL`, `MAX_PERIODIC_EXPR_INTERVAL`, and `PERIODIC_EXPR_TIMESLICE` configuration macros.

periodic_hold_reason = <ClassAd String Expression> When the job is placed on hold due to the **periodic_hold** expression becoming `True`, this expression is evaluated to set the value of `HoldReason` in the job ClassAd. If this expression is `UNDEFINED` or produces an empty or invalid string, a default description is used.

periodic_hold_subcode = <ClassAd Integer Expression> When the job is placed on hold due to the **periodic_hold** expression becoming true, this expression is evaluated to set the value of `HoldReasonSubCode` in the job ClassAd. The default subcode is 0. The `HoldReasonCode` will be set to 3, which indicates that the job went on hold due to a job policy expression.

periodic_release = <ClassAd Boolean Expression> This expression is checked periodically when the job is in the Held state. If the expression becomes `True`, the job will be released.

Only job ClassAd attributes will be defined for use by this ClassAd expression. Note that, by default, this expression is only checked once every 60 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL`, `MAX_PERIODIC_EXPR_INTERVAL`, and `PERIODIC_EXPR_TIMESLICE` configuration macros.

periodic_remove = <ClassAd Boolean Expression> This expression is checked periodically. If it becomes `True`, the job is removed from the queue. If unspecified, the default value is `False`.

See the Examples section of this manual page for an example of a **periodic_remove** 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.

Only job ClassAd attributes will be defined for use by this ClassAd expression. Note that, by default, this expression is only checked once every 60 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL`, `MAX_PERIODIC_EXPR_INTERVAL`, and `PERIODIC_EXPR_TIMESLICE` configuration macros.

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, HTCondor 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>** HTCondor 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 MiB and the block size to 256 KiB 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, HTCondor 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, HTCondor will use a buffer of 512 KiB and a block size of 32 KiB.

compress_files = file1, file2, ... If your job attempts to access any of the files mentioned in this list, HTCondor 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, HTCondor 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 HTCondor 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 HTCondor 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 your 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 HTCondor 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, HTCondor 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

batch_queue = <queuename> Used for **pbs**, **lsf**, and **sge** grid universe jobs. Specifies the name of the PBS/LSF/SGE job queue into which the job should be submitted. If not specified, the default queue is used.

boinc_authenticator_file = <pathname> For grid type **boinc** jobs, specifies a path and file name of the authorization file that grants permission for HTCondor to use the BOINC service. There is no default value when not specified.

cream_attributes = <name=value;...;name=value> Provides a list of attribute/value pairs to be set in a CREAM job description of a grid universe job destined for the CREAM grid system. The pairs are separated by semicolons, and written in New ClassAd syntax.

delegate_job_GSI_credentials_lifetime = <seconds> Specifies the maximum number of seconds for which delegated proxies should be valid. The default behavior when this command is not specified is determined by the configuration variable `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`, which defaults to one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. This setting currently only applies to proxies delegated for non-grid jobs and for HTCondor-C jobs. It does not currently apply to globus grid jobs, which always behave as though this setting were 0. This variable has no effect if the configuration variable `DELEGATE_JOB_GSI_CREDENTIALS` is `False`, because in that case the job proxy is copied rather than delegated.

ec2_access_key_id = <pathname> For grid type **ec2** jobs, identifies the file containing the access key.

ec2_ami_id = <EC2 xMI ID> For grid type **ec2** jobs, identifies the machine image. Services compatible with the EC2 Query API may refer to these with abbreviations other than `AMI`, for example `EMI` is valid for Eucalyptus.

ec2_availability_zone = <zone name> For grid type **ec2** jobs, specifies the Availability Zone that the instance should be run in. This command is optional, unless **ec2_ebs_volumes** is set. As an example, one current zone is `us-east-1b`.

ec2_block_device_mapping = <block-device>:<kernel-device>,<block-device>:<kernel-device>, ... For grid type **ec2** jobs, specifies the block device to kernel device mapping. This command is optional.

ec2_ebs_volumes = <ebs name>:<device name>,<ebs name>:<device name>,... For grid type **ec2** jobs, optionally specifies a list of Elastic Block Store (EBS) volumes to be made available to the instance and the device names they should have in the instance.

ec2_elastic_ip = <elastic IP address> For grid type **ec2** jobs, and optional specification of an Elastic IP address that should be assigned to this instance.

ec2_iam_profile_arn = <IAM profile ARN> For grid type **ec2** jobs, an Amazon Resource Name (ARN) identifying which Identity and Access Management (IAM) (instance) profile to associate with the instance.

ec2_iam_profile_name= <IAM profile name> For grid type **ec2** jobs, a name identifying which Identity and Access Management (IAM) (instance) profile to associate with the instance.

ec2_instance_type = <instance type> For grid type **ec2** jobs, identifies the instance type. Different services may offer different instance types, so no default value is set.

ec2_keypair = <ssh key-pair name> For grid type **ec2** jobs, specifies the name of an SSH key-pair that is already registered with the EC2 service. The associated private key can be used to *ssh* into the virtual machine once it is running.

ec2_keypair_file = <pathname> For grid type **ec2** jobs, specifies the complete path and file name of a file into which HTCondor will write an SSH key for use with ec2 jobs. The key can be used to *ssh* into the virtual machine once it is running. If **ec2_keypair** is specified for a job, **ec2_keypair_file** is ignored.

ec2_parameter_names = ParameterName1, ParameterName2, ... For grid type **ec2** jobs, a space or comma separated list of the names of additional parameters to pass when instantiating an instance.

ec2_parameter_<name> = <value> For grid type **ec2** jobs, specifies the value for the correspondingly named (instance instantiation) parameter. **<name>** is the parameter name specified in the submit command **ec2_parameter_names**, but with any periods replaced by underscores.

ec2_secret_access_key = <pathname> For grid type **ec2** jobs, specifies the path and file name containing the secret access key.

ec2_security_groups = group1, group2, ... For grid type **ec2** jobs, defines the list of EC2 security groups which should be associated with the job.

ec2_security_ids = id1, id2, ... For grid type **ec2** jobs, defines the list of EC2 security group IDs which should be associated with the job.

ec2_spot_price = <bid> For grid type **ec2** jobs, specifies the spot instance bid, which is the most that the job submitter is willing to pay per hour to run this job.

ec2_tag_names = <name0,name1,name...> For grid type **ec2** jobs, specifies the case of tag names that will be associated with the running instance. This is only necessary if a tag name case matters. By default the list will be automatically generated.

ec2_tag_<name> = <value> For grid type **ec2** jobs, specifies a tag to be associated with the running instance. The tag name will be lower-cased, use **ec2_tag_names** to change the case.

WantNameTag = <True | False> For grid type **ec2** jobs, a job may request that its 'name' tag be (not) set by HTCondor. If the job does not otherwise specify any tags, not setting its name tag will eliminate a call by the EC2 GAHP, improving performance.

ec2_user_data = <data> For grid type **ec2** jobs, provides a block of data that can be accessed by the virtual machine. If both **ec2_user_data** and **ec2_user_data_file** are specified for a job, the two blocks of data are concatenated, with the data from this **ec2_user_data** submit command occurring first.

ec2_user_data_file = <pathname> For grid type **ec2** jobs, specifies a path and file name whose contents can be accessed by the virtual machine. If both **ec2_user_data** and **ec2_user_data_file** are specified for a job, the two blocks of data are concatenated, with the data from that **ec2_user_data** submit command occurring first.

ec2_vpc_ip = <a.b.c.d> For grid type **ec2** jobs, that are part of a Virtual Private Cloud (VPC), an optional specification of the IP address that this instance should have within the VPC.

ec2_vpc_subnet = <subnet specification string> For grid type **ec2** jobs, an optional specification of the Virtual Private Cloud (VPC) that this instance should be a part of.

gce_auth_file = <pathname> For grid type **gce** jobs, specifies a path and file name of the authorization file that grants permission for HTCondor to use the Google account. There is no default value when not specified.

gce_image = <image id> For grid type **gce** jobs, the identifier of the virtual machine image representing the HTCondor job to be run. This virtual machine image must already be register with GCE and reside in Google's Cloud Storage service.

gce_json_file = <pathname> For grid type **gce** jobs, specifies the path and file name of a file that contains JSON elements that should be added to the instance description submitted to the GCE service.

gce_machine_type = <machine type> For grid type **gce** jobs, the long form of the URL that describes the machine configuration that the virtual machine instance is to run on.

gce_metadata = <name=value,...,name=value> For grid type **gce** jobs, a comma separated list of name and value pairs that define metadata for a virtual machine instance that is an HTCondor job.

gce_metadata_file = <pathname> For grid type **gce** jobs, specifies a path and file name of the file that contains metadata for a virtual machine instance that is an HTCondor job. Within the file, each name and value pair is on its own line; so, the pairs are separated by the newline character.

gce_preemptible = <True | False> For grid type **gce** jobs, specifies whether the virtual machine instance should be preemptible. The default is for the instance to not be preemptible.

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 HTCondor-G
2. when a grid universe job is released from the hold state
3. when HTCondor-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**.

grid_resource = <grid-type-string> <grid-specific-parameter-list> For each **grid-type-string** value, there are further type-specific values that must specified. This submit description file command allows each to be given in a space-separated list. Allowable **grid-type-string** values are **batch**, **condor**, **cream**, **ec2**, **gt2**, **gt5**, **lsf**, **nordugrid**, **pbs**, **sge**, and **unicore**. The HTCondor manual chapter on Grid Computing details the variety of grid types.

For a **grid-type-string** of **batch**, the single parameter is the name of the local batch system, and will be one of `pbs`, `lsf`, or `sge`.

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.

For a **grid-type-string** of **cream**, there are three parameters. The first parameter is the web services address of the CREAM server. The second parameter is the name of the batch system that sits behind the CREAM server. The third parameter identifies a site-specific queue within the batch system.

For a **grid-type-string** of **ec2**, one additional parameter specifies the EC2 URL.

For a **grid-type-string** of **gt2**, the single parameter is the name of the pre-WS GRAM resource to be used.

For a **grid-type-string** of **gt5**, the single parameter is the name of the pre-WS GRAM resource to be used, which is the same as for the **grid-type-string** of **gt2**.

For a **grid-type-string** of **lsf**, no additional parameters are used.

For a **grid-type-string** of **nordugrid**, the single parameter is the name of the NorduGrid resource to be used.

For a **grid-type-string** of **pbs**, no additional parameters are used.

For a **grid-type-string** of **sge**, no additional parameters are used.

For a **grid-type-string** of **unicore**, the first parameter is the name of the Unicore Usite to be used. The second parameter is the name of the Unicore Vsite to be used.

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**.

use_x509userproxy = <True | False> Set this command to `True` to indicate that the job requires an X.509 user proxy. If **x509userproxy** is set, then that file is used for the proxy. Otherwise, the proxy is looked for in the standard locations. If **x509userproxy** is set or if the job is a grid universe job of grid type **gt2**, **gt5**, **cream**, or **nordugrid**, then the value of **use_x509userproxy** is forced to `True`. Defaults to `False`.

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 HTCondor to access the proxy in a shared file system (for example, AFS). HTCondor 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_PROXY. If that variable is not present, it will search in the default location. Note that proxies are only valid for a limited time. Condor_submit will not submit a job with an expired proxy, it will return an error. Also, if the configuration parameter CRED_MIN_TIME_LEFT is set to some number of seconds, and if the proxy will expire before that many seconds, condor_submit will also refuse to submit the job. That is, if CRED_MIN_TIME_LEFT is set to 60, condor_submit will refuse to submit a job whose proxy will expire 60 seconds from the time of submission.

x509userproxy is relevant when the **universe** is **vanilla**, or when the **universe** is **grid** and the type of grid system is one of **gt2**, **gt5**, **condor**, **cream**, or **nordugrid**. Defining a value causes the proxy to be delegated to the execute machine. Further, VOMS attributes defined in the proxy will appear in the job ClassAd.

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 HTCondor 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 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_disk = file1:device1:permission1, file2:device2:permission2:format2, . . . A list of comma separated disk files. Each disk file is specified by 4 colon separated fields. The first field is the path and file name of the disk file. The second field specifies the device. The third field specifies permissions, and the optional fourth field specifies the image format. If a disk file will be transferred by HTCondor, then the first field should just be the simple file name (no path information).

An example that specifies two disk files:

```
vm_disk = /myxen/diskfile.img:sda1:w,/myxen/swap.img:sda2:w
```

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 HTCondor 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_type = <vmware | xen | kvm> 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. This command is optional; when not specified, all relevant VMware image files are to be listed using **transfer_input_files**.

vmware_should_transfer_files = <True | False> Specifies whether HTCondor 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 HTCondor to utilize a VMware snapshot disk for new or modified files. If not specified, the default value is `True`.

xen_initrd = <image-file> When **xen_kernel** gives a file name for the kernel image to use, this optional command may specify a path to a ramdisk (`initrd`) image file. If the image file will be transferred by HTCondor, then the value should just be the simple file name (no path information).

xen_kernel = <included | path-to-kernel> A value of **included** specifies that the kernel is included in the disk file. If not one of these values, then the value is a path and file name of the kernel to be used. If a kernel file will be transferred by HTCondor, then the value should just be the simple file name (no path information).

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** gives a path to a kernel. Omission for this required case results in an error message during submission.

COMMANDS FOR THE DOCKER UNIVERSE

docker_image = < image-name > Defines the name of the Docker image that is the basis for the docker container.

ADVANCED COMMANDS

accounting_group = <accounting-group-name> Causes jobs to negotiate under the given accounting group. This value is advertised in the job ClassAd as `AcctGroup`. The HTCondor Administrator's manual contains more information about accounting groups.

accounting_group_user = <accounting-group-user-name> Sets the user name associated with the accounting group name for resource usage accounting purposes. If not set, defaults to the value of the job ClassAd attribute `Owner`. This value is advertised in the job ClassAd as `AcctGroupUser`. If an accounting group has not been set with the **accounting_group** command, this command is ignored.

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 is a string that identifies the limit, or it is a ClassAd expression that evaluates to a string, and it is evaluated in the context

of machine ClassAd being considered as a match. Each item in the list also 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. Details on concurrency limits are in the HTCondor Administrator's manual.

concurrency_limits_expr = <ClassAd String Expression> A ClassAd expression that represents the list of resources that this job needs after evaluation. The ClassAd expression may specify machine ClassAd attributes that are evaluated against a matched machine. After evaluation, the list sets **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 (note that `coredumpsize` is not an HTCondor parameter – it is an operating system parameter that can be viewed with the *limit* or *ulimit* command on Unix and in the Registry on Windows). A value of -1 results in no limits being applied to the core file size. If HTCondor is running as root, a **coresize** setting greater than the system `coredumpsize` limit will override the system setting; if HTCondor is *not* running as root, the system `coredumpsize` limit will override **coresize**.

cron_day_of_month = <Cron-evaluated Day> The set of days of the month for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_day_of_week = <Cron-evaluated Day> The set of days of the week for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_hour = <Cron-evaluated Hour> The set of hours of the day for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_minute = <Cron-evaluated Minute> The set of minutes within an hour for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

cron_month = <Cron-evaluated Month> The set of months within a year for which a deferral time applies. The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

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.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

dagman_log = <pathname> DAGMan inserts this command to specify an event log that it watches to maintain the state of the DAG. If the **log** command is not specified in the submit file, DAGMan uses the **log** command to specify the event log.

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.

The HTCondor User's manual section on Time Scheduling for Job Execution has 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.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

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.

The HTCondor User's manual section on Time Scheduling for Job Execution has further details.

description = <string> A string that sets the value of the job ClassAd attribute `JobDescription`. When set, tools which display the executable such as *condor_q* will instead use this string.

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> Advice to HTCondor specifying the maximum virtual image size to which the job will grow during its execution. HTCondor will then execute the job only on machines which have enough resources, (such as virtual memory), to support executing the job. If not specified, HTCondor will automatically make a (reasonably accurate) estimate about the job's size and adjust this estimate as the program runs. If specified and underestimated, the job may crash due to the inability to acquire more address space; for example, if `malloc()` fails. If the image size is overestimated, HTCondor may have difficulty finding machines which have the required resources. *size* is specified in KiB. For example, for an image size of 8 MiB, *size* should be 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 job event log, when a full path is not specified.

For vanilla 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 or grid 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_ad_information_attrs = <attribute-list> A comma-separated list of job ClassAd attribute names. The named attributes and their values are written to the job event log whenever any event is being written to the log. This implements the same thing as the configuration variable `EVENT_LOG_INFORMATION_ATTRS` (see page 224), but it applies to the job event log, instead of the system event log.

JobBatchName = <batch_name> Set the batch name for this submit. The batch name is displayed by *condor_q* **-batch**. It is intended for use by users to give meaningful names to their jobs and to influence how *condor_q* groups jobs for display. This value in a submit file can be overridden by specifying the **-batch-name** argument on the *condor_submit* command line.

job_lease_duration = <number-of-seconds> For vanilla, parallel, VM, and java universe jobs only, the duration in seconds of a job lease. The default value is 2,400, or forty minutes. If a job lease is not desired, the value can be explicitly set to 0 to disable the job lease semantics. The value can also be a ClassAd expression that evaluates to an integer. The HTCondor User's manual section on Special Environment Considerations has further details.

job_machine_attrs = <attr1, attr2, . . . > A comma and/or space separated list of machine attribute names that should be recorded in the job ClassAd in addition to the ones specified by the *condor_schedd* daemon's system configuration variable `SYSTEM_JOB_MACHINE_ATTRS`. When there are multiple run attempts, history of machine attributes from previous run attempts may be kept. The number of run attempts to store may be extended beyond the system-specified history length by using the submit file command **job_machine_attrs_history_length**. A machine attribute named `X` will be inserted into the job ClassAd as an attribute named `MachineAttrX0`. The previous value of this attribute will be named `MachineAttrX1`, the previous to that will be named `MachineAttrX2`, and so on, up to the specified history length. A history of length 1 means that only `MachineAttrX0` will be recorded. The value recorded in the job ClassAd is the evaluation of the machine attribute in the context of the job ClassAd when the *condor_schedd* daemon initiates the start up of the job. If the evaluation results in an `Undefined` or `Error` result, the value recorded in the job ad will be `Undefined` or `Error`, respectively.

want_graceful_removal = <boolean expression> When `True`, this causes a graceful shutdown of the job when the job is removed or put on hold, giving it time to clean up or save state. Otherwise, the job is abruptly killed. The default is `false`.

kill_sig = <signal-number> When HTCondor 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 HTCondor libraries to initiate a checkpoint of the process. For jobs submitted to other universes, the default value, when not defined, is `SIGTERM`, which is the standard way to terminate a program in Unix.

kill_sig_timeout = <seconds> This submit command should no longer be used as of HTCondor version 7.7.3; use **job_max_vacate_time** instead. If **job_max_vacate_time** is not defined, this defines the number of seconds that HTCondor should wait following the sending of the kill signal defined by **kill_sig** and forcibly killing the job. The actual amount of time between sending the signal and forcibly killing the job is the smallest of this value and the configuration variable `KILLING_TIMEOUT`, as defined on the execute machine.

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.

job_max_vacate_time = <integer expression> An integer-valued expression (in seconds) that may be used to adjust the time given to an evicted job for gracefully shutting down. If the job's setting is less than the machine's, the job's is used. If the job's setting is larger than the machine's, the result depends on whether the job has any excess retirement time. If the job has more retirement time left than the machine's max vacate time setting, then retirement time will be converted into vacating time, up to the amount requested by the job.

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 `MachineMaxVacateTime`, or to preferentially run on such machines, explicitly specify this in the requirements and/or rank expressions.

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. Retirement time is a grace period given to a job to finish when a resource claim is about to be preempted. 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 the job will be allowed to run until the retirement time expires, at which point it is hard-killed. The job will be soft-killed when it is getting close to the end of retirement in order to give it time to gracefully shut down. The amount of lead-time for soft-killing is determined by the maximum vacating time granted to the job.

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 `MaxJobRetirementTime`, 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 HTCondor, HTCondor decides which job to run based upon user and job priorities. Setting **nice_user** equal to `True` tells HTCondor 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 HTCondor jobs. Jobs submitted in this fashion have `"nice-user."` prepended to 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 HTCondor 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.

run_as_owner = <True | False> A boolean value that causes the job to be run under the login of the submitter, if supported by the joint configuration of the submit and execute machines. On Unix platforms, this defaults to `True`, and on Windows platforms, it defaults to `False`. May not be used with **load_profile**. See the HTCondor manual Platform-Specific Information chapter for administrative details on configuring Windows to support this option.

stack_size = <size in bytes> This command applies only to Linux platform jobs that are not standard universe jobs. An integer number of bytes, representing the amount of stack space to be allocated for the job. This value replaces the default allocation of stack space, which is unlimited in size.

submit_event_notes = <note> A string that is appended to the submit event in the job's log file. For DAGMan jobs, the string `DAG Node:` and the node's name is automatically defined for **submit_event_notes**, causing the logged submit event to identify the DAG node job submitted.

+<attribute> = <value> A line that begins with a '+' (plus) character instructs *condor_submit* to insert the given *attribute* into the job ClassAd with the given *value*. Note that setting an attribute should not be used in place of one of the specific commands listed above. Often, the command name does not directly correspond to an attribute name; furthermore, many submit commands result in actions more complex than simply setting an attribute or attributes. See 987 for a list of HTCondor job attributes.

PRE AND POST SCRIPTS IMPLEMENTED WITH SPECIALLY-NAMED ATTRIBUTES

+PreCmd = "<executable>" A vanilla universe job may specify that a script is to be run on the execute machine before the job, and this is called a prescript. Definition of this specifically-named attribute causes the script, identified by path and file name, to be executed. The prescript could prepare or initialize the job. Note that this definition of a prescript is different from the PRE script described in DAGMan. The prescript is not automatically transferred with the job, as the main executable is, so it must be entered into the **transfer_input_files** list, when file transfer is enabled.

+PreArgs = "<argument_list>" Defines command line arguments for the prescript, presuming the Old argument syntax.

+PreArguments = "<argument_list>" Defines command line arguments for the prescript, presuming the New argument syntax. An exception to the syntax is that double quotes must be escaped with a backslash instead of another double quote.

Note that if both +PreArgs and +PreArguments are specified, the +PreArguments value is used and the +PreArgs value is ignored.

+PreEnv = "<environment_vars>" Defines the environment for the prescript, presuming the Old environment syntax.

+PreEnvironment = "<environment_vars>" Defines the environment for the prescript, presuming the New environment syntax.

Note that if both +PreEnv and +PreEnvironment are specified, the +PreEnvironment value is used and the +PreEnv value is ignored.

+PostCmd = "<executable>" A vanilla universe job may specify that a script is to be run on the execute machine after the job exits, and this is called a postscript. Definition of this specifically-named attribute causes the script, identified by path and file name, to be executed. The postscript is run if the job exits, but not if the job is evicted. Note that this definition of a postscript is different from the POST script described in DAGMan. The postscript is not automatically transferred with the job, as the main executable is, so it must be entered into the **transfer_input_files** list, when file transfer is enabled.

+PostArgs = "<argument_list>" Defines command line arguments for the postscript, presuming the Old argument syntax.

+PostArguments = "<argument_list>" Defines command line arguments for the postscript, presuming the New argument syntax. An exception to the syntax is that double quotes must be escaped with a backslash instead of another double quote mark.

Note that if both +PostArgs and +PostArguments are specified, the +PostArguments value is used and the +PostArgs value is ignored.

+PostEnv = "<environment_vars>" Defines the environment for the postscript, presuming the Old environment syntax.

+PostEnvironment = "<environment_vars>" Defines the environment for the postscript, presuming the New environment syntax.

Note that if both +PostEnv and +PostEnvironment are specified, the +PostEnvironment value is used and the +PostEnv value is ignored.

If any of the prescript or postscript values are not enclosed in double quotes, they are silently ignored.

Below is an example of the use of starter pre and post scripts:

```
+PreCmd = "my_pre"
+PreArgs = "pre\"arg1 prea'rg2"
+PreEnv = "one=1;two=\"2\""
```

```
+PostCmd = "my_post"
+PostArguments = "post\"arg1 'post''ar g2'"
+PostEnvironment = "one=1 two=\"2\""
```

For this example `PreArgs` generates a first argument of `pre"a1"` and a second argument of `pre'a2`. `PostArguments` generates a first argument of `post"a1` and a second argument of `post'a 2`.

MACROS AND COMMENTS

In addition to commands, the submit description file can contain macros and comments.

Macros Parameterless macros in the form of `$(macro_name:default initial value)` may be used anywhere in HTCondor submit description files to provide textual substitution at submit time. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Two pre-defined macros are supplied by the submit description file parser. The `$(Cluster)` or `$(ClusterId)` macro supplies the value of the `ClusterId` job ClassAd attribute, and the `$(Process)` or `$(ProcId)` 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 an HTCondor process with its own cluster and process numbers on the command line.

The `$(Node)` macro is defined for parallel universe jobs, and is especially relevant for MPI applications. It is a unique value assigned for the duration of the job that essentially identifies the machine (slot) on which a program is executing. Values assigned start at 0 and increase monotonically. The values are assigned as the parallel job is about to start.

Recursive definition of macros is permitted. An example of a construction that works is the following:

```
foo = bar
foo =  snap $(foo)
```

As a result, `foo = snap bar`.

Note that both left- and right- recursion works, so

```
foo = bar
foo =  $(foo) snap
```

has as its result `foo = bar snap`.

The construction

```
foo = $(foo) bar
```

by itself will *not* work, as it does not have an initial base case. Mutually recursive constructions such as:

```
B = bar
C = $(B)
B = $(C) boo
```

will *not* work, and will fill memory with expansions.

A default value may be specified, for use if the macro has no definition. Consider the example

```
D = $(E:24)
```

Where `E` is not defined within the submit description file, the default value 24 is used, resulting in

```
D = 24
```

This is of limited value, as the scope of macro substitution is the submit description file. Thus, either the macro is or is not defined within the submit description file. If the macro is defined, then the default value is useless. If the macro is not defined, then there is no point in using it in a submit command.

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 machine 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)
```

As this form of the substitution macro is only evaluated within the context of the machine ClassAd, use of a scope resolution prefix `TARGET.` or `MY.` is not allowed.

A common use of this form of the substitution 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 example allows HTCondor 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.

```
arguments = $$(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. In this form, the expression may refer to machine attributes by prefacing them with the `TARGET.` scope resolution prefix. 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.

Submit Variables

While processing the **queue** command in a submit file or from the command line, *condor_submit* will set the values of several automatic submit variables so that they can be referred to by statements in the submit file. With the exception of Cluster and Process, if these variables are set by the submit file, they will not be modified during **queue** processing.

ClusterId Set to the integer value that the `ClusterId` attribute that the job ClassAd will have when the job is submitted. All jobs in a single submit will normally have the same value for the `ClusterId`. If the **-dry-run** argument is specified, The value will be 1.

Cluster Alternate name for the ClusterId submit variable. Before HTCondor version 8.4 this was the only name.

ProcId Set to the integer value that the `ProcId` attribute of the job ClassAd will have when the job is submitted. The value will start at 0 and increment by 1 for each job submitted.

Process Alternate name for the ProcId submit variable. Before HTCondor version 8.4 this was the only name.

Node For parallel universes, set to the value #pArAlLeLnOdE# or #MpInOdE# depending on the parallel universe type For other universes it is set to nothing.

Step Set to the step value as it varies from 0 to N-1 where N is the number provided on the **queue** argument. This variable changes at the same rate as ProcId when it changes at all. For submit files that don't make use of the queue number option, Step will always be 0. For submit files that don't make use of any of the foreach options, Step and ProcId will always be the same.

ItemIndex Set to the index within the item list being processed by the various queue foreach options. For submit files that don't make use of any queue foreach list, ItemIndex will always be 0 For submit files that make use of a slice to select only some items in a foreach list, ItemIndex will only be set to selected values.

Row Alternate name for ItemIndex.

Item when a queue foreach option is used and no variable list is supplied, this variable will be set to the value of the current item.

The automatic variables below are set before parsing the submit file, and will not vary during processing unless the submit file itself sets them.

ARCH Set to the CPU architecture of the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYS Set to the name of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYSANDVER Set to the name and major version of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYSMAJORVER Set to the major version of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

OPSYSVER Set to the version of the operating system on the machine running *condor_submit*. The value will be the same as the automatic configuration variable of the same name.

SPOOL Set to the full path of the HTCondor spool directory. The value will be the same as the automatic configuration variable of the same name.

IsLinux Set to true if the operating system of the machine running *condor_submit* is a Linux variant. Set to false otherwise.

IsWindows Set to true if the operating system of the machine running *condor_submit* is a Microsoft Windows variant. Set to false otherwise.

SUBMIT_FILE Set to the full pathname of the submit file being processed by *condor_submit*. If submit statements are read from standard input, it is set to nothing.

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 HTCondor. 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       = vanilla

Arguments      = 15 2000
Output  = foo.out0
Error   = foo.err0
Queue

Arguments      = 30 2000
Output  = foo.out1
Error   = foo.err1
Queue

Arguments      = 45 6000
Output  = foo.out2
Error   = foo.err2
Queue
```

  Or you can get the same results as the above submit file by using a list of arguments with the Queue statement

```
####################
#
# submit description file
# Example 1b: queuing multiple jobs with differing
# command line arguments and output files, alternate syntax
#
####################

Executable     = foo
Universe       = vanilla

# generate different output and error filenames for each process
Output  = foo.out$(Process)
Error   = foo.err$(Process)

Queue Arguments From (
  15 2000
  30 2000
  45 6000
)
```

- Submit Description File Example 2: This submit description file example queues 150 runs of program *foo* which must have been compiled and linked for an Intel x86 processor running RHEL 3. HTCondor 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 HTCondor 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   = OpSys == "LINUX" && Arch =="INTEL"
Rank           = Memory >= 64
Request_Memory = 32 Mb
Image_Size     = 28 Mb

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log
Queue 150
```

- Submit Description File Example 3: This example targets the */bin/sleep* program to run only on a platform running a RHEL 6 operating system. The example presumes that the pool contains machines running more than one version of Linux, and this job needs the particular operating system to run correctly.

```
####################
#
# Example 3: Run on a RedHat 6 machine
#
####################
Universe     = vanilla
Executable   = /bin/sleep
Arguments    = 30
Requirements = (OpSysAndVer == "RedHat6")

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log     = sleep.log
Queue
```

- 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, HTCondor 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 HTCondor jobs to write to the same file, and/or if the user alters any files that need to be accessed by an HTCondor job which is still in the queue. For example, the compressing of data or output files before an HTCondor 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

HTCondor User Manual

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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** *NumberOfProcs*] [**-maxjobs** *NumberOfClusters*] [**-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*] [**-batch-name** *batch_name*] [**-autorescue** *0|1*] [**-dorescuefrom** *number*] [**-allowversionmismatch**] [**-no_recurse**] [**-do_recurse**] [**-update_submit**] [**-import_env**] [**-DumpRescue**] [**-valgrind**] [**-DontAlwaysRunPost**] [**-AlwaysRunPost**] [**-priority** *number*] [**-dont_use_default_node_log**] [**-schedd-daemon-ad-file** *FileName*] [**-schedd-address-file** *FileName*] [**-suppress_notification**] [**-dont_suppress_notification**] [**-DoRecovery**] *DAGInputFile1* [*DAGInputFile2 . . . DAGInputFileN*]

Description

condor_submit_dag is the program for submitting a DAG (directed acyclic graph) of jobs for execution under HTCondor. The program enforces the job dependencies defined in one or more *DAGInputFile*s. Each *DAGInputFile* contains commands to direct the submission of jobs implied by the nodes of a DAG to HTCondor. Extensive documentation is in the HTCondor User Manual section on DAGMan.

Some options may be specified on the command line or in the configuration or in a node job's submit description file. Precedence is given to command line options or configuration over settings from a submit description file. An example is e-mail notifications. When configuration variable `DAGMAN_SUPPRESS_NOTIFICATION` is its default value of `True`, and a node job's submit description file contains

```
notification = Complete
```

e-mail will *not* be sent upon completion, as the value of `DAGMAN_SUPPRESS_NOTIFICATION` is enforced.

Options

-help Display usage information and exit.

-version Display version information and exit.

-no_submit Produce the HTCondor submit description file for DAGMan, but do not submit DAGMan as an HTCondor 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 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 is in effect, any existing old-style rescue DAGs will be deleted, and the original DAG will be run.

-maxidle ***NumberOfProcs*** Sets the maximum number of idle procs allowed before *condor_dagman* stops submitting more node jobs. Note that for this argument, each individual proc within a cluster counts as a towards the limit, which is inconsistent with **-maxjobs** . Once idle procs start to run, *condor_dagman* will resume submitting jobs once the number of idle procs falls below the specified limit. *NumberOfProcs* is a non-negative integer. If this option is omitted, the number of idle procs is limited by the configuration variable `DAGMAN_MAX_JOBS_IDLE` (see 3.5.23), which defaults to 1000. To disable this limit, set *NumberOfProcs* to 0. Note that submit description files that queue multiple procs can cause the *NumberOfProcs* limit to be exceeded. Setting `queue 5000` in the submit description file, where *-maxidle* is set to 250 will result in a cluster of 5000 new procs being submitted to the *condor_schedd*, not 250. In this case, *condor_dagman* will resume submitting jobs when the number of idle procs falls below 250.

-maxjobs ***NumberOfClusters*** Sets the maximum number of clusters within the DAG that will be submitted to HTCondor at one time. Note that for this argument, each cluster counts as one job, no matter how many individual procs are in the cluster. *NumberOfClusters* is a non-negative integer. If this option is omitted, the number of clusters is limited by the configuration variable `DAGMAN_MAX_JOBS_SUBMITTED` (see 3.5.23), which defaults to 0 (unlimited).

-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 non-negative integer. If this option is omitted, the number of PRE scripts is limited by the configuration variable `DAGMAN_MAX_PRE_SCRIPTS` (see 3.5.23), which defaults to 20.

-maxpost ***NumberOfPostScripts*** Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPostScripts* is a non-negative integer. If this option is omitted, the number of POST scripts is limited by the configuration variable `DAGMAN_MAX_POST_SCRIPTS` (see 3.5.23), which defaults to 20.

-notification ***value*** Sets the e-mail notification for DAGMan itself. This information will be used within the HTCondor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See the description of

notification within *condor_submit* manual page for a 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 variable.

-allowlogerror As of verson 8.5.5 this argument is no longer supported, and setting it will generate a warning.

-r *schedd_name* Submit *condor_dagman* to a remote machine, specifically the *condor_schedd* daemon on that machine. The *condor_dagman* job will not run on the local *condor_schedd* (the submit machine), but on the specified one. This is implemented using the **-remote** option to *condor_submit*. Note that this option does not currently specify input files for *condor_dagman*, nor the individual nodes to be taken along! It is assumed that any necessary files will be present on the remote computer, possibly via a shared file system between the local computer and the remote computer. It is also necessary that the user has appropriate permissions to submit a job to the remote machine; the permissions are the same as those required to use *condor_submit*'s **-remote** option. If other options are desired, including transfer of other input files, consider using the **-no_submit** option, modifying the resulting submit file for specific needs, and then using *condor_submit* on that.

-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. See the *condor_dagman* manual page for detailed descriptions of these values. If not specified, no **-debug** *v*alue is passed to *condor_dagman*.

-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.

-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.

-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.

-batch-name *batch_name* Set the batch name for this DAG/workflow. The batch name is displayed by *condor_q* **-batch**. It is intended for use by users to give meaningful names to their workflows and to influence how *condor_q* groups jobs for display. As of version 8.5.5, the batch name set with this argument is propagated to all node jobs of the given DAG (including sub-DAGs), overriding any batch names set in the individual submit files. Note: set the batch name to ' ' (space) to avoid overriding batch names specified in node job submit files. If no batch name is set, the batch name defaults to *DagFile+cluster* (where *DagFile* is the primary DAG file of the top-level DAGMan, and *cluster* is the HTCondor cluster of the top-level DAGMan); the default *will* override any lower-level batch names.

-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. (Note that, starting with version 7.4.0, *condor_dagman* no longer requires an exact version match between itself and the `.condor.sub` file. Instead, a "minimum compatible version" is defined, and any `.condor.sub` file of that version or newer is accepted.)

-no_recurse This optional argument causes *condor_submit_dag* to *not* run itself recursively on nested DAGs (this is now the default; this flag has been kept mainly for backwards compatibility).

-do_recurse This optional argument causes *condor_submit_dag* to run itself recursively on nested DAGs. The default is now that it does *not* run itself recursively; instead the `.condor.sub` files for nested DAGs are generated "lazily" by *condor_dagman* itself. DAG nodes specified with the **SUBDAG EXTERNAL** keyword or with submit file names ending in `.condor.sub` are considered nested DAGs. The `DAGMAN_GENERATE_SUBDAG_SUBMITS` configuration variable may be relevant.

-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.

-import_env This optional argument causes *condor_submit_dag* to import the current environment into the **environment** command of the `.condor.sub` file it generates.

-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. The Rescue DAG file is produced whether or not there are parse errors reading the original DAG input file. The name of the file differs if there was a parse error.

-valgrind This optional argument causes the submit description file generated for the submission of *condor_dagman* to be modified. The executable becomes *valgrind* run on *condor_dagman*, with a specific set of arguments intended for testing *condor_dagman*. Note that this argument is intended for testing purposes only. Using the **-valgrind** option without the necessary *valgrind* software installed will cause the DAG to fail. If the DAG does run, it will run much more slowly than usual.

-DontAlwaysRunPost This option causes the submit description file generated for the submission of *condor_dagman* to be modified. It causes *condor_dagman* to not run the POST script of a node if the PRE script fails. (This was the default behavior prior to HTCondor version 7.7.2, and is again the default behavior from version 8.5.4 onwards.)

-AlwaysRunPost This option causes the submit description file generated for the submission of *condor_dagman* to be modified. It causes *condor_dagman* to always run the POST script of a node, even if the PRE script fails. (This was the default behavior for HTCondor version 7.7.2 through version 8.5.3.)

-priority *number* Sets the minimum job priority of node jobs submitted and running under the *condor_dagman* job submitted by this *condor_submit_dag* command.

-dont_use_default_node_log This option is disabled as of HTCondor version 8.3.1. This causes a compatibility error if the HTCondor version number of the *condor_schedd* is 7.9.0 or older. Tells *condor_dagman* to use the file specified by the job ClassAd attribute `UserLog` to monitor job status. If this command line argument is used, then the job event log file cannot be defined with a macro.

-schedd-daemon-ad-file *FileName* Specifies a full path to a daemon ad file dropped by a *condor_schedd*. Therefore this allows submission to a specific scheduler if several are available without repeatedly querying the *condor_collector*. The value for this argument defaults to the configuration attribute `SCHEDD_DAEMON_AD_FILE`.

-schedd-address-file *FileName* Specifies a full path to an address file dropped by a *condor_schedd*. Therefore this allows submission to a specific scheduler if several are available without repeatedly querying the *condor_collector*. The value for this argument defaults to the configuration attribute `SCHEDD_ADDRESS_FILE`.

-suppress_notification Causes jobs submitted by *condor_dagman* to not send email notification for events. The same effect can be achieved by setting configuration variable `DAGMAN_SUPPRESS_NOTIFICATION` to `True`. This command line option is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself.

-dont_suppress_notification Causes jobs submitted by *condor_dagman* to defer to content within the submit description file when deciding to send email notification for events. The same effect can be achieved by setting configuration variable `DAGMAN_SUPPRESS_NOTIFICATION` to `False`. This command line flag is independent of the **-notification** command line option, which controls notification for the *condor_dagman* job itself. If both **-dont_suppress_notification** and **-suppress_notification** are specified with the same command line, the last argument is used.

-DoRecovery Causes *condor_dagman* to start in recovery mode. (This means that it reads the relevant job user log(s) and "catches up" to the given DAG's previous state before submitting any new jobs.)

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_suspend

suspend jobs from the HTCondor queue

Synopsis

condor_suspend [**-help** | **-version**]

condor_suspend [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname*]| [**-addr** *"<a.b.c.d:port>"*] *cluster* | *cluster.process* | *user* | ***-constraint*** *expression* | *-all*

Description

condor_suspend suspends one or more jobs from the HTCondor job queue. When a job is suspended, the match between the *condor_schedd* and machine is not been broken, such that the claim is still valid. But, the job is not making any progress and HTCondor is no longer generating a load on the machine. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The job(s) to be suspended are identified by one of the 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 suspend 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 Suspend all jobs in the specified cluster

cluster.process Suspend the specific job in the cluster

user Suspend jobs belonging to specified user

-constraint *expression* Suspend all jobs which match the job ClassAd expression constraint

-all Suspend all the jobs in the queue

Exit Status

condor_suspend will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

To suspend all jobs except for a specific user:

```
% condor_suspend -constraint 'Owner =!= "foo"'
```

Run *condor_continue* to continue execution.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_tail

Display the last contents of a running job's standard output or file

Synopsis

condor_tail [**-help**] | [**-version**]

condor_tail [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *name*] [**-debug**] [**-maxbytes** *numbytes*] [**-auto-retry**] [**-follow**] [**-no-stdout**] [**-stderr**] *job-ID* [*filename1*] [*filename2* . . .]

Description

condor_tail displays the last bytes of a file in the sandbox of a running job identified by the command line argument *job-ID*. `stdout` is tailed by default. The number of bytes displayed is limited to 1024, unless changed by specifying the **-maxbytes** option. This limit is applied for each individual tail of a file; for example, when following a file, the limit is applied each subsequent time output is obtained.

Options

-help Display usage information and exit.

-version Display version information and exit.

-pool *centralmanagerhostname[:portnumber]* Specify a pool by giving the central manager's host name and an optional port number.

-name *name* Query the *condor_schedd* daemon identified with *name*.

-debug Display extra debugging information.

-maxbytes *numbytes* Limits the maximum number of bytes transferred per tail access. If not specified, the maximum number of bytes is 1024.

-auto-retry Retry the tail of the file(s) every 2 seconds, if the job is not yet running.

-follow Repetitively tail the file(s), until interrupted.

-no-stdout Do not tail `stdout`.

-stderr Tail `stderr` instead of `stdout`.

Exit Status

The exit status of *condor_tail* is zero on success.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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 HTCondor 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 HTCondor to spool all input files, the job event 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_transform_ads

Transform ClassAds according to specified rules, and output the transformed ClassAds.

Synopsis

condor_transform_ads [**-help [rules]**]

condor_transform_ads **-rules** *rules-file* [**-in[:<form>]** *infile*] [**-out[:<form>[, nosort]]** *outfile*] [<*key*>=<*value*>] [**-long**] [**-json**] [**-xml**] [**-verbose**] [**-terse**] [**-debug**] [**-unit-test**] [**-testing**] [**-convertoldroutes**] [*infile1* ... *infileN*]

Note that exactly one rules file, and at least one input file, must be specified. If no output file is specified, output will be written to `stdout`.

Description

condor_transform_ads reads ClassAds from a set of input files, transforms them according to rules defined in a rules file, and outputs the resulting transformed ClassAds.

See https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=TjsAdTransformLanguage for a description of the transform language.

Options

-help [rules] Display usage information and exit. **-help rules** displays information about the available transformation rules.

-rules *rules-file* Specifies the file containing definitions of the transformation rules.

-in[:<form>] *infile* Specifies an input file containing ClassAd(s) to be transformed. **<form>**, if specified, is one of:

- **long**: traditional long form (default)
- **xml**: XML form
- **json**: JSON ClassAd form
- **new**: "new" ClassAd form without newlines
- **auto**: guess format by reading the input

If − is specified for *infile*, input is read from `stdin`.

-out[:<form>[, nosort] *outfile* Specifies an output file to receive the transformed ClassAd(s). **<form>**, if specified, is one of:

- **long**: traditional long form (default)
- **xml**: XML form
- **json**: JSON ClassAd form
- **new**: "new" ClassAd form without newlines
- **auto**: use the same format as the first input

ClassAds are storted by attribute unless **nosort** is specified.

[*<key>=<value>*] Assign key/value pairs before rules file is parsed; can be used to pass arguments to rules. (More detail needed here.)

-long Use long form for both input and output ClassAd(s). (This is the default.)

-json Use JSON form for both input and output ClassAd(s).

-xml Use XML form for both input and output ClassAd(s).

-verbose Verbose mode, echo transform rules as they are executed.

-terse Disable the **-verbose** option.

-debug More information needed here.

-unit-test More information needed here.

-testing More information needed here.

-convertoldroutes More information needed here.

Exit Status

condor_transform_ads will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Examples

Here's a simple example that transforms the given input ClassAds according to the given rules:

```
# File: my_input
ResidentSetSize = 500
DiskUsage = 2500000
NumCkpts = 0
TransferrErr = false
Err = "/dev/null"

# File: my_rules
EVALSET MemoryUsage ( ResidentSetSize / 100 )
EVALMACRO WantDisk = ( DiskUsage * 2 )
SET RequestDisk ( $(WantDisk) / 1024 )
RENAME NumCkpts NumCheckPoints
DELETE /(.+)Err/

# Command:
condor_transform_ads -rules my_rules -in my_input

# Output:
DiskUsage = 2500000
Err = "/dev/null"
MemoryUsage = 5
NumCheckPoints = 0
RequestDisk = ( 5000000 / 1024 )
ResidentSetSize = 500
```

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_update_machine_ad

update a machine ClassAd

Synopsis

condor_update_machine_ad [**-help** | **-version**]

condor_update_machine_ad [**-pool** *centralmanagerhostname[:portnumber]*] [**-name** *startdname*] *path/to/update-ad*

Description

condor_update_machine_ad modifies the specified *condor_startd* daemon's machine ClassAd. The ClassAd in the file given by `path/to/update-ad` represents the changed attributes. The changes persists until the *condor_startd* restarts. If no file is specified on the command line, *condor_update_machine_ad* reads the update ClassAd from `stdin`.

Contents of the file or `stdin` must contain a complete ClassAd. Each line *must* be terminated by a newline character, including the last line of the file. Lines are of the form

```
<attribute> = <value>
```

Changes to certain ClassAd attributes will cause the *condor_startd* to regenerate values for other ClassAd attributes. An example of this is setting `HasVM`. This will cause `OfflineUniverses`, `VMOfflineTime`, and `VMOfflineReason` to change.

Options

-help Display usage information and exit

-version Display the HTCondor version and exit

-pool ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

-name ***startdname*** Send the command to a machine identified by *startdname*

General Remarks

This tool is intended for the use of system administrators when dealing with offline universes.

Examples

To re-enable matching with the VM universe jobs, place on `stdin` a complete ClassAd (including the ending newline character) to change the value of ClassAd attribute `HasVM`:

```
echo "HasVM = True
" | condor_update_machine_ad
```

To prevent vm universe jobs from matching with the machine:

```
echo "HasVM = False
" | condor_update_machine_ad
```

To prevent vm universe jobs from matching with the machine and specify a reason:

```
echo "HasVM = False
VMOfflineReason = \"Cosmic rays.\"
" | condor_update_machine_ad
```

Note that the quotes around the reason are required by ClassAds, and they must be escaped because of the shell. Using a file instead of `stdin` may be preferable in these situations, because neither quoting nor escape characters are needed.

Exit Status

condor_update_machine_ad will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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 by configuration variable `COLLECTOR_DAEMON_HISTORY_SIZE`.

Options

—help Display usage information and exit.

-h Same as **—help**.

—version Display HTCondor 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 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 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 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_urlfetch

fetch configuration given a URL

Synopsis

condor_urlfetch [**-<daemon>**] *url local-url-cache-file*

Description

Depending on the command line arguments, *condor_urlfetch* sends the result of a query from the *url* to both standard output and to a file specified by *local-url-cache-file*, or it sends the contents of the file specified by *local-url-cache-file* to standard output.

condor_urlfetch is intended to be used as the program to run when defining configuration, such as in the nonfunctional example:

```
LOCAL_CONFIG_FILE = $(LIBEXEC)/condor_urlfetch -$(SUBSYSTEM) \
  http://www.example.com/htcondor-baseconfig  local.config |
```

The pipe character (`|`) at the end of this definition of the location of a configuration file changes the use of the definition. It causes the command listed on the right hand side of this assignment statement to be invoked, and standard output becomes the configuration. The value of `$(SUBSYSTEM)` becomes the daemon that caused this configuration to be read. If `$(SUBSYSTEM)` evaluates to `MASTER`, then the URL query always occurs, and the result is sent to standard output as well as written to the file specified by argument *local-url-cache-file*. When `$(SUBSYSTEM)` evaluates to a daemon other than `MASTER`, then the URL query only occurs if the file specified by *local-url-cache-file* does *not* exist. If the file specified by *local-url-cache-file* does exist, then the contents of this file is sent to standard output.

Note that if the configuration kept at the URL site changes, and reconfiguration is requested, the **-<daemon>** argument needs to be `-MASTER`. This is the only way to guarantee that there will be a query of the changed URL contents, such that they will make their way into the configuration.

Options

-<daemon> The upper case name of the daemon issuing the request for the configuration output. If `-MASTER`, then the URL query always occurs. If a daemon other than `-MASTER`, for example `STARTD` or `SCHEDD`, then the URL query only occurs if the file defined by *local-url-cache-file* does not exist.

Exit Status

condor_urlfetch will exit with a status value of 0 (zero) upon success and non zero otherwise.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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 *condor_submit* manual page for instructions for specifying that HTCondor 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 HTCondor 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 HTCondor 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 HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_userprio

Manage user priorities

Synopsis

***condor_userprio* -help**

condor_userprio [**-pool** *centralmanagerhostname[:portnumber]*] [**Edit option**] | [**Display options**] [**-inputfile** *filename*]

Description

condor_userprio either modifies priority-related information or displays priority-related information. Displayed information comes from the accountant log, where the *condor_negotiator* daemon stores historical usage information in the file at `$(SPOOL)/Accountantnew.log`. Which fields are displayed changes based on command line arguments. *condor_userprio* with no arguments, lists the active users along with their priorities, in increasing priority order. The **-all** option can be used to display more detailed information about each user, resulting in a rather wide display, and 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.

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.

Last Usage Time The most recent time a resource usage has been recorded for the user.

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 **-pool** option can be used to contact a different central manager than the local one (the default).

For security purposes of authentication and authorization, specifying an Edit Option requires the ADMINISTRATOR level of access.

Options

-help Display usage information and exit.

-pool ***centralmanagerhostname[:portnumber]*** Contact the 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 optional port) specified refer to the host name (and port) of the *condor_negotiator* to query for user priorities. This is slightly different than most HTCondor tools that support a **-pool** option, and instead expect the host name (and port) of the *condor_collector*.

-inputfile ***filename*** Introduced for debugging purposes, read priority information from *filename*. The contents of *filename* are expected to be the same as captured output from running a `condor_userprio -long` command.

-delete ***username*** (Edit option) Remove the specified *username* from HTCondor's accounting.

-resetall (Edit option) Reset the accumulated usage of all the users to zero.

-resetusage ***username*** (Edit option) Reset the accumulated usage of the user specified by *username* to zero.

-setaccum ***username value*** (Edit option) Set the accumulated usage of the user specified by *username* to the specified floating point *value*.

-setbegin ***username value*** (Edit option) Set the begin usage time of the user specified by *username* to the specified *value*.

-setfactor ***username value*** (Edit option) Set the priority factor of the user specified by *username* to the specified *value*.

-setlast ***username value*** (Edit option) Set the last usage time of the user specified by *username* to the specified *value*.

-setprio ***username value*** (Edit option) Set the real priority of the user specified by *username* to the specified *value*.

-activefrom ***month day year*** (Display option) Display information for users who have some recorded accumulated usage since the specified date.

-all (Display option) Display all available fields about each group or user.

-allusers (Display option) Display information for all the users who have some recorded accumulated usage.

-negotiator (Display option) Force the query to come from the negotiator instead of the collector.

-autoformat[:jlhVr,tng] ***attr1 [attr2 ...]*** **or -af[:jlhVr,tng]** ***attr1 [attr2 ...]*** (Display option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

j print the job ID as the first field,

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may *not* be used together. The **l** and **h** characters may *not* be used together.

-constraint <*expr*> (Display option) To be used in conjunction with the **-long -modular** or the **-autoformat** options. Displays users and groups that match the `<expr>`.

-debug[:<opts>] (Display option) Without **:<opts>** specified, use configured debug level to send debugging output to `stderr`. With **:<opts>** specified, these options are debug levels that override any configured debug levels for this command's execution to send debugging output to `stderr`.

-flat (Display option) Display information such that users within hierarchical groups are *not* listed with their group.

-getreslist ***username*** (Display option) Display all the resources currently allocated to the user specified by *username*.

-grouporder (Display option) Display submitter information with accounting group entries at the top of the list, and in breadth-first order within the group hierarchy tree.

-grouprollup (Display option) For hierarchical groups, the display shows sums as computed for groups, and these sums include sub groups.

-hierarchical (Display option) Display information such that users within hierarchical groups are listed with their group.

-legacy (Display option) For use with the **-long** option, displays attribute names and values as a single ClassAd.

-long (Display option) A verbose output which displays entire ClassAds.

-modular (Display option) Modifies the display when using the **-long** option, such that attribute names and values are shown as distinct ClassAds.

-most (Display option) Display fields considered to be the most useful. This is the default set of fields displayed.

-priority (Display option) Display fields with user priority information.

-quotas (Display option) Display fields relevant to hierarchical group quotas.

-usage (Display option) Display usage information for each group or user.

Examples

Example 1 Since the output varies due to command line arguments, here is an example of the default output for a pool that does not use Hierarchical Group Quotas. This default output is the same as given with the **-most** Display option.

```
Last Priority Update:  1/19 13:14
                      Effective   Priority   Res   Total Usage  Time Since
User Name              Priority    Factor   In Use (wghted-hrs) Last Usage
---------------------- ---------- --------- ------ ------------ ----------
www-cndr@cs.wisc.edu         0.56      1.00      0    591998.44    0+16:30
joey@cs.wisc.edu             1.00      1.00      1       990.15 <now>
suzy@cs.wisc.edu             1.53      1.00      0       261.78    0+09:31
leon@cs.wisc.edu             1.63      1.00      2     12597.82 <now>
raj@cs.wisc.edu              3.34      1.00      0      8049.48    0+01:39
jose@cs.wisc.edu             3.62      1.00      4     58137.63 <now>
betsy@cs.wisc.edu           13.47      1.00      0      1475.31    0+22:46
petra@cs.wisc.edu          266.02    500.00      1    288082.03 <now>
carmen@cs.wisc.edu         329.87     10.00    634   2685305.25 <now>
carlos@cs.wisc.edu         687.36     10.00      0     76555.13    0+14:31
ali@proj1.wisc.edu        5000.00  10000.00      0      1315.56    0+03:33
apu@nnland.edu            5000.00  10000.00      0       482.63    0+09:56
```

```
pop@proj1.wisc.edu         26688.11  10000.00      1     49560.88 <now>
franz@cs.wisc.edu          29352.06    500.00    109    600277.88 <now>
martha@nnland.edu          58030.94  10000.00      0     48212.79    0+12:32
izzi@nnland.edu            62106.40  10000.00      0      6569.75    0+02:26
marta@cs.wisc.edu          62577.84    500.00     29    193706.30 <now>
kris@proj1.wisc.edu       100597.94  10000.00      0     20814.24    0+04:26
boss@proj1.wisc.edu       318229.25  10000.00      3    324680.47 <now>
---------------------- ------------ --------- ------ ------------ ----------
Number of users: 19                             784   4969073.00    0+23:59
```

Example 2 This is an example of the default output for a pool that uses hierarchical groups, and the groups accept surplus. This leads to a very wide display.

```
% condor_userprio -pool crane.cs.wisc.edu -allusers
Last Priority Update:  1/19 13:18
Group                                  Config     Use    Effective   Priority   Res   Total Usage  Time Since
  User Name                             Quota   Surplus   Priority    Factor   In Use (wghted-hrs) Last Usage
---------------------------------- --------- ------- ------------ --------- ------ ------------ -----------
<none>                                   0.00     yes                    1.00      0         6.78    9+03:52
  johnsm@crane.cs.wisc.edu                                  0.50      1.00      0         6.62    9+19:42
  John.Smith@crane.cs.wisc.edu                              0.50      1.00      0         0.02    9+03:52
  Sedge@crane.cs.wisc.edu                                   0.50      1.00      0         0.05   13+03:03
  Duck@crane.cs.wisc.edu                                    0.50      1.00      0         0.02   31+00:28
  other@crane.cs.wisc.edu                                   0.50      1.00      0         0.04   16+03:42
Duck                                     2.00       no                    1.00      0         0.02   13+02:57
  goose@crane.cs.wisc.edu                                   0.50      1.00      0         0.02   13+02:57
Sedge                                    4.00       no                    1.00      0         0.17    9+03:07
  johnsm@crane.cs.wisc.edu                                  0.50      1.00      0         0.13    9+03:08
  Half@crane.cs.wisc.edu                                    0.50      1.00      0         0.02   31+00:02
  John.Smith@crane.cs.wisc.edu                              0.50      1.00      0         0.05    9+03:07
  other@crane.cs.wisc.edu                                   0.50      1.00      0         0.01   28+19:34
---------------------------------- --------- ------- ------------ --------- ------ ------------ -----------
Number of users: 10                             ByQuota                          0         6.97
```

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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

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** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* | **-constraint** *expression* | **-all**]

Description

condor_vacate causes HTCondor 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. HTCondor 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 HTCondor restarts the job from the beginning somewhere else. *condor_vacate* has no effect on a machine with no HTCondor job currently running.

There is generally no need for the user or administrator to explicitly run *condor_vacate*. HTCondor 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 *hostname* Send the command to a machine identified by *hostname*

hostname Send the command to a machine identified by *hostname*

-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>"*

-constraint *expression* Apply this command only to machines matching the given ClassAd *expression*

-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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_vacate_job

vacate jobs in the HTCondor 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 HTCondor 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. HTCondor 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 HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_version

print HTCondor version and platform information

Synopsis

condor_version [**-help**]

condor_version [**-arch**] [**-opsys**] [**-syscall**]

Description

With no arguments, *condor_version* prints the currently installed HTCondor 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 HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_wait

Wait for jobs to finish

Synopsis

condor_wait [**-help** | **-version**]

condor_wait [**-debug**] [**-status**] [**-echo**] [**-wait** *seconds*] [**-num** *number-of-jobs*] *log-file* [**job ID**]

Description

condor_wait watches a job event 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.

-status Show job start and terminate information.

-echo Print the events out to `stdout`.

-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 HTCondor 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

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

condor_who

Display information about owners of jobs and jobs running on an execute machine

Synopsis

condor_who [*help options*] [*address options*] [*display options*]

Description

condor_who queries and displays information about the user that owns the jobs running on a machine. It is intended to be run on an execute machine.

The options that may be supplied to *condor_who* belong to three groups:

- **Help options** provide information about the *condor_who* tool.
- **Address options** allow destination specification for query.
- **Display options** control the formatting and which of the queried information to display.

At any time, only one **help option** and one **address option** may be specified. Any number of **display options** may be specified.

condor_who obtains its information about jobs by talking to one or more *condor_startd* daemons. So, *condor_who* must identify the command port of any *condor_startd* daemons. An **address option** provides this information. If *no* **address option** is given on the command line, then *condor_who* searches using this ordering:

1. A defined value of the environment variable `CONDOR_CONFIG` specifies the directory where log and address files are to be scanned for needed information.
2. With the aim of finding all *condor_startd* daemons, *condor_who* utilizes the same algorithm it would using the **-allpids** option. The Linux *ps* or the Windows *tasklist* program obtains all PIDs. As Linux `root` or Windows `administrator`, the Linux *lsof* or the Windows *netstat* identifies open sockets and from there the PIDs of listen sockets. Correlating the two lists of PIDs results in identifying the command ports of all *condor_startd* daemons.

Options

-help (help option) Display usage information

-daemons (help option) Display information about the daemons running on the specified machine, including the daemon's PID, IP address and command port

-diagnostic (help option) Display extra information helpful for debugging

-verbose (help option) Display PIDs and addresses of daemons

-address *hostaddress* (address option) Identify the *condor_startd* host address to query

-allpids (address option) Query all local *condor_startd* daemons

-logdir *directoryname* (address option) Specifies the directory containing log and address files that *condor_who* will scan to search for command ports of *condor_start* daemons to query

-pid *PID* (address option) Use the given *PID* to identify the *condor_startd* daemon to query

-long (display option) Display entire ClassAds

-wide (display option) Displays fields without truncating them in order to fit screen width

-format *fmt attr* (display option) Display attribute *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. %r prints the unevaluated, or raw values. 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`. %v identifies the type of the attribute, and then prints the value in an appropriate format. %V identifies the type of the attribute, and then prints the value in an appropriate format as it would appear in the **-long** format. As an example, strings used with %V will have quote marks. 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.

-autoformat[:lhVr,tng] *attr1 [attr2 ...]* or -af[:lhVr,tng] *attr1 [attr2 ...]* (display option) Display attribute(s) or expression(s) formatted in a default way according to attribute types. This option takes an arbitrary number of attribute names as arguments, and prints out their values, with a space between each value and a newline character after the last value. It is like the **-format** option without format strings.

It is assumed that no attribute names begin with a dash character, so that the next word that begins with dash is the start of the next option. The **autoformat** option may be followed by a colon character and formatting qualifiers to deviate the output formatting from the default:

l label each field,

h print column headings before the first line of output,

V use %V rather than %v for formatting (string values are quoted),

r print "raw", or unevaluated values,

, add a comma character after each field,

t add a tab character before each field instead of the default space character,

n add a newline character after each field,

g add a newline character between ClassAds, and suppress spaces before each field.

Use **-af:h** to get tabular values with headings.

Use **-af:lrng** to get -long equivalent format.

The newline and comma characters may *not* be used together. The **l** and **h** characters may *not* be used together.

Examples

Example 1 Sample output from the local machine, which is running a single HTCondor job. Note that the output of the PROGRAM field will be truncated to fit the display, similar to the artificial truncation shown in this example output.

```
% condor_who

OWNER                    CLIENT            SLOT JOB RUNTIME    PID    PROGRAM
smith1@crane.cs.wisc.edu crane.cs.wisc.edu    2 320.0 0+00:00:08 7776 D:\scratch\condor\execut
```

Example 2 Verbose sample output.

```
% condor_who -verbose

LOG directory "D:\scratch\condor\master\test/log"

Daemon       PID      Exit       Addr                      Log, Log.Old
------       ---      ----       ----                      ---, -------
Collector    6788               <128.105.136.32:7977> CollectorLog, CollectorLog.old
Credd        8148               <128.105.136.32:9620> CredLog, CredLog.old
Master       5976               <128.105.136.32:64980> MasterLog,
Match MatchLog, MatchLog.old
Negotiator   6600 NegotiatorLog, NegotiatorLog.old
Schedd       6336               <128.105.136.32:64985> SchedLog, SchedLog.old
Shadow ShadowLog,
Slot1 StarterLog.slot1,
Slot2        7272               <128.105.136.32:65026> StarterLog.slot2,
Slot3 StarterLog.slot3,
Slot4 StarterLog.slot4,
```

```
SoftKill SoftKillLog,
Startd       7416               <128.105.136.32:64984> StartLog, StartLog.old
Starter StarterLog,
TOOL                                                    TOOLLog,

OWNER                    CLIENT            SLOT JOB RUNTIME    PID    PROGRAM
smith1@crane.cs.wisc.edu crane.cs.wisc.edu    2 320.0 0+00:01:28 7776 D:\scratch\condor\execut
```

Exit Status

condor_who will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

gidd_alloc

find a GID within the specified range which is not used by any process

Synopsis

gidd_alloc *min-gid max-gid*

Description

This program will scan the alive PIDs, looking for which GID is unused in the supplied, inclusive range specified by the required arguments *min-gid* and *max-gid*. Upon finding one, it will add the GID to its own supplementary group list, and then scan the PIDs again expecting to find only itself using the GID. If no collision has occurred, the program exits, otherwise it retries.

General Remarks

This is a program only available for the Linux ports of HTCondor.

Exit Status

gidd_alloc will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

procd_ctl

command line interface to the *condor_procd*

Synopsis

procd_ctl -h

procd_ctl -A *address-file* [**command**]

Description

This is a programmatic interface to the *condor_procd* daemon. It may be used to cause the *condor_procd* to do anything that the *condor_procd* is capable of doing, such as tracking and managing process families.

This is a program only available for the Linux ports of HTCondor.

The **-h** option prints out usage information and exits. The *address-file* specification within the **-A** argument specifies the path and file name of the address file which the named pipe clients must use to speak with the *condor_procd*.

One command is given to the *condor_procd*. The choices for the command are defined by the Options.

Options

TRACK_BY_ASSOCIATED_GID *GID* [*PID*] Use the specified *GID* to track the specified family rooted at *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

GET_USAGE [*PID*] Get the total usage information about the PID family at *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

DUMP [*PID*] Print out information about both the root *PID* being watched and the tree of processes under this root *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

LIST [*PID*] With no *PID* given, print out information about all the watched processes. If the optional *PID* is specified, print out information about the process specified by *PID* and all its child processes.

SIGNAL_PROCESS *signal* [*PID*] Send the *signal* to the process specified by *PID*. If the optional *PID* is not specified, then the PID used is the one given or assumed by *condor_procd*.

SUSPEND_FAMILY *PID* Suspend the process family rooted at *PID*.

CONTINUE_FAMILY *PID* Continue execution of the process family rooted at *PID*.

KILL_FAMILY *PID* Kill the process family rooted at *PID*.

UNREGISTER_FAMILY *PID* Stop tracking the process family rooted at *PID*.

SNAPSHOT Perform a snapshot of the tracked family tree.

QUIT Disconnect from the *condor_procd* and exit.

General Remarks

This program may be used in a standalone mode, independent of HTCondor, to track process families. The programs *procd_ctl* and *gidd_alloc* are used with the *condor_procd* in standalone mode to interact with the daemon and inquire about certain state of running processes on the machine, respectively.

Exit Status

procd_ctl will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

Author

Center for High Throughput Computing, University of Wisconsin–Madison

Copyright

Chapter 12

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 1040.

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 987.

`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 1005.

`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 1021.

`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 1022.

`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 1033.

`Submitter` Each submitter is described by a ClassAd. ClassAd attributes that appear in a Submitter ClassAd are listed and described in the unnumbered subsection labeled Submitter ClassAd Attributes on page 1036.

Defrag Each *condor_defrag* daemon describes its state. ClassAd attributes that appear in a Defrag ClassAd are listed and described in the unnumbered subsection labeled Defrag ClassAd Attributes on page 1037.

Collector Each *condor_collector* daemon describes its state. ClassAd attributes that appear in a Collector ClassAd are listed and described in the unnumbered subsection labeled Collector ClassAd Attributes on page 1038.

Query This section has not yet been written

In addition, statistics are published for each DaemonCore daemon. These attributes are listed and described in the unnumbered subsection labeled DaemonCore Statistics Attributes on page 1041.

Job ClassAd Attributes

Absent: Boolean set to true `True` if the ad is absent.

AcctGroup: The accounting group name, as set in the submit description file via the **accounting_group** command. This attribute is only present if an accounting group was requested by the submission. See section 3.6.7 for more information about accounting groups.

AcctGroupUser: The user name associated with the accounting group. This attribute is only present if an accounting group was requested by the submission.

AllRemoteHosts: String containing a comma-separated list of all the remote machines running a parallel or mpi universe job.

Args: A string representing the command line arguments passed to the job, when those arguments are specified using the *old* syntax, as specified in section 11.

Arguments: A string representing the command line arguments passed to the job, when those arguments are specified using the *new* syntax, as specified in section 11.

BatchQueue: For grid universe jobs destined for PBS, LSF or SGE, the name of the queue in the remote batch system.

BlockReadKbytes: The integer number of KiB read from disk for this job.

BlockReads: The integer number of disk blocks read for this job.

BlockWriteKbytes: The integer number of KiB written to disk for this job.

BlockWrites: The integer number of blocks written to disk for this job.

BoincAuthenticatorFile: Used for grid type boinc jobs; a string taken from the definition of the submit description file command **boinc_authenticator_file**. Defines the path and file name of the file containing the authenticator string to use to authenticate to the BOINC service.

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 HTCondor.

Cmd: The path to and the file name of the job to be executed.

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, so the total committed wall time spent running is

```
CommittedTime - CommittedSuspensionTime
```

CommittedSlotTime: This attribute is identical to `CommittedTime` except that the time is multiplied by the `SlotWeight` of the machine(s) that ran the job. This relies on `SlotWeight` being listed in `SYSTEM_JOB_MACHINE_ATTRS`.

CommittedSuspensionTime: A running total of the number of seconds the job has spent in suspension during time in which the job was not evicted without a checkpoint. This number is updated when the job is checkpointed and when it exits.

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).

ConcurrencyLimits: A string list, delimited by commas and space characters. The items in the list identify named resources that the job requires. The value can be a ClassAd expression which, when evaluated in the context of the job ClassAd and a matching machine ClassAd, results in a string list.

CumulativeSlotTime: This attribute is identical to `RemoteWallClockTime` except that the time is multiplied by the `SlotWeight` of the machine(s) that ran the job. This relies on `SlotWeight` being listed in `SYSTEM_JOB_MACHINE_ATTRS`.

CumulativeSuspensionTime: A running total of the number of seconds the job has spent in suspension for the life of the job.

CumulativeTransferTime: The total time, in seconds, that condor has spent transferring the input and output sandboxes for the life of the job.

CurrentHosts: The number of hosts in the claimed state, due to this job.

DAGManJobId: For a DAGMan node job only, the `ClusterId` job ClassAd attribute of the *condor_dagman* job which is the parent of this node job. For nested DAGs, this attribute holds only the `ClusterId` of the job's immediate parent.

DAGParentNodeNames: For a DAGMan node job only, a comma separated list of each *JobName* which is a parent node of this job's node. This attribute is passed through to the job via the *condor_submit* command line, if it does not exceed the line length defined with `_POSIX_ARG_MAX`. For example, if a node job has two parents with *JobName*s B and C, the *condor_submit* command line will contain

```
-append +DAGParentNodeNames=B,C
```

DAGManNodesLog: For a DAGMan node job only, gives the path to an event log used exclusively by DAGMan to monitor the state of the DAG's jobs. Events are written to this log file in addition to any log file specified in the job's submit description file.

DAGManNodesMask: For a DAGMan node job only, a comma-separated list of the event codes that should be written to the log specified by `DAGManNodesLog`, known as the auxiliary log. All events not specified in the `DAGManNodesMask` string are not written to the auxiliary event log. The value of this attribute is determined by DAGMan, and it is passed to the job via the *condor_submit* command line. By default, the following events are written to the auxiliary job log:

- `Submit`, event code is 0
- `Execute`, event code is 1
- `Executable error`, event code is 2
- `Job evicted`, event code is 4
- `Job terminated`, event code is 5
- `Shadow exception`, event code is 7
- `Job aborted`, event code is 9
- `Job suspended`, event code is 10
- `Job unsuspended`, event code is 11
- `Job held`, event code is 12
- `Job released`, event code is 13
- `Post script terminated`, event code is 16
- `Globus submit`, event code is 17
- `Grid submit`, event code is 27

If `DAGManNodesLog` is not defined, it has no effect. The value of `DAGManNodesMask` does not affect events recorded in the job event log file referred to by `UserLog`.

DelegateJobGSICredentialsLifetime: An integer that specifies the maximum number of seconds for which delegated proxies should be valid. The default behavior is determined by the configuration setting `DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME`, which defaults to one day. A value of 0 indicates that the delegated proxy should be valid for as long as allowed by the credential used to create the proxy. This setting currently only applies to proxies delegated for non-grid jobs and HTCondor-C jobs. It does not currently apply to globus grid jobs, which always behave as though this setting were 0. This setting has no effect if the configuration setting `DELEGATE_JOB_GSI_CREDENTIALS` is false, because in that case the job proxy is copied rather than delegated.

DiskUsage: Amount of disk space (KiB) in the HTCondor execute directory on the execute machine that this job has used. An initial value may be set at the job's request, placing into the job's submit description file a setting such as

```
# 1 megabyte initial value
+DiskUsage = 1024
```

vm universe jobs will default to an initial value of the disk image size. If not initialized by the job, non-**vm** universe jobs will default to an initial value of the sum of the job's executable and all input files.

EC2AccessKeyId: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_access_key_id**. Defines the path and file name of the file containing the EC2 Query API's access key.

EC2AmiID: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_ami_id**. Identifies the machine image of the instance.

EC2BlockDeviceMapping: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_block_device_mapping**. Defines the map from block device names to kernel device names for the instance.

EC2ElasticIp: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_elastic_ip**. Specifies an Elastic IP address to associate with the instance.

EC2IamProfileArn: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_iam_profile_arn**. Specifies the IAM (instance) profile to associate with this instance.

EC2IamProfileName: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_iam_profile_name**. Specifies the IAM (instance) profile to associate with this instance.

EC2InstanceName: Used for grid type ec2 jobs; a string set for the job once the instance starts running, as assigned by the EC2 service, that represents the unique ID assigned to the instance by the EC2 service.

EC2InstanceName: Used for grid type ec2 jobs; a string set for the job once the instance starts running, as assigned by the EC2 service, that represents the unique ID assigned to the instance by the EC2 service.

EC2InstanceType: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_instance_type**. Specifies a service-specific instance type.

EC2KeyPair: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_keypair**. Defines the key pair associated with the EC2 instance.

EC2ParameterNames: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_parameter_names**. Contains a space or comma separated list of the names of additional parameters to pass when instantiating an instance.

EC2SpotPrice: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_spot_price**. Defines the maximum amount per hour a job submitter is willing to pay to run this job.

EC2SpotRequestID: Used for grid type ec2 jobs; identifies the spot request HTCondor made on behalf of this job.

EC2StatusReasonCode: Used for grid type ec2 jobs; reports the reason for the most recent EC2-level state transition. Can be used to determine if a spot request was terminated due to a rise in the spot price.

EC2TagNames: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_tag_names**. Defines the set, and case, of tags associated with the EC2 instance.

EC2KeyPairFile: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_keypair_file**. Defines the path and file name of the file into which to write the SSH key used to access the image, once it is running.

EC2RemoteVirtualMachineName: Used for grid type ec2 jobs; a string set for the job once the instance starts running, as assigned by the EC2 service, that represents the host name upon which the instance runs, such that the user can communicate with the running instance.

EC2SecretAccessKey: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_secret_access_key**. Defines that path and file name of the file containing the EC2 Query API's secret access key.

EC2SecurityGroups: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_security_groups**. Defines the list of EC2 security groups which should be associated with the job.

EC2SecurityIDs: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_security_ids**. Defines the list of EC2 security group IDs which should be associated with the job.

EC2UserData: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_user_data**. Defines a block of data that can be accessed by the virtual machine.

EC2UserDataFile: Used for grid type ec2 jobs; a string taken from the definition of the submit description file command **ec2_user_data_file**. Specifies a path and file name of a file containing data that can be accessed by the virtual machine.

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.

EncryptExecuteDirectory: A boolean value taken from the submit description file command **encrypt_execute_directory**. It specifies if HTCondor should encrypt the remote scratch directory on the machine where the job executes.

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

```
time() - EnteredCurrentStatus
```

will equal the number of seconds that the job has been on hold.

Env: A string representing the environment variables passed to the job, when those arguments are specified using the *old* syntax, as specified in section 11.

Environment: A string representing the environment variables passed to the job, when those arguments are specified using the *new* syntax, as specified in section 11.

ExecutableSize: Size of the executable in KiB.

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 HTCondor 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`.

GceAuthFile: Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_auth_file**. Defines the path and file name of the file containing authorization credentials to use the GCE service.

GceImage: Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_image**. Identifies the machine image of the instance.

GceJsonFile: Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_json_file**. Specifies the path and file name of a file containing a set of JSON object members that should be added to the instance description submitted to the GCE service.

GceMachineType: Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_machine_type**. Specifies the hardware profile that should be used for a GCE instance.

GceMetadata: Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_metadata**. Defines a set of name/value pairs that can be accessed by the virtual machine.

GceMetadataFile: Used for grid type gce jobs; a string taken from the definition of the submit description file command **gce_metadata_file**. Specifies a path and file name of a file containing a set of name/value pairs that can be accessed by the virtual machine.

GcePreemptible: Used for grid type gce jobs; a boolean taken from the definition of the submit description file command **gce_preemptible**. Specifies whether the virtual machine instance created in GCE should be preemptible.

GlobalJobId: A string intended to be a unique job identifier within a pool. It currently contains the *condor_schedd* daemon `Name` attribute, a job identifier composed of attributes `ClusterId` and `ProcId` separated by a period, and the job's submission time in seconds since 1970-01-01 00:00:00 UTC, separated by `#` characters. The value `submit.example.com#152.3#1358363336` is an example.

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.

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.

HoldReasonCode: An integer value that represents the reason that a job was put on hold.

Integer Code	*Reason for Hold*	*HoldReasonSubCode*
1	The user put the job on hold with *condor_hold*.	
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 *condor_starter* failed to start the executable.	The Unix errno number.
7	The standard output file for the job could not be opened.	The Unix errno number.
8	The standard input file for the job could not be opened.	The Unix errno number.
9	The standard output stream for the job could not be opened.	The Unix errno number.
10	The standard input stream for the job could not be opened.	The Unix errno number.
11	An internal HTCondor protocol error was encountered when transferring files.	
12	The *condor_starter* or *condor_shadow* failed to receive or write job files.	The Unix errno number.
13	The *condor_starter* or *condor_shadow* failed to read or send job files.	The Unix errno number.
14	The initial working directory of the job cannot be accessed.	The Unix errno 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 *condor_shadow* version available on the submitting machine.	
18	An internal HTCondor protocol error was encountered when transferring files.	
19	`<Keyword>_HOOK_PREPARE_JOB` was defined but could not be executed or returned failure.	
20	The job missed its deferred execution time and therefore failed to run.	
21	The job was put on hold because `WANT_HOLD` in the machine policy was true.	
22	Unable to initialize job event log.	
23	Failed to access user account.	
24	No compatible shadow.	
25	Invalid cron settings.	
26	`SYSTEM_PERIODIC_HOLD` evaluated to true.	
27	The system periodic job policy evaluated to undefined.	
28	Failed while using glexec to set up the job's working directory (chown sandbox to the user).	
30	Failed while using glexec to prepare output for transfer (chown sandbox to condor).	
32	The maximum total input file transfer size was exceeded. (See `MAX_TRANSFER_INPUT_MB`.)	

continued...

Integer Code	*Reason for Hold*	*HoldReasonSubCode*
33	The maximum total output file transfer size was exceeded. (See `MAX_TRANSFER_OUTPUT_MB`.)	
34	Memory usage exceeds a memory limit.	
35	Specified Docker image was invalid.	
36	Job failed when sent the checkpoint signal it requested.	
37	User error in the EC2 universe:	
	Public key file not defined.	1
	Private key file not defined.	2
	Grid resource string missing EC2 service URL.	4
	Failed to authenticate.	9
	Can't use existing SSH keypair with the given server's type.	10
	You, or somebody like you, cancelled this request.	20
38	Internal error in the EC2 universe:	
	Grid resource type not EC2.	3
	Grid resource type not set.	5
	Grid job ID is not for EC2.	7
	Unexpected remote job status.	21
39	Adminstrator error in the EC2 universe:	
	EC2_GAHP not defined.	6
40	Connection problem in the EC2 universe	
	... while creating an SSH keypair.	11
	... while starting an on-demand instance.	12
	... while requesting a spot instance.	17
41	Server error in the EC2 universe:	
	Abnormal instance termination reason.	13
	Unrecognized instance termination reason.	14
	Resource was down for too long.	22
42	Instance potentially lost due to an error in the EC2 universe:	
	Connection error while terminating an instance.	15
	Failed to terminate instance too many times.	16
	Connection error while terminating a spot request.	17
	Failed to terminated a spot request too many times.	18
	Spot instance request purged before instance ID acquired.	19

HoldReasonSubCode: An integer value that represents further information to go along with the `HoldReasonCode`, for some values of `HoldReasonCode`. See `HoldReasonCode` for the values.

HookKeyword: A string that uniquely identifies a set of job hooks, and added to the ClassAd once a job is fetched.

ImageSize: Maximum observed memory image size (i.e. virtual memory) of the job in KiB. The initial value is equal to the size of the executable for non-vm universe jobs, and 0 for vm universe jobs. When the job writes a checkpoint, 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. How quickly

this updated information becomes visible to *condor_q* is controlled by SHADOW_QUEUE_UPDATE_INTERVAL and STARTER_UPDATE_INTERVAL.

Under Linux, ProportionalSetSize is a better indicator of memory usage for jobs with significant sharing of memory between processes, because ImageSize is simply the sum of virtual memory sizes across all of the processes in the job, which may count the same memory pages more than once.

IwdFlushNFSCache: A boolean expression that controls whether or not HTCondor attempts to flush a submit machine's NFS cache, in order to refresh an HTCondor job's initial working directory. The value will be True, unless a job explicitly adds this attribute, setting it to False.

JobAdInformationAttrs: A comma-separated list of attribute names. The named attributes and their values are written in the job event log whenever any event is being written to the log. This is the same as the configuration setting EVENT_LOG_INFORMATION_ATTRS (see page 224) but it applies to the job event log instead of the system event log.

JobDescription: A string that may be defined for a job by setting **description** in the submit description file. When set, tools which display the executable such as *condor_q* will instead use this string. For interactive jobs that do not have a submit description file, this string will default to "Interactive job".

JobCurrentStartDate: Time at which the job most recently began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

JobCurrentStartExecutingDate: Time at which the job most recently finished transferring its input sandbox and began executing. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

JobCurrentStartTransferOutputDate: Time at which the job most recently finished executing and began transferring its output sandbox. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970)

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 2.14.4 for details on job leases.

JobMaxVacateTime: An integer expression that specifies the time in seconds requested by the job for being allowed to gracefully shut down.

JobNotification: An integer indicating what events should be emailed to the user. The integer values correspond to the user choices for the submit command **notification**.

Value	*Notification value*
0	Never
1	Always
2	Complete
3	Error

JobPrio: 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). Due to a long standing bug in the 8.6 series and earlier, the job classad that is internal to the *condor_startd* and *condor_starter* sets this to the time that the job most recently began executing. This bug is scheduled to be fixed in the 8.7 series.

JobStatus: Integer which indicates the current status of the job.

Value	*Status*
1	Idle
2	Running
3	Removed
4	Completed
5	Held
6	Transferring Output
7	Suspended

JobUniverse: Integer which indicates the job universe.

Value	*Universe*
1	standard
5	vanilla, docker
7	scheduler
8	MPI
9	grid
10	java
11	parallel
12	local
13	vm

KeepClaimIdle: An integer value that represents the number of seconds that the *condor_schedd* will continue to keep a claim, in the Claimed Idle state, after the job with this attribute defined completes, and there are no other jobs ready to run from this user. This attribute may improve the performance of linear DAGs, in the case when a dependent job can not be scheduled until its parent has completed. Extending the claim on the machine may permit the dependent job to be scheduled with less delay than with waiting for the *condor_negotiator* to match with a new machine.

KillSig: The Unix signal number that the job wishes to be sent before being forcibly killed. It is relevant only for jobs running on Unix machines.

KillSigTimeout: This attribute is replaced by the functionality in `JobMaxVacateTime` as of HTCondor version 7.7.3. The number of seconds that the job (other than the standard universe) requests the *condor_starter* wait after sending the signal defined as `KillSig` and before forcibly removing the job. The actual amount of time will be the minimum of this value and the execute machine's configuration variable `KILLING_TIMEOUT`.

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 HTCondor-G last tried to find a match for the job, but failed to do so.

LastRemotePool: The name of the *condor_collector* of the pool in which a job ran via flocking in the most recent run attempt. This attribute is not defined if the job did not run via flocking.

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).

LeaveJobInQueue: A boolean expression that defaults to `False`, causing the job to be removed from the queue upon completion. An exception is if the job is submitted using `condor_submit -spool`. For this case, the default expression causes the job to be kept in the queue for 10 days after completion.

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.

MachineAttr<X><N>: Machine attribute of name <X> that is placed into this job ClassAd, as specified by the configuration variable `SYSTEM_JOB_MACHINE_ATTRS`. With the potential for multiple run attempts, <N> represents an integer value providing historical values of this machine attribute for multiple runs. The most recent run will have a value of <N> equal to `0`. The next most recent run will have a value of <N> equal to `1`.

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.

MaxTransferInputMB: This integer expression specifies the maximum allowed total size in Mbytes of the input files that are transferred for a job. This expression does *not* apply to grid universe, standard universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not set, the system setting `MAX_TRANSFER_INPUT_MB` is used. If the observed size of all input files at submit time is larger than the limit, the job will be immediately placed on hold with a `HoldReasonCode` value of 32. If the job passes this initial test, but the size of the input files increases or the limit decreases so that the limit is violated, the job will be placed on hold at the time when the file transfer is attempted.

MaxTransferOutputMB: This integer expression specifies the maximum allowed total size in Mbytes of the output files that are transferred for a job. This expression does *not* apply to grid universe, standard universe, or files transferred via file transfer plug-ins. The expression may refer to attributes of the job. The special value -1 indicates no limit. If not set, the system setting `MAX_TRANSFER_OUTPUT_MB` is used. If the total size of the job's output files to be transferred is larger than the limit, the job will be placed on hold with a `HoldReasonCode` value of 33. The output will be transferred up to the point when the limit is hit, so some files may be fully transferred, some partially, and some not at all.

MemoryUsage: An integer expression in units of Mbytes that represents the peak memory usage for the job. Its purpose is to be compared with the value defined by a job with the **request_memory** submit command, for purposes of policy evaluation.

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 when `True` indicates that this job is a *nice* job, raising its user priority value, thus causing it to run on a machine only when no other HTCondor jobs want the machine.

Nonessential: A boolean value only relevant to grid universe jobs, which when `True` tells HTCondor to simply abort (remove) any problematic job, instead of putting the job on hold. It is the equivalent of doing *condor_rm* followed by *condor_rm* **-forcex** any time the job would have otherwise gone on hold. If not explicitly set to `True`, the default value will be `False`.

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.

NumJobCompletions: An integer, initialized to zero, that is incremented by the *condor_shadow* each time the job's executable exits of its own accord, with or without errors, and successfully completes file transfer (if requested). Jobs which have done so normally enter the completed state; this attribute is therefore normally only of use when, for example, `on_exit_remove` or `on_exit_hold` is set.

NumJobMatches: An integer that is incremented by the *condor_schedd* each time the job is matched with a resource ad by the negotiator.

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 reconnected: those in the **vanilla** and **java** universes.

NumJobStarts: An integer count of the number of times the job started executing. This is not (yet) defined for **standard** universe jobs.

NumPids: A count of the number of child processes that this job has.

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. **Note that this attribute is incremented every time the job is matched, even if the match is rejected by the execute machine; in other words, the value of this attribute may be greater than the number of times the job actually ran.**

NumSystemHolds: An integer that is incremented each time HTCondor-G places a job on hold due to some sort of error condition. This counter is useful, since HTCondor-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.

OtherJobRemoveRequirements: A string that defines a list of jobs. When the job with this attribute defined is removed, all other jobs defined by the list are also removed. The string is an expression that defines a constraint equivalent to the one implied by the command

```
condor_rm -constraint <constraint>
```

This attribute is used for jobs managed with *condor_dagman* to ensure that node jobs of the DAG are removed when the *condor_dagman* job itself is removed. Note that the list of jobs defined by this attribute must not form a cyclic removal of jobs, or the *condor_schedd* will go into an infinite loop when any of the jobs is removed.

OutputDestination: A URL, as defined by submit command **output_destination**.

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, HTCondor's behavior changes:

"WAIT_FOR_ALL" HTCondor will wait until every node in the parallel job has completed to consider the job finished.

PreJobPrio1: An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. The range of valid values is `INT_MIN + 1` to `INT_MAX`. When not explicitly set for a job, `INT_MIN`, the lowest possible priority, is used for comparison purposes. This attribute, when set, is considered first: before `PreJobPrio2`, before `JobPrio`, before `PostJobPrio1`, before `PostJobPrio2`, and before `QDate`.

PreJobPrio2: An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. The range of valid values is `INT_MIN + 1` to `INT_MAX`. When not explicitly set for a job, `INT_MIN`, the lowest possible priority, is used for comparison purposes. This attribute, when set, is considered after `PreJobPrio1`, but before `JobPrio`, before `PostJobPrio1`, before `PostJobPrio2`, and before `QDate`.

PostJobPrio1: An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. The range of valid values is `INT_MIN + 1` to `INT_MAX`. When not explicitly set for a job, `INT_MIN`, the lowest possible priority, is used for comparison purposes. This attribute, when set, is considered after `PreJobPrio1`, after `PreJobPrio1`, and after `JobPrio`, but before `PostJobPrio2`, and before `QDate`.

PostJobPrio2: An integer value representing a user's priority to affect of choice of jobs to run. A larger value gives higher priority. The range of valid values is `INT_MIN + 1` to `INT_MAX`. When not explicitly set for a job, `INT_MIN`, the lowest possible priority, is used for comparison purposes. This attribute, when set, is considered after `PreJobPrio1`, after `PreJobPrio1`, after `JobPrio`, and after `PostJobPrio1`, but before `QDate`.

PreserveRelativeExecutable: When `True`, the *condor_starter* will not prepend `Iwd` to `Cmd`, when `Cmd` is a relative path name and `TransferExecutable` is `False`. The default value is `False`. This attribute is primarily of interest for users of `USER_JOB_WRAPPER` for the purpose of allowing an executable's location to be resolved by the user's path in the job wrapper.

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.

ProportionalSetSizeKb: On Linux execute machines with kernel version more recent than 2.6.27, this is the maximum observed proportional set size (PSS) in KiB, summed across all processes in the job. If the execute machine does not support monitoring of PSS or PSS has not yet been measured, this attribute will be undefined. PSS differs from `ImageSize` in how memory shared between processes is accounted. The PSS for one process is the sum of that process' memory pages divided by the number of processes sharing each of the pages. `ImageSize` is the same, except there is no division by the number of processes sharing the pages.

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).

RecentBlockReadKbytes: The integer number of KiB read from disk for this job over the previous time interval defined by configuration variable `STATISTICS_WINDOW_SECONDS`.

RecentBlockReads: The integer number of disk blocks read for this job over the previous time interval defined by configuration variable `STATISTICS_WINDOW_SECONDS`.

RecentBlockWriteKbytes: The integer number of KiB written to disk for this job over the previous time interval defined by configuration variable `STATISTICS_WINDOW_SECONDS`.

RecentBlockWrites: The integer number of blocks written to disk for this job over the previous time interval defined by configuration variable `STATISTICS_WINDOW_SECONDS`.

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.

RemotePool: The name of the *condor_collector* of the pool in which a job is running via flocking. This attribute is not defined if the job is not running via flocking.

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.

CumulativeRemoteSysCpu: The total number of seconds of system CPU time the job used on remote machines, summed over all execution attempts.

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. A job in the virtual machine universe will only report this attribute if run on a KVM hypervisor.

CumulativeRemoteUserCpu: The total number of seconds of user CPU time the job used on remote machines, summed over all execution attempts.

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. `CommittedTime`, on the other hand, is just like `RemoteWallClockTime` except it does get reset to 0 whenever the job is evicted without a checkpoint.

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.

RequestCpus: The number of CPUs requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum number of CPUs that are needed in the created dynamic slot.

RequestDisk: The amount of disk space in KiB requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum amount of disk space needed in the created dynamic slot.

RequestedChroot: A full path to the directory that the job requests the *condor_starter* use as an argument to `chroot()`.

RequestMemory: The amount of memory space in MiB requested for this job. If dynamic *condor_startd* provisioning is enabled, it is the minimum amount of memory needed in the created dynamic slot. If not set by the job, its definition is specified by configuration variable `JOB_DEFAULT_REQUESTMEMORY`.

ResidentSetSize: Maximum observed physical memory in use by the job in KiB while running.

StackSize: Utilized for Linux jobs only, the number of bytes allocated for stack space for this job. This number of bytes replaces the default allocation of 512 Mbytes.

StageOutFinish: An attribute representing a Unix epoch time that is defined for a job that is spooled to a remote site using `condor_submit -spool` or HTCondor-C and causes HTCondor to hold the output in the spool while the job waits in the queue in the `Completed` state. This attribute is defined when retrieval of the output finishes.

StageOutStart: An attribute representing a Unix epoch time that is defined for a job that is spooled to a remote site using `condor_submit -spool` or HTCondor-C and causes HTCondor to hold the output in the spool while the job waits in the queue in the `Completed` state. This attribute is defined when retrieval of the output begins.

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.

SubmitterAutoregroup: A boolean attribute defined by the *condor_negotiator* when it makes a match. It will be `True` if the resource was claimed via negotiation when the configuration variable `GROUP_AUTOREGROUP` was `True`. It will be `False` otherwise.

SubmitterGlobalJobId: When HTCondor-C submits a job to a remote *condor_schedd*, it sets this attribute in the remote job ad to match the `GlobalJobId` attribute of the original, local job.

SubmitterGroup: The accounting group name defined by the *condor_negotiator* when it makes a match.

SubmitterNegotiatingGroup: The accounting group name under which the resource negotiated when it was claimed, as set by the *condor_negotiator*.

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`.

TransferInputSizeMB: The total size in Mbytes of input files to be transferred for the job. Files transferred via file transfer plug-ins are not included. This attribute is automatically set by *condor_submit*; jobs submitted via other submission methods, such as SOAP, may not define this attribute.

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`.

TransferringInput: A boolean value that indicates whether the job is currently transferring input files. The value is `Undefined` if the job is not scheduled to run or has not yet attempted to start transferring input. When this value is `True`, to see whether the transfer is active or queued, check `TransferQueued`.

TransferringOutput: A boolean value that indicates whether the job is currently transferring output files. The value is `Undefined` if the job is not scheduled to run or has not yet attempted to start transferring output. When this value is `True`, to see whether the transfer is active or queued, check `TransferQueued`.

TransferQueued: A boolean value that indicates whether the job is currently waiting to transfer files because of limits placed by `MAX_CONCURRENT_DOWNLOADS` or `MAX_CONCURRENT_UPLOADS`.

UserLog: The full path and file name on the submit machine of the log file of job events.

WantGracefulRemoval: A boolean expression that, when `True`, specifies that a graceful shutdown of the job should be done when the job is removed or put on hold.

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.

X509UserProxy: The full path and file name of the file containing the X.509 user proxy.

X509UserProxyEmail:

For a job with an X.509 proxy credential, this is the email address extracted from the proxy.

X509UserProxyExpiration: For a job that defines the submit description file command **x509userproxy**, this is the time at which the indicated X.509 proxy credential will expire, measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

X509UserProxyFirstFQAN: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this is the VOMS Fully Qualified Attribute Name (FQAN) of the primary role of the credential. A credential may have multiple roles defined, but by convention the one listed first is the primary role.

X509UserProxyFQAN: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this is a serialized list of the DN and all FQAN. A comma is used as a separator, and any existing commas in the DN or FQAN are replaced with the string `&comma;`. Likewise, any ampersands in the DN or FQAN are replaced with `&`.

X509UserProxySubject: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this attribute contains the Distinguished Name (DN) of the credential used to submit the job.

X509UserProxyVOName: For a vanilla or grid universe job that defines the submit description file command **x509userproxy**, this is the name of the VOMS virtual organization (VO) that the user's credential is part of.

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.

The following job ClassAd attributes appear in the job ClassAd only for the *condor_dagman* job submitted under DAGMan. They represent status information for the DAG.

DAG_InRecovery: The value 1 if the DAG is in recovery mode, and The value 0 otherwise.

DAG_NodesDone: The number of DAG nodes that have finished successfully. This means that the entire node has finished, not only an actual HTCondor job or jobs.

DAG_NodesFailed: The number of DAG nodes that have failed. This value includes all retries, if there are any.

DAG_NodesPostrun: The number of DAG nodes for which a POST script is running or has been deferred because of a POST script throttle setting.

DAG_NodesPrerun: The number of DAG nodes for which a PRE script is running or has been deferred because of a PRE script throttle setting.

DAG_NodesQueued: The number of DAG nodes for which the actual HTCondor job or jobs are queued. The queued jobs may be in any state.

DAG_NodesReady: The number of DAG nodes that are ready to run, but which have not yet started running.

DAG_NodesTotal: The total number of nodes in the DAG, including the FINAL node, if there is a FINAL node.

DAG_NodesUnready: The number of DAG nodes that are not ready to run. This is a node in which one or more of the parent nodes has not yet finished.

DAG_Status: The overall status of the DAG, with the same values as the macro `$DAG_STATUS` used in DAGMan FINAL nodes.

Value	*Status*
0	OK
1	error; an error condition different than those listed here
2	one or more nodes in the DAG have failed
3	the DAG has been aborted by an ABORT-DAG-ON specification
4	removed; the DAG has been removed by *condor_rm*
5	a cycle was found in the DAG
6	the DAG has been suspended (see section 2.10.8)

The following job ClassAd attributes do *not* appear in the job ClassAd as kept by the *condor_schedd* daemon. They appear in the job ClassAd written to the job's execute directory while the job is running.

CpusProvisioned: The number of Cpus allocated to the job. With statically-allocated slots, it is the number of Cpus allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `RequestCpus`, but may be larger due to the minimum given to a dynamic slot.

DiskProvisioned: The amount of disk space in KiB allocated to the job. With statically-allocated slots, it is the amount of disk space allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `RequestDisk`, but may be larger due to the minimum given to a dynamic slot.

MemoryProvisioned: The amount of memory in MiB allocated to the job. With statically-allocated slots, it is the amount of memory space allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `RequestMemory`, but may be larger due to the minimum given to a dynamic slot.

<Name>Provisioned: The amount of the custom resource identified by `<Name>` allocated to the job. For jobs using GPUs, `<Name>` will be `GPUs`. With statically-allocated slots, it is the amount of the resource allocated to the slot. With dynamically-allocated slots, it is based upon the job attribute `Request<Name>`, but may be larger due to the minimum given to a dynamic slot.

Machine ClassAd Attributes

Activity: String which describes HTCondor 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

"Retiring": Waiting for a job to finish or for the maximum retirement time to expire

Arch: String with the architecture of the machine. Currently supported architectures have the following string definitions:

"INTEL": Intel x86 CPU (Pentium, Xeon, etc).

"X86_64": AMD/Intel 64-bit X86

These strings show definitions for architectures no longer supported:

"IA64": Intel Itanium

"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": 32-bit PowerPC

"PPC64": 64-bit PowerPC

CanHibernate: The *condor_startd* has the capability to shut down or hibernate a machine when certain configurable criteria are met. However, before the *condor_startd* can shut down a machine, the hardware itself must support hibernation, as must the operating system. When the *condor_startd* initializes, it checks for this support. If the machine has the ability to hibernate, then this boolean ClassAd attribute will be `True`. By default, it is `False`.

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, . . . , and 6 = Saturday.

ClockMin: The number of minutes passed since midnight.

CondorLoadAvg: The load average contributed by HTCondor, either from remote jobs or running benchmarks.

CondorVersion: A string containing the HTCondor version number for the *condor_startd* daemon, the release date, and the build identification number.

ConsoleIdle: The number of seconds since activity on the system console keyboard or console mouse has last been detected. The value can be modified with `SLOTS_CONNECTED_TO_CONSOLE` as defined at 3.5.8.

Cpus: The number of CPUs (cores) in this slot. It is 1 for a single CPU slot, 2 for a dual CPU slot, etc. For a partitionable slot, it is the remaining number of CPUs in the partitionable slot.

CpuFamily: On Linux machines, the Cpu family, as defined in the /proc/cpuinfo file.

CpuModel: On Linux machines, the Cpu model number, as defined in the /proc/cpuinfo file.

CpuCacheSize: On Linux machines, the size of the L3 cache, in kbytes, as defined in the /proc/cpuinfo file.

CurrentRank: A float which represents this machine owner's affinity for running the HTCondor job which it is currently hosting. If not currently hosting an HTCondor 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.

DetectedCpus: Set by the value of configuration variable `DETECTED_CORES`.

DetectedMemory: Set by the value of configuration variable `DETECTED_MEMORY`. Specified in MiB.

Disk: The amount of disk space on this machine available for the job in KiB (for example, 23000 = 23 MiB). Specifically, this is the amount of disk space available in the directory specified in the HTCondor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro. For static slots, this value will be the same as machine ClassAd attribute `TotalSlotDisk`. For partitionable slots, this value will be the quantity of disk space remaining in the partitionable slot.

Draining: This attribute is `True` when the slot is draining and undefined if not.

DrainingRequestId: This attribute contains a string that is the request id of the draining request that put this slot in a draining state. It is undefined if the slot is not draining.

DotNetVersions: The .NET framework versions currently installed on this computer. Default format is a comma delimited list. Current definitions:

"1.1": for .Net Framework 1.1

"2.0": for .Net Framework 2.0

"3.0": for .Net Framework 3.0

"3.5": for .Net Framework 3.5

"4.0Client": for .Net Framework 4.0 Client install

"4.0Full": for .Net Framework 4.0 Full install

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).

ExpectedMachineGracefulDrainingBadput: The job run time in cpu-seconds that would be lost if graceful draining were initiated at the time this ClassAd was published. This calculation assumes that jobs will run for the full retirement time and then be evicted without saving a checkpoint.

ExpectedMachineGracefulDrainingCompletion: The estimated time at which graceful draining of the machine could complete if it were initiated at the time this ClassAd was published and there are no active claims. This is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This value is computed with the assumption that the machine policy will not suspend jobs during draining while the machine is waiting for the job to use up its retirement time. If suspension happens, the upper bound on how long draining could take is unlimited. To avoid suspension during draining, the `SUSPEND` and `CONTINUE` expressions could be configured to pay attention to the `Draining` attribute.

ExpectedMachineGracefulQuickBadput: The job run time in cpu-seconds that would be lost if quick or fast draining were initiated at the time this ClassAd was published. This calculation assumes that all evicted jobs will not save a checkpoint.

ExpectedMachineQuickDrainingCompletion: Time at which quick or fast draining of the machine could complete if it were initiated at the time this ClassAd was published and there are no active claims. 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 HTCondor 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.

HasDocker: A boolean value set to `True` if the machine is capable of executing docker universe jobs.

HasEncryptExecuteDirectory: A boolean value set to `True` if the machine is capable of encrypting execute directories.

HasFileTransfer: A boolean value that when `True` identifies that the machine can use the file transfer mechanism.

HasFileTransferPluginMethods: A string of comma-separated file transfer protocols that the machine can support. The value can be modified with `FILETRANSFER_PLUGINS` as defined at 3.5.11.

Has_sse4_1: A boolean value set to `True` if the machine being advertised supports the SSE 4.1 instructions, and `Undefined` otherwise.

Has_sse4_2: A boolean value set to `True` if the machine being advertised supports the SSE 4.2 instructions, and `Undefined` otherwise.

has_ssse3: A boolean value set to `True` if the machine being advertised supports the SSSE 3 instructions, and `Undefined` otherwise.

has_avx: A boolean value set to `True` if the machine being advertised supports the avx instructions, and `Undefined` otherwise.

HasSingularity: A boolean value set to `True` if the machine being advertised supports running jobs within Singularity containers.

HasVM: If the configuration triggers the detection of virtual machine software, a boolean value reporting the success thereof; otherwise undefined. May also become `False` if HTCondor determines that it can't start a VM (even if the appropriate software is detected).

IsWakeAble: A boolean value that when `True` identifies that the machine has the capability to be woken into a fully powered and running state by receiving a Wake On LAN (WOL) packet. This ability is a function of the operating system, the network adapter in the machine (notably, wireless network adapters usually do not have this function), and BIOS settings. When the *condor_startd* initializes, it tries to detect if the operating system and network adapter both support waking from hibernation by receipt of a WOL packet. The default value is `False`.

IsWakeEnabled: If the hardware and software have the capacity to be woken into a fully powered and running state by receiving a Wake On LAN (WOL) packet, this feature can still be disabled via the BIOS or software. If BIOS or the operating system have disabled this feature, the *condor_startd* sets this boolean attribute to `False`.

JobPreemptions: The total number of times a running job has been preempted on this machine.

JobRankPreemptions: The total number of times a running job has been preempted on this machine due to the machine's rank of jobs since the *condor_startd* started running.

JobStarts: The total number of jobs which have been started on this machine since the *condor_startd* started running.

JobUserPrioPreemptions: The total number of times a running job has been preempted on this machine based on a fair share allocation of the pool since the *condor_startd* started running.

JobVM_VCPUS: An attribute defined if a vm universe job is running on this slot. Defined by the number of virtualized CPUs in the virtual machine.

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. Pseudo-terminals have virtual keyboard activity from telnet and rlogin sessions. Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`. The value can be modified with `SLOTS_CONNECTED_TO_KEYBOARD` as defined at 3.5.8.

KFlops: Relative floating point performance as determined via a Linpack benchmark.

LastDrainStartTime: Time when draining of this *condor_startd* was last initiated (e.g. due to *condor_defrag* or *condor_drain*).

LastHeardFrom: Time when the HTCondor 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 representing the current load average.

Machine: A string with the machine's fully qualified host name.

MachineMaxVacateTime: An integer expression that specifies the time in seconds the machine will allow the job to gracefully shut down.

MaxJobRetirementTime: When the *condor_startd* wants to kick the job off, a job which has run for less than this number of seconds will not be hard-killed. The *condor_startd* will wait for the job to finish or to exceed this amount of time, whichever comes sooner. If the job vacating policy grants the job X seconds of vacating time, a preempted job will be soft-killed X seconds before the end of its retirement time, so that hard-killing of the job will not happen until the end of the retirement time if the job does not finish shutting down before then. This is an expression evaluated in the context of the job ClassAd, so it may refer to job attributes as well as machine attributes.

Memory: The amount of RAM in MiB in this slot. For static slots, this value will be the same as in `TotalSlotMemory`. For a partitionable slot, this value will be the quantity remaining in the partitionable slot.

Mips: Relative integer performance as determined via a Dhrystone benchmark.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in KiB.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in KiB.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

MyAddress: String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd. When using CCB, *condor_shared_port*, and/or an additional private network interface, that information will be included here as well.

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.

Offline<name>: A string that lists specific instances of a user-defined machine resource, identified by `name`. Each instance is currently unavailable for purposes of match making.

OfflineUniverses: A ClassAd list that specifies which job universes are presently offline, both as strings and as the corresponding job universe number. Could be used the the startd to refuse to start jobs in offline universes:

```
START = OfflineUniverses is undefined || (! member( JobUniverse, OfflineUniverses )
```

May currently only contain `"VM"` and `13`.

OpSys: String describing the operating system running on this machine. Currently supported operating systems have the following string definitions:

"LINUX": for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, LINUX 2.6.x, or LINUX 3.10.0 kernel systems, as well as Scientific Linux, Ubuntu versions 14.04, and Debian 7.0 (wheezy) and 8.0 (jessie)

"OSX": for Darwin

"FREEBSD7": for FreeBSD 7

"FREEBSD8": for FreeBSD 8

"WINDOWS": for all versions of Windows

"SOLARIS5.10": for Solaris 2.10 or 5.10

"SOLARIS5.11": for Solaris 2.11 or 5.11

These strings show definitions for operating systems no longer supported:

"SOLARIS28": for Solaris 2.8 or 5.8

"SOLARIS29": for Solaris 2.9 or 5.9

OpSysAndVer: A string indicating an operating system and a version number.

For Linux operating systems, it is the value of the `OpSysName` attribute concatenated with the string version of the `OpSysMajorVer` attribute:

"RedHat5": for RedHat Linux version 5

"RedHat6": for RedHat Linux version 6

"RedHat7": for RedHat Linux version 7

"Fedora16": for Fedora Linux version 16

"Debian6": for Debian Linux version 6

"Debian7": for Debian Linux version 7

"Debian8": for Debian Linux version 8

"Debian9": for Debian Linux version 9

"Ubuntu14": for Ubuntu 14.04

"SL5": for Scientific Linux version 5

"SL6": for Scientific Linux version 6

"SLFermi5": for Fermi's Scientific Linux version 5

"SLFermi6": for Fermi's Scientific Linux version 6

"SLCern5": for CERN's Scientific Linux version 5

"SLCern6": for CERN's Scientific Linux version 6

For MacOS operating systems, it is the value of the `OpSysShortName` attribute concatenated with the string version of the `OpSysVer` attribute:

"MacOSX605": for MacOS version 10.6.5 (Snow Leopard)

"MacOSX703": for MacOS version 10.7.3 (Lion)

For BSD operating systems, it is the value of the `OpSysName` attribute concatenated with the string version of the `OpSysMajorVer` attribute:

"FREEBSD7": for FreeBSD version 7

"FREEBSD8": for FreeBSD version 8

For Solaris Unix operating systems, it is the same value as the `OpSys` attribute:

"SOLARIS5.10": for Solaris 2.10 or 5.10

"SOLARIS5.11": for Solaris 2.11 or 5.11

For Windows operating systems, it is the value of the `OpSys` attribute concatenated with the string version of the `OpSysMajorVer` attribute:

"WINDOWS500": for Windows 2000

"WINDOWS501": for Windows XP

"WINDOWS502": for Windows Server 2003

"WINDOWS600": for Windows Vista

"WINDOWS601": for Windows 7

OpSysLegacy: A string that holds the long-standing values for the `OpSys` attribute. Currently supported operating systems have the following string definitions:

"LINUX": for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, LINUX 2.6.x, or LINUX 3.10.0 kernel systems, as well as Scientific Linux, Ubuntu versions 14.04, and Debian 7 and 8

"OSX": for Darwin

"FREEBSD7": for FreeBSD version 7

"FREEBSD8": for FreeBSD version 8

"SOLARIS5.10": for Solaris 2.10 or 5.10

"SOLARIS5.11": for Solaris 2.11 or 5.11

"WINDOWS": for all versions of Windows

OpSysLongName: A string giving a full description of the operating system. For Linux platforms, this is generally the string taken from `/etc/hosts`, with extra characters stripped off Debian versions.

"Red Hat Enterprise Linux Server release 5.7 (Tikanga)": for RedHat Linux version 5

"Red Hat Enterprise Linux Server release 6.2 (Santiago)": for RedHat Linux version 6

"Red Hat Enterprise Linux Server release 7.0 (Maipo)": for RedHat Linux version 7.0

"Ubuntu 14.04.1 LTS": for Ubuntu 14.04 point release 1

"Debian GNU/Linux 7": for Debian 7.0 (wheezy)

"Debian GNU/Linux 8": for Debian 8.0 (jessie)

"Fedora release 16 (Verne)": for Fedora Linux version 16

"MacOSX 6.5": for MacOS version 10.6.5 (Snow Leopard)

"MacOSX 7.3": for MacOS version 10.7.3 (Lion)

"FreeBSD8.2-RELEASE-p3": for FreeBSD version 8

"SOLARIS5.10": for Solaris 2.10 or 5.10

"SOLARIS5.11": for Solaris 2.11 or 5.11

"Windows XP SP3": for Windows XP

"Windows 7 SP2": for Windows 7

OpSysMajorVer: An integer value representing the major version of the operating system.

5: for RedHat Linux version 5 and derived platforms such as Scientific Linux

6: for RedHat Linux version 6 and derived platforms such as Scientific Linux

7: for RedHat Linux version 7

14: for Ubuntu 14.04

7: for Debian 7

8: for Debian 8

16: for Fedora Linux version 16

6: for MacOS version 10.6.5 (Snow Leopard)

7: for MacOS version 10.7.3 (Lion)

7: for FreeBSD version 7

8: for FreeBSD version 8

5: for Solaris 2.10, 5.10, 2.11, or 5.11

501: for Windows XP

600: for Windows Vista

601: for Windows 7

OpSysName: A string containing a terse description of the operating system.

"RedHat": for RedHat Linux version 6 and 7

"Fedora": for Fedora Linux version 16

"Ubuntu": for Ubuntu versions 14.04

"Debian": for Debian versions 7 and 8

"SnowLeopard": for MacOS version 10.6.5 (Snow Leopard)

"Lion": for MacOS version 10.7.3 (Lion)

"FREEBSD": for FreeBSD version 7 or 8

"SOLARIS5.10": for Solaris 2.10 or 5.10

"SOLARIS5.11": for Solaris 2.11 or 5.11

"WindowsXP": for Windows XP

"WindowsVista": for Windows Vista

"Windows7": for Windows 7

"SL": for Scientific Linux

"SLFermi": for Fermi's Scientific Linux

"SLCern": for CERN's Scientific Linux

OpSysShortName: A string containing a short name for the operating system.

"RedHat": for RedHat Linux version 5, 6 or 7

"Fedora": for Fedora Linux version 16

"Debian": for Debian Linux version 6 or 7 or 8

"Ubuntu": for Ubuntu versions 14.04

"MacOSX": for MacOS version 10.6.5 (Snow Leopard) or for MacOS version 10.7.3 (Lion)

"FreeBSD": for FreeBSD version 7 or 8

"SOLARIS5.10": for Solaris 2.10 or 5.10

"SOLARIS5.11": for Solaris 2.11 or 5.11

"XP": for Windows XP

"Vista": for Windows Vista

"7": for Windows 7

"SL": for Scientific Linux

"SLFermi": for Fermi's Scientific Linux

"SLCern": for CERN's Scientific Linux

OpSysVer: An integer value representing the operating system version number.

700: for RedHat Linux version 7.0

602: for RedHat Linux version 6.2

1600: for Fedora Linux version 16.0

1404: for Ubuntu 14.04

700: for Debian 7.0

800: for Debian 8.0

704: for FreeBSD version 7.4

802: for FreeBSD version 8.2

605: for MacOS version 10.6.5 (Snow Leopard)

703: for MacOS version 10.7.3 (Lion)

500: for Windows 2000

501: for Windows XP

502: for Windows Server 2003

600: for Windows Vista or Windows Server 2008

601: for Windows 7 or Windows Server 2008

PartitionableSlot: For SMP machines, a boolean value identifying that this slot may be partitioned.

RecentJobPreemptions: The total number of jobs which have been preempted from this machine in the last twenty minutes.

RecentJobRankPreemptions: The total number of times a running job has been preempted on this machine due to the machine's rank of jobs in the last twenty minutes.

RecentJobStarts: The total number of jobs which have been started on this machine in the last twenty minutes.

RecentJobUserPrio: The total number of times a running job has been preempted on this machine based on a fair share allocation of the pool in the last twenty minutes.

Requirements: A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before HTCondor will allow the job to use this machine.

RetirementTimeRemaining: An integer number of seconds after `MyCurrentTime` when the running job can be evicted. `MaxJobRetirementTime` is the expression of how much retirement time the machine offers to new jobs, whereas `RetirementTimeRemaining` is the negotiated amount of time remaining for the current running job. This may be less than the amount offered by the machine's `MaxJobRetirementTime` expression, because the job may ask for less.

SingularityVersion: A string containing the version of Singularity available, if the machine being advertised supports running jobs within a Singularity container (see `HasSingularity`).

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 HTCondor version 6.9.3. For older versions of HTCondor, see `VirtualMachineID` below.

SlotType: For SMP machines with partitionable slots, the partitionable slot will have this attribute set to `"Partitionable"`, and all dynamic slots will have this attribute set to `"Dynamic"`.

SlotWeight: This specifies the weight of the slot when calculating usage, computing fair shares, and enforcing group quotas. For example, claiming a slot with `SlotWeight = 2` is equivalent to claiming two `SlotWeight = 1` slots. See the description of `SlotWeight` on page 257.

StartdIpAddr: String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd. When using CCB, *condor_shared_port*, and/or an additional private network interface, that information will be included here as well.

State: String which publishes the machine's HTCondor state. Can be:

"Owner": The machine owner is using the machine, and it is unavailable to HTCondor.

"Unclaimed": The machine is available to run HTCondor jobs, but a good match is either not available or not yet found.

"Matched": The HTCondor central manager has found a good match for this resource, but an HTCondor scheduler has not yet claimed it.

"Claimed": The machine is claimed by a remote *condor_schedd* and is probably running a job.

"Preempting": An HTCondor 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.

"Drained": This slot is not accepting jobs, because the machine is being drained.

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.

TotalCondorLoadAvg: The load average contributed by HTCondor summed across all slots on the machine, either from remote jobs or running benchmarks.

TotalCpus: The number of CPUs (cores) that are on the machine. This is in contrast with `Cpus`, which is the number of CPUs in the slot.

TotalDisk: The quantity of disk space in KiB available across the machine (not the slot). For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalSlotDisk`.

TotalLoadAvg: A floating point number representing the current load average summed across all slots on the machine.

TotalMachineDrainingBadput: The total job runtime in cpu-seconds that has been lost due to job evictions caused by draining since this *condor_startd* began executing. In this calculation, it is assumed that jobs are evicted without checkpointing.

TotalMachineDrainingUnclaimedTime: The total machine-wide time in cpu-seconds that has not been used (i.e. not matched to a job submitter) due to draining since this *condor_startd* began executing.

TotalMemory: The quantity of RAM in MiB available across the machine (not the slot). For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalSlotMemory`.

TotalSlotCpus: The number of CPUs (cores) in this slot. For static slots, this value will be the same as in `Cpus`.

TotalSlotDisk: The quantity of disk space in KiB given to this slot. For static slots, this value will be the same as machine ClassAd attribute `Disk`. For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalDisk`.

TotalSlotMemory: The quantity of RAM in MiB given to this slot. For static slots, this value will be the same as machine ClassAd attribute `Memory`. For partitionable slots, where there is one partitionable slot per machine, this value will be the same as machine ClassAd attribute `TotalMemory`.

TotalSlots: A sum of the static slots, partitionable slots, and dynamic slots on the machine at the current time.

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 HTCondor 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 HTCondor 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 KiB. On Linux platforms, it is the sum of paging space and physical memory, which more accurately represents the virtual memory size of the machine.

VM_AvailNum: The maximum number of vm universe jobs that can be started on this machine. This maximum is set by the configuration variable `VM_MAX_NUMBER`.

VM_Guest_Mem: An attribute defined if a vm universe job is running on this slot. Defined by the amount of memory in use by the virtual machine, given in Mbytes.

VM_Memory: Gives the amount of memory available for starting additional VM jobs on this machine, given in Mbytes. The maximum value is set by the configuration variable `VM_MEMORY`.

VM_Networking: A boolean value indicating whether networking is allowed for virtual machines on this machine.

VM_Type: The type of virtual machine software that can run on this machine. The value is set by the configuration variable VM_TYPE.

VMOfflineReason: The reason the VM universe went offline (usually because a VM universe job failed to launch).

VMOfflineTime: The time that the VM universe went offline.

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

RemoteAutoregroup: A boolean attribute which is True if this resource was claimed via negotiation when the configuration variable GROUP_AUTOREGROUP is True. It is False otherwise.

RemoteGroup: The accounting group name corresponding to the submitter that claimed this resource.

RemoteNegotiatingGroup: The accounting group name under which this resource negotiated when it was claimed. This attribute will frequently be the same as attribute RemoteGroup, but it may differ in cases such as when configuration variable GROUP_AUTOREGROUP is True, in which case it will have the name of the root group, identified as <none>.

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 HTCondor job which is waiting for the current job to finish or be preempted. If not currently hosting an HTCondor 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.

There are a few attributes that are applicable to machines that are offline, that is, hibernating.

MachineLastMatchTime: The Unix epoch time when this offline ClassAd would have been matched to a job, if the machine were online. In addition, the slot1 ClassAd of a multi-slot machine will have `slot<X>_MachineLastMatchTime` defined, where `<X>` is replaced by the slot id of each of the slots with `MachineLastMatchTime` defined.

Offline: A boolean value, that when `True`, indicates this machine is in an offline state in the *condor_collector*. Such ClassAds are stored persistently, such that they will continue to exist after the *condor_collector* restarts.

Unhibernate: A boolean expression that specifies when a hibernating machine should be woken up, for example, by *condor_rooster*.

For machines with user-defined or custom resource specifications, including GPUs, the following attributes will be in the ClassAd for each slot. In the name of the attribute, `<name>` is substituted with the configured name given to the resource.

Assigned<name>: A space separated list that identifies which of these resources are currently assigned to slots.

Offline<name>: A space separated list that indicates which of these resources is unavailable for match making.

Total<name>: An integer quantity of the total number of these resources.

For machines with custom resource specifications that include GPUs, the following attributes may be in the ClassAd for each slot, depending on the value of configuration variable `MACHINE_RESOURCE_INVENTORY_GPUs` and what GPUs are detected. In the name of the attribute, `<name>` is substituted with the *prefix string* assigned for the GPU.

<name>BoardTempC: For NVIDIA devices, a dynamic attribute representing the temperature in Celsius of the board containing the GPU.

<name>Capability: The CUDA-defined capability for the GPU.

<name>ClockMhz: For CUDA or Open CL devices, the integer clocking speed of the GPU in MHz.

<name>ComputeUnits: For CUDA or Open CL devices, the integer number of compute units per GPU.

<name>CoresPerCU: For CUDA devices, the integer number of cores per compute unit.

<name>DeviceName: For CUDA or Open CL devices, a string representing the manufacturer's proprietary device name.

<name>DieTempC: For NVIDIA devices, a dynamic attribute representing the temperature in Celsius of the GPU die.

<name>DriverVersion: For CUDA devices, a string representing the manufacturer's driver version.

<name>ECCEnabled: For CUDA or Open CL devices, a boolean value representing whether error correction is enabled.

<name>EccErrorsDoubleBit: For NVIDIA devices, a count of the number of double bit errors detected for this GPU.

<name>EccErrorsSingleBit: For NVIDIA devices, a count of the number of single bit errors detected for this GPU.

<name>FanSpeedPct: For NVIDIA devices, a value between 0 and 100 (inclusive), used to represent the level of fan operation as percentage of full fan speed.

<name>GlobalMemoryMb: For CUDA or Open CL devices, the quantity of memory in Mbytes in this GPU.

<name>OpenCLVersion: For Open CL devices, a string representing the manufacturer's version number.

<name>RuntimeVersion: For CUDA devices, a string representing the manufacturer's version number.

The following attributes are advertised for a machine in which partitionable slot preemption is enabled.

ChildAccountingGroup: A ClassAd list containing the values of the `AccountingGroup` attribute for each dynamic slot of the partitionable slot.

ChildActivity: A ClassAd list containing the values of the `Activity` attribute for each dynamic slot of the partitionable slot.

ChildCpus: A ClassAd list containing the values of the `Cpus` attribute for each dynamic slot of the partitionable slot.

ChildCurrentRank: A ClassAd list containing the values of the `CurrentRank` attribute for each dynamic slot of the partitionable slot.

ChildEnteredCurrentState: A ClassAd list containing the values of the `EnteredCurrentState` attribute for each dynamic slot of the partitionable slot.

ChildMemory: A ClassAd list containing the values of the `Memory` attribute for each dynamic slot of the partitionable slot.

ChildName: A ClassAd list containing the values of the `Name` attribute for each dynamic slot of the partitionable slot.

ChildRemoteOwner: A ClassAd list containing the values of the `RemoteOwner` attribute for each dynamic slot of the partitionable slot.

ChildRemoteUser: A ClassAd list containing the values of the `RemoteUser` attribute for each dynamic slot of the partitionable slot.

ChildRetirementTimeRemaining: A ClassAd list containing the values of the `RetirementTimeRemaining` attribute for each dynamic slot of the partitionable slot.

ChildState: A ClassAd list containing the values of the `State` attribute for each dynamic slot of the partitionable slot.

PslotRollupInformation: A boolean value set to `True` in both the partitionable and dynamic slots, when configuration variable `ADVERTISE_PSLOT_ROLLUP_INFORMATION` is `True`, such that the *condor_negotiator* knows when partitionable slot preemption is possible and can directly preempt a dynamic slot when appropriate.

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.

CondorVersion: A string containing the HTCondor version number, the release date, and the build identification number.

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.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in Kbytes.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in Kbytes.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

MyAddress: String with the IP and port address of the *condor_master* daemon which is publishing this ClassAd.

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

Autoclusters: A Statistics attribute defining the number of active autoclusters.

CollectorHost: The name of the main *condor_collector* which this *condor_schedd* daemon reports to, as copied from `COLLECTOR_HOST`. If a *condor_schedd* flocks to other *condor_collector* daemons, this attribute still represents the "home" *condor_collector*, so this value can be used to discover if a *condor_schedd* is currently flocking.

CondorVersion: A string containing the HTCondor version number, the release date, and the build identification number.

DaemonCoreDutyCycle: A Statistics attribute defining the ratio of the time spent handling messages and events to the elapsed time for the time period defined by `StatsLifetime` of this *condor_schedd*. A value near 0.0 indicates an idle daemon, while a value near 1.0 indicates a daemon running at or above capacity.

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).

DetectedCpus: The number of detected machine CPUs/cores.

DetectedMemory: The amount of detected machine RAM in MBytes.

JobQueueBirthdate: Description is not yet written.

JobsAccumBadputTime: A Statistics attribute defining the sum of the all of the time jobs which did not complete successfully have spent running over the lifetime of this *condor_schedd*.

JobsAccumExceptionalBadputTime: A Statistics attribute defining the sum of the all of the time jobs which did not complete successfully due to *condor_shadow* exceptions have spent running over the lifetime of this *condor_schedd*.

JobsAccumRunningTime: A Statistics attribute defining the sum of the all of the time jobs have spent running in the time interval defined by attribute `StatsLifetime`.

JobsAccumTimeToStart: A Statistics attribute defining the sum of all the time jobs have spent waiting to start in the time interval defined by attribute `StatsLifetime`.

JobsBadputRuntimes: A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by time spent running, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute `JobsRuntimesHistogramBuckets`.

JobsBadputSizes: A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by image size, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute `JobsSizesHistogramBuckets`.

JobsCheckpointed: A Statistics attribute defining the number of times jobs that have exited with a *condor_shadow* exit code of `JOB_CKPTED` in the time interval defined by attribute `StatsLifetime`.

JobsCompleted: A Statistics attribute defining the number of jobs successfully completed in the time interval defined by attribute `StatsLifetime`.

JobsCompletedRuntimes: A Statistics attribute defining a histogram count of jobs that completed successfully as classified by time spent running, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute `JobsRuntimesHistogramBuckets`.

JobsCompletedSizes: A Statistics attribute defining a histogram count of jobs that completed successfully as classified by image size, over the lifetime of this *condor_schedd*. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute `JobsSizesHistogramBuckets`.

JobsCoredumped: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_COREDUMPED` in the time interval defined by attribute `StatsLifetime`.

JobsDebugLogError: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `DPRINTF_ERROR` in the time interval defined by attribute `StatsLifetime`.

JobsExecFailed: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_EXEC_FAILED` in the time interval defined by attribute `StatsLifetime`.

JobsExited: A Statistics attribute defining the number of times that jobs that exited (successfully or not) in the time interval defined by attribute `StatsLifetime`.

JobsExitedAndClaimClosing: A Statistics attribute defining the number of times jobs have exited with a *condor_shadow* exit code of `JOB_EXITED_AND_CLAIM_CLOSING` in the time interval defined by attribute `StatsLifetime`.

JobsExitedNormally: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_EXITED` or with an exit code of `JOB_EXITED_AND_CLAIM_CLOSING` in the time interval defined by attribute `StatsLifetime`.

JobsExitException: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_EXCEPTION` or with an unknown status in the time interval defined by attribute `StatsLifetime`.

JobsKilled: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_KILLED` in the time interval defined by attribute `StatsLifetime`.

JobsMissedDeferralTime: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_MISSED_DEFERRAL_TIME` in the time interval defined by attribute `StatsLifetime`.

JobsNotStarted: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_NOT_STARTED` in the time interval defined by attribute `StatsLifetime`.

JobsRestartReconnectsAttempting: A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* is currently attempting to reconnect to, in order to recover a job that was running when the *condor_schedd* was restarted.

JobsRestartReconnectsBadput: A Statistics attribute defining a histogram count of *condor_startd* daemons that the *condor_schedd* could not reconnect to in order to recover a job that was running when the *condor_schedd* was restarted, as classified by the time the job spent running. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute `JobsRuntimesHistogramBuckets`.

JobsRestartReconnectsFailed: A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* tried and failed to reconnect to in order to recover a job that was running when the *condor_schedd* was restarted.

JobsRestartReconnectsInterrupted: A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* attempted to reconnect to, in order to recover a job that was running when the *condor_schedd* was restarted, but the attempt was interrupted, for example, because the job was removed.

JobsRestartReconnectsLeaseExpired: A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* could not attempt to reconnect to, in order to recover a job that was running when the *condor_schedd* was restarted, because the job lease had already expired.

JobsRestartReconnectsSucceeded: A Statistics attribute defining the number of *condor_startd* daemons the *condor_schedd* has successfully reconnected to, in order to recover a job that was running when the *condor_schedd* was restarted.

JobsRunning: A Statistics attribute representing the number of jobs currently running.

JobsRunningRuntimes: A Statistics attribute defining a histogram count of jobs currently running, as classified by elapsed runtime. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute `JobsRuntimesHistogramBuckets`.

JobsRunningSizes: A Statistics attribute defining a histogram count of jobs currently running, as classified by image size. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute `JobsSizesHistogramBuckets`.

JobsRuntimesHistogramBuckets: A Statistics attribute defining the predefined bucket boundaries for histogram statistics that classify run times. Defined as

```
JobsRuntimesHistogramBuckets = "30Sec, 1Min, 3Min, 10Min, 30Min, 1Hr, 3Hr,
        6Hr, 12Hr, 1Day, 2Day, 4Day, 8Day, 16Day"
```

JobsShadowNoMemory: A Statistics attribute defining the number of times that jobs have exited because there was not enough memory to start the *condor_shadow* in the time interval defined by attribute `StatsLifetime`.

JobsShouldHold: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_HOLD` in the time interval defined by attribute `StatsLifetime`.

JobsShouldRemove: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_REMOVE` in the time interval defined by attribute `StatsLifetime`.

JobsShouldRequeue: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_REQUEUE` in the time interval defined by attribute `StatsLifetime`.

JobsSizesHistogramBuckets: A Statistics attribute defining the predefined bucket boundaries for histogram statistics that classify image sizes. Defined as

```
JobsSizesHistogramBuckets = "64Kb, 256Kb, 1Mb, 4Mb, 16Mb, 64Mb, 256Mb,
        1Gb, 4Gb, 16Gb, 64Gb, 256Gb"
```

Note that these values imply powers of two in numbers of bytes.

JobsStarted: A Statistics attribute defining the number of jobs started in the time interval defined by attribute `StatsLifetime`.

JobsSubmitted: A Statistics attribute defining the number of jobs submitted in the time interval defined by attribute `StatsLifetime`.

Machine: A string with the machine's fully qualified host name.

MaxJobsRunning: The same integer value as set by the evaluation of the configuration variable `MAX_JOBS_RUNNING`. See the definition at section 3.5.9 on page 262.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in Kbytes.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in Kbytes.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

MyAddress: String with the IP and port address of the *condor_schedd* daemon which is publishing this ClassAd.

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.

NumJobStartsDelayed: The number times a job requiring a *condor_shadow* daemon could have been started, but was not started because of the values of configuration variables `JOB_START_COUNT` and `JOB_START_DELAY`.

NumPendingClaims: The number of machines (*condor_startd* daemons) matched to this *condor_schedd* daemon, which this *condor_schedd* knows about, but has not yet managed to claim.

NumUsers: The integer number of distinct users with jobs in this *condor_schedd*'s queue.

PublicNetworkIpAddr: Description is not yet written.

RecentDaemonCoreDutyCycle: A Statistics attribute defining the ratio of the time spent handling messages and events to the elapsed time in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsAccumBadputTime: A Statistics attribute defining the sum of the all of the time that jobs which did not complete successfully have spent running in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsAccumRunningTime: A Statistics attribute defining the sum of the all of the time jobs which have exited in the previous time interval defined by attribute `RecentStatsLifetime` spent running.

RecentJobsAccumTimeToStart: A Statistics attribute defining the sum of all the time jobs which have exited in the previous time interval defined by attribute `RecentStatsLifetime` had spent waiting to start.

RecentJobsBadputRuntimes: A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by time spent running, in the previous time interval defined by attribute `RecentStatsLifetime`. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute `JobsRuntimesHistogramBuckets`.

RecentJobsBadputSizes: A Statistics attribute defining a histogram count of jobs that did not complete successfully, as classified by image size, in the previous time interval defined by attribute `RecentStatsLifetime`. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute `JobsSizesHistogramBuckets`.

RecentJobsCheckpointed: A Statistics attribute defining the number of times jobs that have exited with a *condor_shadow* exit code of JOB_CKPTED in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsCompleted: A Statistics attribute defining the number of jobs successfully completed in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsCompletedRuntimes: A Statistics attribute defining a histogram count of jobs that completed successfully, as classified by time spent running, in the previous time interval defined by attribute RecentStatsLifetime. Counts within the histogram are separated by a comma and a space, where the time interval classification is defined in the ClassAd attribute JobsRuntimesHistogramBuckets.

RecentJobsCompletedSizes: A Statistics attribute defining a histogram count of jobs that completed successfully, as classified by image size, in the previous time interval defined by attribute RecentStatsLifetime. Counts within the histogram are separated by a comma and a space, where the size classification is defined in the ClassAd attribute JobsSizesHistogramBuckets.

RecentJobsCoredumped: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_COREDUMPED in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsDebugLogError: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of DPRINTF_ERROR in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsExecFailed: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXEC_FAILED in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsExited: A Statistics attribute defining the number of times that jobs have exited normally in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsExitedAndClaimClosing: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXITED_AND_CLAIM_CLOSING in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsExitedNormally: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXITED or with an exit code of JOB_EXITED_AND_CLAIM_CLOSING in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsExitException: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_EXCEPTION or with an unknown status in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsKilled: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_KILLED in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsMissedDeferralTime: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of JOB_MISSED_DEFERRAL_TIME in the previous time interval defined by attribute RecentStatsLifetime.

RecentJobsNotStarted: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_NOT_STARTED` in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsShadowNoMemory: A Statistics attribute defining the number of times that jobs have exited because there was not enough memory to start the *condor_shadow* in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsShouldHold: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_HOLD` in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsShouldRemove: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_REMOVE` in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsShouldRequeue: A Statistics attribute defining the number of times that jobs have exited with a *condor_shadow* exit code of `JOB_SHOULD_REQUEUE` in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsStarted: A Statistics attribute defining the number of jobs started in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentJobsSubmitted: A Statistics attribute defining the number of jobs submitted in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentShadowsReconnections: A Statistics attribute defining the number of times that *condor_shadow* daemons lost connection to their *condor_starter* daemons and successfully reconnected in the previous time interval defined by attribute `RecentStatsLifetime`. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable `STATISTICS_TO_PUBLISH` is set to 2 or higher.

RecentShadowsRecycled: A Statistics attribute defining the number of times *condor_shadow* processes have been recycled for use with a new job in the previous time interval defined by attribute `RecentStatsLifetime`. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable `STATISTICS_TO_PUBLISH` is set to 2 or higher.

RecentShadowsStarted: A Statistics attribute defining the number of *condor_shadow* daemons started in the previous time interval defined by attribute `RecentStatsLifetime`.

RecentStatsLifetime: A Statistics attribute defining the time in seconds over which statistics values have been collected for attributes with names that begin with `Recent`. This value starts at 0, and it may grow to a value as large as the value defined for attribute `RecentWindowMax`.

RecentStatsTickTime: A Statistics attribute defining the time that attributes with names that begin with `Recent` were last updated, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable `STATISTICS_TO_PUBLISH` is set to 2 or higher.

RecentWindowMax: A Statistics attribute defining the maximum time in seconds over which attributes with names that begin with `Recent` are collected. The value is set by the configuration variable

STATISTICS_WINDOW_SECONDS, which defaults to 1200 seconds (20 minutes). This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

ScheddIpAddr: String with the IP and port address of the *condor_schedd* daemon which is publishing this Scheduler ClassAd.

ServerTime: Description is not yet written.

ShadowsReconnections: A Statistics attribute defining the number of times *condor_shadow*s lost connection to their *condor_starter*s and successfully reconnected in the previous StatsLifetime seconds. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

ShadowsRecycled: A Statistics attribute defining the number of times *condor_shadow* processes have been recycled for use with a new job in the previous StatsLifetime seconds. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

ShadowsRunning: A Statistics attribute defining the number of *condor_shadow* daemons currently running that are owned by this *condor_schedd*.

ShadowsRunningPeak: A Statistics attribute defining the maximum number of *condor_shadow* daemons running at one time that were owned by this *condor_schedd* over the lifetime of this *condor_schedd*.

ShadowsStarted: A Statistics attribute defining the number of *condor_shadow* daemons started in the previous time interval defined by attribute StatsLifetime.

StartLocalUniverse: The same boolean value as set in the configuration variable START_LOCAL_UNIVERSE. See the definition at section 3.5.9 on page 261.

StartSchedulerUniverse: The same boolean value as set in the configuration variable START_SCHEDULER_UNIVERSE. See the definition at section 3.5.9 on page 262.

StatsLastUpdateTime: A Statistics attribute defining the time that statistics about jobs were last updated, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970). This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

StatsLifetime: A Statistics attribute defining the time in seconds over which statistics have been collected for attributes with names that do *not* begin with Recent. This statistic only appears in the Scheduler ClassAd if the level of verbosity set by the configuration variable STATISTICS_TO_PUBLISH is set to 2 or higher.

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, not including local or scheduler universe jobs.

TotalJobAds: The total number of all jobs (in all states) from this *condor_schedd* daemon.

TotalLocalJobsIdle: The total number of **local universe** jobs from this *condor_schedd* daemon that are currently idle.

TotalLocalJobsRunning: The total number of **local universe** jobs from this *condor_schedd* daemon that are currently running.

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, not including local or scheduler universe jobs.

TotalSchedulerJobsIdle: The total number of **scheduler universe** jobs from this *condor_schedd* daemon that are currently idle.

TotalSchedulerJobsRunning: The total number of **scheduler universe** jobs from this *condor_schedd* daemon that are currently running.

TransferQueueUserExpr A ClassAd expression that provides the name of the transfer queue that the *condor_schedd* will be using for job file transfer.

UpdateInterval: The interval, in seconds, between publication of this *condor_schedd* ClassAd and the previous publication.

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.

When using file transfer concurrency limits, the following additional I/O usage statistics are published. These includes the sum and rate of bytes transferred as well as time spent reading and writing to files and to the network. These statistics are reported for the sum of all users and may also be reported individually for recently active users by increasing the verbosity level `STATISTICS_TO_PUBLISH = TRANSFER:2`. Each of the per-user statistics is prefixed by a user name in the form `Owner_<username>_FileTransferUploadBytes`. In this case, the attribute represents activity by the specified user. The published user name is actually the file transfer queue name, as defined by configuration variable `TRANSFER_QUEUE_USER_EXPR`. This expression defaults to `Owner_` followed by the name of the job owner. The attributes that are rates have a suffix that specifies the time span of the exponential moving average. By default the time spans that are published are 1m, 5m, 1h, and 1d. This can be changed by configuring configuration variable `TRANSFER_IO_REPORT_TIMESPANS`. These attributes are only reported once a full time span has accumulated.

FileTransferDiskThrottleExcess_<timespan> The exponential moving average of the disk load that exceeds the upper limit set for the disk load throttle. Periods of time in which there is no excess and no waiting transfers do not contribute to the average. This attribute is published only if configuration variable `FILE_TRANSFER_DISK_LOAD_THROTTLE` is defined.

FileTransferDiskThrottleHigh The desired upper limit for the disk load from file transfers, as configured by FILE_TRANSFER_DISK_LOAD_THROTTLE. This attribute is published only if configuration variable FILE_TRANSFER_DISK_LOAD_THROTTLE is defined.

FileTransferDiskThrottleLevel The current concurrency limit set by the disk load throttle. The limit is applied to the sum of uploads and downloads. This attribute is published only if configuration variable FILE_TRANSFER_DISK_LOAD_THROTTLE is defined.

FileTransferDiskThrottleLow The lower limit for the disk load from file transfers, as configured by FILE_TRANSFER_DISK_LOAD_THROTTLE. This attribute is published only if configuration variable FILE_TRANSFER_DISK_LOAD_THROTTLE is defined.

FileTransferDiskThrottleShortfall_<timespan> The exponential moving average of the disk load that falls below the upper limit set for the disk load throttle. Periods of time in which there is no excess and no waiting transfers do not contribute to the average. This attribute is published only if configuration variable FILE_TRANSFER_DISK_LOAD_THROTTLE is defined.

FileTransferDownloadBytes Total number of bytes downloaded as output from jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferDownloadBytes. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferDownloadBytesPerSecond_<timespan> Exponential moving average over the specified time span of the rate at which bytes have been downloaded as output from jobs. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferDownloadBytesPerSecond_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferFileReadLoad_<timespan> Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time reading from files to be transferred as input to jobs. One file transfer process spending nearly all of its time reading files will generate a load close to 1.0. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileReadLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferFileReadSeconds Total number of submit-side transfer process seconds spent reading from files to be transferred as input to jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileReadSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferFileWriteLoad_<timespan> Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time writing to files transferred as output from jobs. One file transfer process spending nearly all of its time writing to files will generate a load close to 1.0. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileWriteLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferFileWriteSeconds Total number of submit-side transfer process seconds spent writing to files transferred as output from jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferFileWriteSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferNetReadLoad_<timespan> Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time reading from the network when transferring output from jobs. One file transfer process spending nearly all of its time reading from the network will generate a load close to 1.0. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow reads from the disk on the execute side. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetReadLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferNetReadSeconds Total number of submit-side transfer process seconds spent reading from the network when transferring output from jobs since this *condor_schedd* was started. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow reads from the disk on the execute side. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetReadSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferNetWriteLoad_<timespan> Exponential moving average over the specified time span of the rate at which submit-side file transfer processes have spent time writing to the network when transferring input to jobs. One file transfer process spending nearly all of its time writing to the network will generate a load close to 1.0. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow writes to the disk on the execute side. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetWriteLoad_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferNetWriteSeconds Total number of submit-side transfer process seconds spent writing to the network when transferring input to jobs since this *condor_schedd* was started. The reason a file transfer process may spend a long time writing to the network could be a network bottleneck on the path between the submit and execute machine. It could also be caused by slow writes to the disk on the execute side. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferNetWriteSeconds. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferUploadBytes Total number of bytes uploaded as input to jobs since this *condor_schedd* was started. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferUploadBytes. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

FileTransferUploadBytesPerSecond_<timespan> Exponential moving average over the specified time span of the rate at which bytes have been uploaded as input to jobs. The time spans that are published are configured by TRANSFER_IO_REPORT_TIMESPANS, which defaults to 1m, 5m, 1h, and 1d. When less than one full time span has accumulated, the attribute is not published. If STATISTICS_TO_PUBLISH contains TRANSFER:2, for each active user, this attribute is also published prefixed by the user name, with the name Owner_<username>_FileTransferUploadBytesPerSecond_<timespan>. The published user name is actually the file transfer queue name, as defined by configuration variable TRANSFER_QUEUE_USER_EXPR.

TransferQueueMBWaitingToDownload Number of megabytes of output files waiting to be downloaded.

TransferQueueMBWaitingToUpload Number of megabytes of input files waiting to be uploaded.

TransferQueueNumWaitingToDownload Number of jobs waiting to transfer output files.

TransferQueueNumWaitingToUpload Number of jobs waiting to transfer input files.

Negotiator ClassAd Attributes

CondorVersion: A string containing the HTCondor version number, the release date, and the build identification number.

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).

LastNegotiationCycleActiveSubmitterCount<X>: The integer number of submitters the *condor_negotiator* attempted to negotiate with in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleCandidateSlots<X>: The number of slot ClassAds after filtering by NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT. This is the number of slots actually considered for matching. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleDuration<X>: The number of seconds that it took to complete the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleEnd<X>: The time, represented as the number of seconds since the Unix epoch, at which the negotiation cycle ended. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleMatches<X>: The number of successful matches that were made in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleMatchRate<X>: The number of matched jobs divided by the duration of this cycle giving jobs per second. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleMatchRateSustained<X>: The number of matched jobs divided by the period of this cycle giving jobs per second. The period is the time elapsed between the end of the previous cycle and the end of this cycle, and so this rate includes the interval between cycles. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleNumIdleJobs<X>: The number of idle jobs considered for matchmaking. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleNumJobsConsidered<X>: The number of jobs requests returned from the schedulers for consideration. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleNumSchedulers<X>: The number of individual schedulers negotiated with during matchmaking. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCyclePeriod<X>: The number of seconds elapsed between the end of the previous negotiation cycle and the end of this cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCyclePhase1Duration<X>: The duration, in seconds, of Phase 1 of the negotiation cycle: the process of getting submitter and machine ClassAds from the *condor_collector*. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCyclePhase2Duration<X>: The duration, in seconds, of Phase 2 of the negotiation cycle: the process of filtering slots and processing accounting group configuration. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCyclePhase3Duration<X>: The duration, in seconds, of Phase 3 of the negotiation cycle: sorting submitters by priority. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCyclePhase4Duration<X>: The duration, in seconds, of Phase 4 of the negotiation cycle: the process of matching slots to jobs in conjunction with the schedulers. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleRejections<X>: The number of rejections that occurred in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleSlotShareIter<X>: The number of iterations performed during the negotiation cycle. Each iteration includes the reallocation of remaining slots to accounting groups, as defined by the implementation of hierarchical group quotas, together with the negotiation for those slots. The maximum number of iterations is limited by the configuration variable GROUP_QUOTA_MAX_ALLOCATION_ROUNDS. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleSubmittersFailed<X>: A string containing a space and comma-separated list of the names of all submitters who failed to negotiate in the negotiation cycle. One possible cause of failure is a communication timeout. This list does not include submitters who ran out of time due to NEGOTIATOR_MAX_TIME_PER_SUBMITTER. Those are listed separately in LastNegotiationCycleSubmittersOutOfTime<X>. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleSubmittersOutOfTime<X>: A string containing a space and comma separated list of the names of all submitters who ran out of time due to NEGOTIATOR_MAX_TIME_PER_SUBMITTER in the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleSubmittersShareLimit: A string containing a space and comma separated list of names of submitters who encountered their fair-share slot limit during the negotiation cycle. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleTime<X>: The time, represented as the number of second elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the negotiation cycle started. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleTotalSlots<X>: The total number of slot ClassAds received by the *condor_negotiator*. The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

LastNegotiationCycleTrimmedSlots<X>: The number of slot ClassAds left after trimming currently claimed slots (when enabled). The number <X> appended to the attribute name indicates how many negotiation cycles ago this cycle happened.

Machine: A string with the machine's fully qualified host name.

MyAddress: String with the IP and port address of the *condor_negotiator* daemon which is publishing this ClassAd.

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.

Submitter ClassAd Attributes

CondorVersion: A string containing the HTCondor version number, the release date, and the build identification number.

FlockedJobs: The number of jobs from this submitter that are running in another pool.

HeldJobs: The number of jobs from this submitter that are in the hold state.

IdleJobs: The number of jobs from this submitter that are now idle. Scheduler and Local universe jobs are not included in this count.

LocalJobsIdle: The number of Local universe jobs from this submitter that are now idle.

LocalJobsRunning: The number of Local universe jobs from this submitter that are running.

MyAddress: The IP address associated with the *condor_schedd* daemon used by the submitter.

Name: The fully qualified name of the user or accounting group. It will be of the form `name@submit.domain`.

RunningJobs: The number of jobs from this submitter that are running now. Scheduler and Local universe jobs are not included in this count.

ScheddIpAddr: The IP address associated with the *condor_schedd* daemon used by the submitter. This attribute is obsolete Use MyAddress instead.

ScheddName: The fully qualified host name of the machine that the submitter submitted from. It will be of the form `submit.domain`.

SchedulerJobsIdle: The number of Scheduler universe jobs from this submitter that are now idle.

SchedulerJobsRunning: The number of Scheduler universe jobs from this submitter that are running.

SubmitterTag: The fully qualified host name of the central manager of the pool used by the submitter, if the job flocked to the local pool. Or, it will be the empty string if submitter submitted from within the local pool.

WeightedIdleJobs: A total number of requested cores across all Idle jobs from the submitter, weighted by the slot weight. As an example, if `SLOT_WEIGHT = CPUS`, and a job requests two CPUs, the weight of that job is two.

WeightedRunningJobs: A total number of requested cores across all Running jobs from the submitter.

Defrag ClassAd Attributes

AvgDrainingBadput: Fraction of time CPUs in the pool have spent on jobs that were killed during draining of the machine. This is calculated in each polling interval by looking at `TotalMachineDrainingBadput`. Therefore, it treats evictions of jobs that do and do not produce checkpoints the same. When the *condor_startd* restarts, its counters start over from 0, so the average is only over the time since the daemons have been alive.

AvgDrainingUnclaimedTime: Fraction of time CPUs in the pool have spent unclaimed by a user during draining of the machine. This is calculated in each polling interval by looking at `TotalMachineDrainingUnclaimedTime`. When the *condor_startd* restarts, its counters start over from 0, so the average is only over the time since the daemons have been alive.

DaemonStartTime: The time that this daemon was started, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

DrainedMachines: A count of the number of fully drained machines which have arrived during the run time of this *condor_defrag* daemon.

DrainFailures: Total count of failed attempts to initiate draining during the lifetime of this *condor_defrag* daemon.

DrainSuccesses: Total count of successful attempts to initiate draining during the lifetime of this *condor_defrag* daemon.

Machine: A string with the machine's fully qualified host name.

MachinesDraining: Number of machines that were observed to be draining in the last polling interval.

MachinesDrainingPeak: Largest number of machines that were ever observed to be draining.

MeanDrainedArrived: The mean time in seconds between arrivals of fully drained machines.

MonitorSelfAge: The number of seconds that this daemon has been running.

MonitorSelfCPUUsage: The fraction of recent CPU time utilized by this daemon.

MonitorSelfImageSize: The amount of virtual memory consumed by this daemon in KiB.

MonitorSelfRegisteredSocketCount: The current number of sockets registered by this daemon.

MonitorSelfResidentSetSize: The amount of resident memory used by this daemon in KiB.

MonitorSelfSecuritySessions: The number of open (cached) security sessions for this daemon.

MonitorSelfTime: The time, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which this daemon last checked and set the attributes with names that begin with the string `MonitorSelf`.

MyAddress: String with the IP and port address of the *condor_defrag* daemon which is publishing this ClassAd.

MyCurrentTime: The time, represented as the number of seconds elapsed since the Unix epoch (00:00:00 UTC, Jan 1, 1970), at which the *condor_defrag* daemon last sent a ClassAd update to the *condor_collector*.

Name: The name of this daemon; typically the same value as the `Machine` attribute, but could be customized by the site administrator via the configuration variable `DEFRAG_NAME`.

RecentDrainFailures: Count of failed attempts to initiate draining during the past `RecentStatsLifetime` seconds.

RecentDrainSuccesses: Count of successful attempts to initiate draining during the past `RecentStatsLifetime` seconds.

RecentStatsLifetime: A Statistics attribute defining the time in seconds over which statistics values have been collected for attributes with names that begin with `Recent`.

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.

WholeMachines: Number of machines that were observed to be defragmented in the last polling interval.

WholeMachinesPeak: Largest number of machines that were ever observed to be simultaneously defragmented.

Collector ClassAd Attributes

CollectorIpAddr: String with the IP and port address of the *condor_collector* daemon which is publishing this ClassAd.

CondorVersion: A string containing the HTCondor version number, the release date, and the build identification number.

CurrentJobsRunningAll: An integer value representing the sum of all jobs running under all universes.

CurrentJobsRunning<universe>: An integer value representing the current number of jobs running under the universe which forms the attribute name. For example

```
CurrentJobsRunningVanilla = 567
```

identifies that the *condor_collector* counts 567 vanilla universe jobs currently running. `<universe>` is one of `Unknown`, `Standard`, `Vanilla`, `Scheduler`, `Java`, `Parallel`, `VM`, or `Local`. There are other universes, but they are not listed here, as they represent ones that are no longer used in Condor.

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).

HostsClaimed: Description is not yet written.

HostsOwner: Description is not yet written.

HostsTotal: Description is not yet written.

HostsUnclaimed: Description is not yet written.

IdleJobs: Description is not yet written.

Machine: A string with the machine's fully qualified host name.

MaxJobsRunning<universe: An integer value representing the sum of all `MaxJobsRunning<universe>` values.

MaxJobsRunning<universe>: An integer value representing largest number of currently running jobs ever seen under the universe which forms the attribute name, over the life of this *condor_collector* process. For example

```
MaxJobsRunningVanilla = 401
```

identifies that the *condor_collector* saw 401 vanilla universe jobs currently running at one point in time, and that was the largest number it had encountered. `<universe>` is one of `Unknown`, `Standard`, `Vanilla`, `Scheduler`, `Java`, `Parallel`, `VM`, or `Local`. There are other universes, but they are not listed here, as they represent ones that are no longer used in Condor.

MyAddress: String with the IP and port address of the *condor_collector* daemon which is publishing this ClassAd.

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.

RunningJobs: Definition not yet written.

StartdAds: The integer number of unique *condor_startd* daemon ClassAds counted at the most recent time the *condor_collector* updated its own ClassAd.

StartdAdsPeak: The largest integer number of unique *condor_startd* daemon ClassAds seen at any one time, since the *condor_collector* began executing.

SubmitterAds: The integer number of unique submitters counted at the most recent time the *condor_collector* updated its own ClassAd.

SubmitterAdsPeak: The largest integer number of unique submitters seen at any one time, since the *condor_collector* began executing.

UpdateInterval: Description is not yet written.

UpdateSequenceNumber: An integer that begins at 0, and increments by one each time the same ClassAd is again advertised.

UpdatesInitial: A Statistics attribute representing a count of unique ClassAds seen, over the lifetime of this *condor_collector*. Counts per ClassAd are advertised in attributes named by ClassAd type as `UpdatesInitial_<ClassAd-Name>`. `<ClassAd-Name>` is each of `CkptSrvr`, `Collector`, `Defrag`, `Master`, `Schedd`, `Start`, `StartdPvt`, and `Submittor`.

UpdatesLost: A Statistics attribute representing the count of updates lost, over the lifetime of this *condor_collector*. Counts per ClassAd are advertised in attributes named by ClassAd type as `UpdatesLost_<ClassAd-Name>`. `<ClassAd-Name>` is each of `CkptSrvr`, `Collector`, `Defrag`, `Master`, `Schedd`, `Start`, `StartdPvt`, and `Submittor`.

UpdatesLostMax: A Statistics attribute defining the largest number of updates lost at any point in time, over the lifetime of this *condor_collector*. ClassAd sequence numbers are used to detect lost ClassAds.

UpdatesLostRatio: A Statistics attribute defining the floating point ratio of the total number of updates to the number of updates lost over the lifetime of this *condor_collector*. ClassAd sequence numbers are used to detect lost ClassAds. A value of 1 indicates that all ClassAds have been lost.

UpdatesTotal: A Statistics attribute representing the count of the number of ClassAd updates received over the lifetime of this *condor_collector*. Counts per ClassAd are advertised in attributes named by ClassAd type as `UpdatesTotal_<ClassAd-Name>`. `<ClassAd-Name>` is each of `CkptSrvr`, `Collector`, `Defrag`, `Master`, `Schedd`, `Start`, `StartdPvt`, and `Submittor`.

ClassAd Attributes Added by the *condor_collector*

AuthenticatedIdentity: The authenticated name assigned by the *condor_collector* to the daemon that published the ClassAd.

AuthenticationMethod: The authentication method used by the *condor_collector* to determine the `AuthenticatedIdentity`.

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). This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

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 289 for more information on this setting. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

UpdatesLost: An integer count of the number of updates from the daemon that the *condor_collector* can definitively determine were lost since the *condor_collector* started running. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

UpdatesSequenced: An integer count of the number of updates received from the daemon, for which the *condor_collector* can tell how many were or were not lost, since the *condor_collector* started running. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

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. This attribute is added if `COLLECTOR_DAEMON_STATS` is `True`.

DaemonCore Statistics Attributes

DebugOuts: Description not yet written.

PipeMessages: Description not yet written.

PipeRuntime: Description not yet written.

SelectWaittime: Description not yet written.

SignalRuntime: Description not yet written.

Signals: Description not yet written.

SocketRuntime: Description not yet written.

SockMessages: Description not yet written.

TimerRuntime: Description not yet written.

TimersFired: Description not yet written.

Chapter 13

Appendix B: Codes and Other Needed Values

When a *condor_shadow* daemon exits, the *condor_shadow* exit code is recorded in the *condor_schedd* log, and it identifies why the job exited. Prose in the log appears of the form

```
Shadow pid XXXXX for job XX.X exited with status YYY
```

where `YYY` is the exit code, or

```
Shadow pid XXXXX for job XX.X reports job exit reason 100.
```

where the exit code is the value 100. Table 13.1 lists these codes.

Table 13.2 lists codes that appear as the first field within a job event log file. See more detailed descriptions of these values in section 2.6.7.

Table 13.1: *condor_shadow* Exit Codes

Value	*Error Name*	*Description*
4	JOB_EXCEPTION	the job exited with an exception
44	DPRINTF_ERROR	there was a fatal error with dprintf()
100	JOB_EXITED	the job exited (not killed)
101	JOB_CKPTED	the job did produce a checkpoint
102	JOB_KILLED	the job was killed
103	JOB_COREDUMPED	the job was killed and a core file was produced
105	JOB_NO_MEM	not enough memory to start the condor_shadow
106	JOB_SHADOW_USAGE	incorrect arguments to condor_shadow
107	JOB_NOT_CKPTED	the job vacated without a checkpoint
107	JOB_SHOULD_REQUEUE	same number as JOB_NOT_CKPTED, to achieve the same behavior. This exit code implies that we want the job to be put back in the job queue and run again.
108	JOB_NOT_STARTED	can not connect to the *condor_startd* or request refused
109	JOB_BAD_STATUS	job status != RUNNING on start up
110	JOB_EXEC_FAILED	exec failed for some reason other than ENOMEM
111	JOB_NO_CKPT_FILE	there is no checkpoint file (as it was lost)
112	JOB_SHOULD_HOLD	the job should be put on hold
113	JOB_SHOULD_REMOVE	the job should be removed
114	JOB_MISSED_DEFERRAL_TIME	the job goes on hold, because it did not run within the specified window of time
115	JOB_EXITED_AND_CLAIM_CLOSING	the job exited (not killed) but the *condor_startd* is not accepting any more jobs on this claim

Table 13.2: Event Codes in a Job Event Log

Event Code	*Description*
000	Submit
001	Execute
002	Executable error
003	Checkpointed
004	Job evicted
005	Job terminated
006	Image size
007	Shadow exception
008	Generic
009	Job aborted
010	Job suspended
011	Job unsuspended
012	Job held
013	Job released
014	Node execute
015	Node terminated
016	Post script terminated
017	Globus submit (no longer used)
018	Globus submit failed
019	Globus resource up (no longer used)
020	Globus resource down (no longer used)
021	Remote error
022	Job disconnected
023	Job reconnected
024	Job reconnect failed
025	Grid resource up
026	Grid resource down
027	Grid submit
028	Job ClassAd attribute values added to event log
029	Job status unknown
030	Job status known
031	Grid job stage in
032	Grid job stage out
033	Job ClassAd attribute update
034	DAGMan PRE_SKIP defined

Table 13.3: Well-Known Port Numbers

Server	*Port Number*
condor_negotiator	9614 (obsolete, now dynamically allocated)
condor_collector	9618
GT2 gatekeeper	2119
gridftp	2811
GT4 web services	8443

Table 13.4: DaemonCore Commands

Number	*Name*
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
60018	DC_PURGE_LOG

Table 13.5: DaemonCore Daemon Exit Codes

Exit Code	*Description*
0	Normal exit of daemon
99	`DAEMON_SHUTDOWN` evaluated to `True`

Index

-maxidle macro, 729
<DaemonName>_ENVIRONMENT macro, 239
<Keyword>_HOOK_EVICT_CLAIM macro, 334, 540
<Keyword>_HOOK_FETCH_WORK macro, 334, 539, 540, 542
<Keyword>_HOOK_JOB_CLEANUP macro, 335, 546
<Keyword>_HOOK_JOB_EXIT macro, 334, 541, 542
<Keyword>_HOOK_JOB_EXIT_TIMEOUT macro, 334
<Keyword>_HOOK_JOB_FINALIZE macro, 335, 546
<Keyword>_HOOK_PREPARE_JOB macro, 334, 540, 992
<Keyword>_HOOK_REPLY_CLAIM macro, 334
<Keyword>_HOOK_REPLY_FETCH macro, 334, 540
<Keyword>_HOOK_TRANSLATE_JOB macro, 335, 546, 588
<Keyword>_HOOK_UPDATE_JOB_INFO macro, 334, 541, 542, 546
<NAME>_LIMIT macro, 295
<Name>Provisioned
    job ClassAd attribute, 1004
<SUBSYS>_CLASSAD_USER_MAP_NAMES macro, 217
<SUBSYS>_DEBUG macro, 550
<SUBSYS>_LOCK macro, 550
<SUBSYS>_LOG macro, 550
<SUBSYS>_LOG_KEEP_OPEN macro, 550
<none> group, 348
<var>_ATTRS macro, 730
<var>_EXPRS macro, 730
$
    as a literal character in a submit description file, 930
$ENV
    in submit description file, 931
$RANDOM_CHOICE()
    in submit description file, 931
$RANDOM_CHOICE() function macro, 27, 197
$RANDOM_INTEGER()
    in configuration, 27, 197
$$
    as literal characters in a submit description file, 931
_CONDOR_JOB_AD environment variable, 42
_CONDOR_JOB_IWD environment variable, 42
_CONDOR_MACHINE_AD environment variable, 42
_CONDOR_SCRATCH_DIR environment variable, 42
_CONDOR_SLOT environment variable, 42
_CONDOR_WRAPPER_ERROR_FILE environment variable, 43

ABORT_ON_EXCEPTION macro, 213
Absent
    job ClassAd attribute, 986
absent ClassAd, 454
ABSENT_EXPIRE_ADS_AFTER macro, 290, 455
ABSENT_REQUIREMENTS macro, 290, 455
ABSENT_SUBMITTER_LIFETIME macro, 266
ABSENT_SUBMITTER_UPDATE_RATE macro, 266
ACCOUNTANT_LOCAL_DOMAIN macro, 291
accounting
    by group, 347
AcctGroup
    job ClassAd attribute, 986
AcctGroupUser
    job ClassAd attribute, 986
activities and state figure, 360
activity
    of a machine, 359
    transitions, 360–370
    transitions summary, 368
ADD_SIGNIFICANT_ATTRIBUTES macro, 274
ADD_WINDOWS_FIREWALL_EXCEPTION macro, 243
administrator's manual, 157–502
ADVERTISE_IPV4_FIRST macro, 217

ADVERTISE_PSLOT_ROLLUP_INFORMATION macro, 245
AFS
interaction with, 154
AfterHours macro, 373
agents
condor_shadow, 14
ALIVE_INTERVAL macro, 248, 267, 358
ALL_DEBUG macro, 223
ALLOW_* macros macro, 417
ALLOW_ADMIN_COMMANDS macro, 243
ALLOW_ADMINISTRATOR macro, 415
ALLOW_ADVERTISE_MASTER macro, 415
ALLOW_ADVERTISE_SCHEDD macro, 415
ALLOW_ADVERTISE_STARTD macro, 415
ALLOW_CLIENT macro, 317, 397
ALLOW_CLIENT macro, 415
ALLOW_CONFIG macro, 644
ALLOW_CONFIG macro, 415
ALLOW_DAEMON macro, 415
ALLOW_NEGOTIATOR macro, 415
ALLOW_OWNER macro, 415
ALLOW_PSLOT_PREEMPTION macro, 295, 725
ALLOW_READ macro, 415
ALLOW_SCRIPTS_TO_RUN_AS_EXECUTABLES macro, 214
ALLOW_SOAP macro, 415
ALLOW_VM_CRUFT macro, 42, 256, 1016
ALLOW_WRITE macro, 163
ALLOW_WRITE macro, 415
AllRemoteHosts
job ClassAd attribute, 986
ALWAYS_REUSEADDR macro, 233
ALWAYS_USE_LOCAL_CKPT_SERVER macro, 238
ALWAYS_VM_UNIV_USE_NOBODY macro, 323
Amazon EC2 Query API, 570
API
Chirp, 613
Command line, 613
DRMAA, 602
Perl module, 613
Python bindings, 621
ReadUserLog class, 603
Web Service, 590
APPEND_PREF_STANDARD macro, 284
APPEND_PREF_VANILLA macro, 284
APPEND_RANK macro, 284
APPEND_RANK_STANDARD macro, 284
APPEND_RANK_VANILLA macro, 284
APPEND_REQ_STANDARD macro, 284
APPEND_REQ_VANILLA macro, 284
APPEND_REQUIREMENTS macro, 284
ARCH macro, 200
Args
job ClassAd attribute, 986
Arguments
job ClassAd attribute, 986
argv[0]
HTCondor use of, 156
ASSIGN_CPU_AFFINITY macro, 280
AUTH_SSL_CLIENT_CADIR macro, 320, 406
AUTH_SSL_CLIENT_CAFILE macro, 320, 406
AUTH_SSL_CLIENT_CERTFILE macro, 321, 406
AUTH_SSL_CLIENT_KEYFILE macro, 321, 406
AUTH_SSL_SERVER_CADIR macro, 320, 406
AUTH_SSL_SERVER_CAFILE macro, 320, 406
AUTH_SSL_SERVER_CERTFILE macro, 320, 406
AUTH_SSL_SERVER_KEYFILE macro, 321, 406
authentication, 401–410
GSI, 403
Kerberos, 406
Kerberos principal, 407
SSL, 406
unified map file, 404, 410
using a file system, 410
using a remote file system, 410
Windows, 410
authorization
for security, 414
of Unix netgroups, 417
AUTO_INCLUDE_SHARED_PORT_IN_DAEMON_LIST macro, 229
automatic variables
in submit description file, 22
available platforms, 5

Backfill, 478
BOINC Configuration in HTCondor, 481
BOINC Installation, 480
BOINC Overview, 479
Defining HTCondor policy, 479
Overview, 478

backfill state, 356, 367
BACKFILL_SYSTEM macro, 252, 479
BASE_CGROUP macro, 299, 486
batch grid type, 568
batch system, 9
BATCH_GAHP macro, 302, 569
BATCH_GAHP_CHECK_STATUS_ATTEMPTS macro, 302
BatchQueue
    job ClassAd attribute, 986
BENCHMARKS_<JobName>_ARGS macro, 338
BENCHMARKS_<JobName>_CWD macro, 338
BENCHMARKS_<JobName>_ENV macro, 338
BENCHMARKS_<JobName>_EXECUTABLE macro, 336
BENCHMARKS_<JobName>_JOB_LOAD macro, 337
BENCHMARKS_<JobName>_KILL macro, 338
BENCHMARKS_<JobName>_MODE macro, 336
BENCHMARKS_<JobName>_PERIOD macro, 336
BENCHMARKS_<JobName>_PREFIX macro, 336
BENCHMARKS_<JobName>_SLOTS macro, 336
BENCHMARKS_CONFIG_VAL macro, 335
BENCHMARKS_JOBLIST macro, 336
BENCHMARKS_MAX_JOB_LOAD macro, 337
BIN macro, 207
BIND_ALL_INTERFACES macro, 228, 436
BlockReadKbytes
    job ClassAd attribute, 986
BlockReads
    job ClassAd attribute, 986
BlockWriteKbytes
    job ClassAd attribute, 986
BlockWrites
    job ClassAd attribute, 986
BOINC, 577
BOINC grid jobs, 577
BOINC_Arguments macro, 481, 484
BOINC_Environment macro, 481
BOINC_Error macro, 482
BOINC_Executable macro, 480, 481, 483
BOINC_GAHP macro, 303
BOINC_InitialDir macro, 480, 481, 484
BOINC_Output macro, 481
BOINC_Owner macro, 480, 481, 484
BOINC_Universe macro, 481
BoincAuthenticatorFile
    job ClassAd attribute, 986
Bosco commands
    bosco_cluster, 734
    bosco_findplatform, 736
    bosco_install, 737
    bosco_ssh_start, 738
    bosco_start, 739
    bosco_stop, 740
    bosco_uninstall, 741
bosco_cluster command, 734
bosco_findplatform command, 736
bosco_install command, 737
bosco_ssh_start command, 738
bosco_start command, 739
bosco_stop command, 740
bosco_uninstall command, 741

C_GAHP_CONTACT_SCHEDD_DELAY macro, 302
C_GAHP_DEBUG macro, 223
C_GAHP_LOG macro, 302, 556
C_GAHP_WORKER_THREAD_LOG macro, 302
CCB (HTCondor Connection Brokering), 439
CCB_ADDRESS macro, 228, 439, 441
CCB_HEARTBEAT_INTERVAL macro, 229
CCB_POLLING_INTERVAL macro, 229
CCB_POLLING_MAX_INTERVAL macro, 229
CCB_POLLING_TIMESLICE macro, 229
CCB_READ_BUFFER macro, 229
CCB_RECONNECT_FILE macro, 229
CCB_SWEEP_INTERVAL macro, 229
CCB_WRITE_BUFFER macro, 229
central manager, 157
    installation issues, 162
certificate
    X.509, 403
CERTIFICATE_MAPFILE macro, 321, 410
CERTIFICATE_MAPFILE_ASSUME_HASH_KEYS macro, 321, 411, 705
cgroup based process tracking, 486
CGROUP_MEMORY_LIMIT_POLICY macro, 280, 488
cgroups
    resource limits, 488
checkpoint, 2, 3, 13, 522
    compression, 523
    implementation, 522
    library interface, 525

periodic, 3, 523
stand alone, 523
checkpoint image, 13
checkpoint server, 158
configuration of, 445
installation, 444–448
multiple servers, 446
CHECKPOINT_PLATFORM macro, 246
Chirp, 66
Chirp.jar, 67
ChirpClient, 66
ChirpInputStream, 66
ChirpOutputStream, 66
Chirp API, 613
CHIRP_DELAYED_UPDATE_MAX_ATTRS macro, 282
CHIRP_DELAYED_UPDATE_PREFIX macro, 282
CHOWN_JOB_SPOOL_FILES macro, 276, 718
CKPT_PROBE macro, 213
CKPT_SERVER_CHECK_PARENT_INTERVAL macro, 237
CKPT_SERVER_CLASSAD_FILE macro, 237
CKPT_SERVER_CLEAN_INTERVAL macro, 237
CKPT_SERVER_CLIENT_TIMEOUT macro, 273
CKPT_SERVER_CLIENT_TIMEOUT_RETRY macro, 273
CKPT_SERVER_DEBUG macro, 446
CKPT_SERVER_DIR macro, 237, 445
CKPT_SERVER_HOST macro, 237, 439, 446, 447
CKPT_SERVER_INTERVAL macro, 237
CKPT_SERVER_LOG macro, 446
CKPT_SERVER_MAX_PROCESSES macro, 238
CKPT_SERVER_MAX_RESTORE_PROCESSES macro, 238
CKPT_SERVER_MAX_STORE_PROCESSES macro, 238
CKPT_SERVER_REMOVE_STALE_CKPT_INTERVAL macro, 238
CKPT_SERVER_SOCKET_BUFSIZE macro, 238
CKPT_SERVER_STALE_CKPT_AGE_CUTOFF macro, 238
CkptArch
job ClassAd attribute, 986
CkptOpSys
job ClassAd attribute, 986
claim lease, 358
CLAIM_PARTITIONABLE_LEFTOVERS macro, 254
CLAIM_WORKLIFE macro, 248, 369
claimed state, 355, 364
ClassAd, 2, 3, 10, 503–522
absent ClassAd, 454
attributes, 10, 506
Collector attributes, 1037
DaemonCore statistics attributes, 1040
DaemonMaster attributes, 1020
Defrag attributes, 1036
expression examples, 517
expression functions, 506
expression operators, 506, 516
expression syntax of Old ClassAds, 505
job, 10
job attributes, 986
machine, 10
machine attributes, 1004
machine example, 11
Negotiator attributes, 1032
Scheduler attributes, 1021
scope of evaluation, MY., 515
scope of evaluation, TARGET., 515
submitter attributes, 1035
ClassAd attribute
CurrentTime, 1020
rank, 29, 519
rank examples, 29
requirements, 29, 519
ClassAd attribute added by the condor_collector, 1039
AuthenticatedIdentity, 1039
AuthenticationMethod, 1039
LastHeardFrom, 1039
UpdatesHistory, 289, 1039
UpdatesLost, 288, 1039
UpdatesSequenced, 288, 1039
UpdatesTotal, 288, 1039
ClassAd attribute, ephemeral
RemoteAutoregroup, 345
RemoteGroup, 344
RemoteGroupQuota, 344
RemoteGroupResourcesInUse, 344
RemoteNegotiatingGroup, 344
RemoteUserPrio, 344
RemoteUserResourcesInUse, 344
Slot<N>_RemoteUserPrio, 344
SubmitterAutoregroup, 345

SubmitterGroup, 344
SubmitterGroupQuota, 344
SubmitterGroupResourcesInUse, 344
SubmitterNegotiatingGroup, 344
SubmitterUserPrio, 344
SubmitterUserResourcesInUse, 344
ClassAd Collector attribute
CollectorIpAddr, 1037
CondorVersion, 1037
CurrentJobsRunning<universe>, 1037
CurrentJobsRunningAll, 1037
DaemonStartTime, 1037
HostsClaimed, 1037
HostsOwner, 1037
HostsTotal, 1037
HostsUnclaimed, 1037
IdleJobs, 1037
Machine, 1037
MaxJobsRunning<universe>, 1038
MaxJobsRunningAll, 1038
MyAddress, 1038
MyCurrentTime, 1038
Name, 1038
RunningJobs, 1038
StartdAds, 1038
StartdAdsPeak, 1038
SubmitterAds, 1038
SubmitterAdsPeak, 1038
UpdateInterval, 1038
UpdateSequenceNumber, 1038
UpdatesInitial, 1038
UpdatesInitial_<ClassAd-Name>, 1038
UpdatesLost, 1038
UpdatesLost_<ClassAd-Name>, 1038
UpdatesLostMax, 1039
UpdatesLostRatio, 1039
UpdatesTotal, 1039
UpdatesTotal_<ClassAd-Name>, 1039
ClassAd DaemonMaster attribute
CkptServer, 1020
CondorVersion, 1020
DaemonStartTime, 1020
Machine, 1020
MasterIpAddr, 1020
MonitorSelfAge, 1020
MonitorSelfCPUUsage, 1020
MonitorSelfImageSize, 1021
MonitorSelfRegisteredSocketCount, 1021
MonitorSelfResidentSetSize, 1021
MonitorSelfSecuritySessions, 1021
MonitorSelfTime, 1021
MyAddress, 1021
MyCurrentTime, 1021
Name, 1021
PublicNetworkIpAddr, 1021
RealUid, 1021
UpdateSequenceNumber, 1021
ClassAd Defrag attribute
AvgDrainingBadput, 1036
AvgDrainingUnclaimedTime, 1036
DaemonStartTime, 1036
DrainedMachines, 1036
DrainFailures, 1036
DrainSuccesses, 1036
Machine, 1036
MachinesDraining, 1036
MachinesDrainingPeak, 1036
MeanDrainedArrived, 1036
MonitorSelfAge, 1036
MonitorSelfCPUUsage, 1036
MonitorSelfImageSize, 1036
MonitorSelfRegisteredSocketCount, 1036
MonitorSelfResidentSetSize, 1036
MonitorSelfSecuritySessions, 1036
MonitorSelfTime, 1036
MyAddress, 1036
MyCurrentTime, 1036
Name, 1036
RecentDrainFailures, 1037
RecentDrainSuccesses, 1037
RecentStatsLifetime, 1037
UpdateSequenceNumber, 1037
WholeMachines, 1037
WholeMachinesPeak, 1037
ClassAd functions, 506
ceiling(), 509
debug(), 512
envV1ToV2(), 513
eval(), 375, 507
floor(), 508
formatTime(), 511
ifThenElse(), 507

int(), 508
interval(), 512
isBoolean(), 508
isError(), 508
isInteger(), 508
isReal(), 508
isString(), 508
isUndefined(), 508
join(), 510
mergeEnvironment(), 513
pow(), 509
quantize(), 509
random(), 510
real(), 508
regexp(), 514
regexps(), 514
round(), 510
size(), 511
split(), 511
splitSlotName(), 511
splitUserName(), 511
strcat(), 510
strcmp(), 510
stricmp(), 511
string(), 508
stringList_regexpMember(), 514
stringListAvg(), 513
stringListIMember(), 514
stringListMax(), 513
stringListMember(), 514
stringListMin(), 513
stringListsIntersect(), 514
stringListSize(), 513
stringListSum(), 513
substr(), 510
time(), 511
toLower(), 511
toUpper(), 511
unparse(), 507
userHome(), 515
userMap(), 515
ClassAd job attribute
<Name>Provisioned, 1004
Absent, 986
AccountingGroup, 347
AcctGroup, 347, 986
AcctGroupUser, 347, 986
AllRemoteHosts, 986
Args, 986
Arguments, 986
BatchQueue, 986
BlockReadKbytes, 986
BlockReads, 986
BlockWriteKbytes, 986
BlockWrites, 986
BoincAuthenticatorFile, 986
CkptArch, 986
CkptOpSys, 986
ClusterId, 929, 986
Cmd, 987
CommittedSlotTime, 987
CommittedSuspensionTime, 987
CommittedTime, 987
CompletionDate, 987
ConcurrencyLimits, 987
CpusProvisioned, 1004
CumulativeRemoteSysCpu, 1000
CumulativeRemoteUserCpu, 1000
CumulativeSlotTime, 987
CumulativeSuspensionTime, 987
CumulativeTransferTime, 987
CurrentHosts, 987
DAG_InRecovery, 1003
DAG_NodesDone, 1003
DAG_NodesFailed, 1003
DAG_NodesPostrun, 1003
DAG_NodesPrerun, 1003
DAG_NodesQueued, 1003
DAG_NodesReady, 1003
DAG_NodesTotal, 1003
DAG_NodesUnready, 1003
DAG_Status, 1003
DAGManJobId, 987
DAGManNodesLog, 987
DAGManNodesMask, 988
DAGParentNodeNames, 84, 987
DeferralPrepTime, 149
DeferralTime, 148
DeferralWindow, 148
DelegateJobGSICredentialsLifetime, 988
DiskProvisioned, 1004
DiskUsage, 988

EC2AccessKeyId, 989
EC2AmiID, 989
EC2BlockDeviceMapping, 989
EC2ElasticIp, 989
EC2IamProfileArn, 989
EC2IamProfileName, 989
EC2InstanceName, 989
EC2InstanceType, 989
EC2KeyPair, 989
EC2KeyPairFile, 989
EC2ParameterNames, 989
EC2RemoteVirtualMachineName, 990
EC2SecretAccessKey, 990
EC2SecurityGroups, 990
EC2SecurityIDs, 990
EC2SpotPrice, 989
EC2SpotRequestID, 989
EC2StatusReasonCode, 989
EC2TagNames, 989
EC2UserData, 990
EC2UserDataFile, 990
EmailAttributes, 990
EncryptExecuteDirectory, 990
EnteredCurrentStatus, 990
Env, 990
Environment, 990
ExecutableSize, 990
ExitBySignal, 990
ExitCode, 990
ExitSignal, 991
ExitStatus, 991
GceAuthFile, 991
GceImage, 991
GceJsonFile, 991
GceMachineType, 991
GceMetadata, 991
GceMetadataFile, 991
GcePreemptible, 991
GlobalJobId, 991
GridJobStatus, 991
GridResource, 991
HoldKillSig, 991
HoldReason, 991
HoldReasonCode, 991
HoldReasonSubCode, 993
ImageSize, 993
IwdFlushNFSCache, 155, 994
JobAdInformationAttrs, 994
JobCurrentStartDate, 994
JobCurrentStartExecutingDate, 994
JobCurrentStartTransferOutputDate, 994
JobDescription, 994
JobLeaseDuration, 156, 994
JobMaxVacateTime, 994
JobNotification, 994
JobPrio, 994
JobRunCount, 994
JobStartDate, 995
JobStatus, 995
JobUniverse, 995
KeepClaimIdle, 995
KillSig, 995
KillSigTimeout, 996
LastCheckpointPlatform, 996
LastCkptServer, 996
LastCkptTime, 996
LastMatchTime, 996
LastRejMatchReason, 996
LastRejMatchTime, 996
LastRemotePool, 996
LastSuspensionTime, 996
LastVacateTime, 996
LeaveJobInQueue, 996
LocalSysCpu, 996
LocalUserCpu, 996
MachineAttr<X><N>, 996
MaxHosts, 996
MaxJobRetirementTime, 996
MaxTransferInputMB, 997
MaxTransferOutputMB, 997
MemoryProvisioned, 1004
MemoryUsage, 997
MinHosts, 997
NextJobStartDelay, 997
NiceUser, 997
Nonessential, 997
NTDomain, 997
NumCkpts, 997
NumGlobusSubmits, 997
NumJobCompletions, 997
NumJobMatches, 997
NumJobReconnects, 998

NumJobStarts, 998
NumPids, 998
NumRestarts, 998
NumShadowExceptions, 998
NumShadowStarts, 998
NumSystemHolds, 998
OtherJobRemoveRequirements, 998
OutputDestination, 998
Owner, 998
ParallelShutdownPolicy, 998
PostJobPrio1, 999
PostJobPrio2, 999
PreJobPrio1, 999
PreJobPrio2, 999
PreserveRelativeExecutable, 999
ProcId, 999
ProportionalSetSizeKb, 999
QDate, 999
RecentBlockReadKbytes, 999
RecentBlockReads, 999
RecentBlockWriteKbytes, 999
RecentBlockWrites, 1000
ReleaseReason, 1000
RemoteIwd, 1000
RemotePool, 1000
RemoteSysCpu, 1000
RemoteUserCpu, 1000
RemoteWallClockTime, 1000
RemoveKillSig, 1000
RequestCpus, 1000
RequestDisk, 1000
RequestedChroot, 1000
RequestMemory, 1000
ResidentSetSize, 1000
StackSize, 1000
StageOutFinish, 1001
StageOutStart, 1001
StreamErr, 1001
StreamOut, 1001
SubmitterAutoregroup, 1001
SubmitterGlobalJobId, 1001
SubmitterGroup, 1001
SubmitterNegotiatingGroup, 1001
TotalSuspensions, 1001
TransferErr, 1001
TransferExecutable, 1001
TransferIn, 1001
TransferInputSizeMB, 1001
TransferOut, 1002
TransferQueued, 1002
TransferringInput, 1002
TransferringOutput, 1002
UserLog, 1002
VM_MACAddr, 1003
WantGracefulRemoval, 1002
WindowsBuildNumber, 1002
WindowsMajorVersion, 1002
WindowsMinorVersion, 1002
X509UserProxy, 1002
X509UserProxyEmail, 1002
X509UserProxyExpiration, 1002
X509UserProxyFirstFQAN, 1002
X509UserProxyFQAN, 1002
X509UserProxySubject, 1003
X509UserProxyVOName, 1003
ClassAd machine attribute
Activity, 1004
Arch, 1004
AvailSince, 260
AvailTime, 260
AvailTimeEstimate, 260
CanHibernate, 1005
CheckpointPlatform, 1005
ClockDay, 1005
ClockMin, 1005
CondorLoadAvg, 382, 1005
CondorVersion, 1005
ConsoleIdle, 1005
CpuCacheSize, 1005
CpuFamily, 1005
CpuModel, 1005
Cpus, 1005
CurrentRank, 1005
DetectedCpus, 1005
DetectedMemory, 1005
Disk, 1005
DotNetVersions, 1006
Draining, 1006
DrainingRequestId, 1006
DynamicSlot, 1006
EnteredCurrentActivity, 1006
ExpectedMachineGracefulDrainingBadput, 1006

ExpectedMachineGracefulDrainingCompletion, 1006
ExpectedMachineQuickDrainingBadput, 1006
ExpectedMachineQuickDrainingCompletion, 1006
FileSystemDomain, 1006
has_avx, 1007
Has_sse4_1, 1007
Has_sse4_2, 1007
has_ssse3, 1007
HasDocker, 497, 1007
HasEncryptExecuteDirectory, 1007
HasFileTransfer, 1007
HasFileTransferPluginMethods, 1007
HasSingularity, 1007
HasVM, 1007
HookKeyword, 993
IsWakeAble, 1007
IsWakeEnabled, 1007
JobPreemptions, 1007
JobRankPreemptions, 1007
JobStarts, 1007
JobUserPrioPreemptions, 1007
JobVM_VCPUS, 1008
KeyboardIdle, 1008
KFlops, 1008
LastAvailInterval, 260
LastDrainStartTime, 1008
LastHeardFrom, 1008
LoadAvg, 382, 1008
Machine, 1008
MachineMaxVacateTime, 1008
MaxJobRetirementTime, 1008
Memory, 1008
Mips, 1008
MonitorSelfAge, 1008
MonitorSelfCPUUsage, 1008
MonitorSelfImageSize, 1008
MonitorSelfRegisteredSocketCount, 1008
MonitorSelfResidentSetSize, 1008
MonitorSelfSecuritySessions, 1008
MonitorSelfTime, 1009
MyAddress, 1009
MyType, 1009
Name, 1009
Offline<name>, 1009
OfflineUniverses, 1009
OpSys, 1009
OpSysAndVer, 1009
OpSysLegacy, 1010
OpSysLongName, 1011
OpSysMajorVer, 1011
OpSysName, 1012
OpSysShortName, 1012
OpSysVer, 1013
PartitionableSlot, 1013
RecentJobPreemptions, 1013
RecentJobRankPreemptions, 1013
RecentJobStarts, 1013
RecentJobUserPrioPreemptions, 1013
Requirements, 1013
RetirementTimeRemaining, 1013
SingularityVersion, 1014
SlotID, 1014
SlotType, 1014
SlotWeight, 1014
StartdIpAddr, 1014
State, 1014
TargetType, 1014
TotalCondorLoadAvg, 382, 1014
TotalCpus, 1014
TotalDisk, 1014
TotalLoadAvg, 382, 1015
TotalMachineDrainingBadput, 1015
TotalMachineDrainingUnclaimedTime, 1015
TotalMemory, 1015
TotalSlotCpus, 1015
TotalSlotDisk, 1015
TotalSlotMemory, 1015
TotalSlots, 1015
TotalTimeBackfillBusy, 1015
TotalTimeBackfillIdle, 1015
TotalTimeBackfillKilling, 1015
TotalTimeClaimedBusy, 1015
TotalTimeClaimedIdle, 1015
TotalTimeClaimedRetiring, 1015
TotalTimeClaimedSuspended, 1016
TotalTimeMatchedIdle, 1016
TotalTimeOwnerIdle, 1016
TotalTimePreemptingKilling, 1016
TotalTimePreemptingVacating, 1016
TotalTimeUnclaimedBenchmarking, 1016
TotalTimeUnclaimedIdle, 1016

UidDomain, 1016
VirtualMachineID, 1016
VirtualMemory, 1016
VM_AvailNum, 1016
VM_Guest_Mem, 1016
VM_Memory, 1016
VM_Networking, 1016
VM_Type, 1017
VMOfflineReason, 1017
VMOfflineTime, 1017
WindowsBuildNumber, 1017
WindowsMajorVersion, 1017
WindowsMinorVersion, 1017
ClassAd machine attribute (for a user-defined resource)
Assigned<name>, 1018
Offline<name>, 1018
Total<name>, 1018
ClassAd machine attribute (for GPU resources)
<name>BoardTempC, 1019
<name>Capability, 1019
<name>ClockMhz, 1019
<name>ComputeUnits, 1019
<name>CoresPerCU, 1019
<name>DeviceName, 1019
<name>DieTempC, 1019
<name>DriverVersion, 1019
<name>ECCEnabled, 1019
<name>EccErrorsDoubleBit, 1019
<name>EccErrorsSingleBit, 1019
<name>FanSpeedPct, 1019
<name>GlobalMemoryMb, 1019
<name>OpenCLVersion, 1019
<name>RuntimeVersion, 1019
ClassAd machine attribute (for pslot preemption)
ChildAccountingGroup, 1019
ChildActivity, 1019
ChildCpus, 1019
ChildCurrentRank, 1019
ChildEnteredCurrentState, 1020
ChildMemory, 1020
ChildName, 1020
ChildRemoteOwner, 1020
ChildRemoteUser, 1020
ChildRetirementTimeRemaining, 1020
ChildState, 1020
PslotRollupInformation, 1020
ClassAd machine attribute (in Claimed State)
ClientMachine, 1017
PreemptingOwner, 1017
PreemptingRank, 1017
PreemptingUser, 1017
RemoteAutoregroup, 1017
RemoteGroup, 1017
RemoteNegotiatingGroup, 1017
RemoteOwner, 1017
RemoteUser, 1017
TotalClaimRunTime, 1017
TotalClaimSuspendTime, 1018
TotalJobRunTime, 1018
TotalJobSuspendTime, 1018
ClassAd machine attribute (when offline)
MachineLastMatchTime, 1018
Offline, 1018
Unhibernate, 1018
ClassAd machine attribute (when running)
JobId, 1018
JobStart, 1018
LastPeriodicCheckpoint, 1018
ClassAd Negotiator attribute
CondorVersion, 1032
DaemonStartTime, 1032
LastNegotiationCycleActiveSubmitterCount<X>, 1032
LastNegotiationCycleCandidateSlots<X>, 1032
LastNegotiationCycleDuration<X>, 1032
LastNegotiationCycleEnd<X>, 1033
LastNegotiationCycleMatches<X>, 1033
LastNegotiationCycleMatchRate<X>, 1033
LastNegotiationCycleMatchRateSustained<X>, 1033
LastNegotiationCycleNumIdleJobs<X>, 1033
LastNegotiationCycleNumJobsConsidered<X>, 1033
LastNegotiationCycleNumSchedulers<X>, 1033
LastNegotiationCyclePeriod<X>, 1033
LastNegotiationCyclePhase1Duration<X>, 1033
LastNegotiationCyclePhase2Duration<X>, 1033
LastNegotiationCyclePhase3Duration<X>, 1033
LastNegotiationCyclePhase4Duration<X>, 1033
LastNegotiationCycleRejections<X>, 1033
LastNegotiationCycleSlotShareIter<X>, 1034
LastNegotiationCycleSubmittersFailed<X>, 1034
LastNegotiationCycleSubmittersOutOfTime<X>, 1034

LastNegotiationCycleSubmittersShareLimit, 1034
LastNegotiationCycleTime<X>, 1034
LastNegotiationCycleTotalSlots<X>, 1034
LastNegotiationCycleTrimmedSlots<X>, 1034
Machine, 1034
MyAddress, 1034
MyCurrentTime, 1034
Name, 1034
NegotiatorIpAddr, 1034
PublicNetworkIpAddr, 1034
UpdateSequenceNumber, 1035
ClassAd Scheduler attribute
Autoclusters, 1021
CollectorHost, 1021
CondorVersion, 1021
DaemonCoreDutyCycle, 1021
DaemonStartTime, 1021
DetectedCpus, 1021
DetectedMemory, 1021
FileTransferDiskThrottleExcess, 1029
FileTransferDiskThrottleHigh, 1029
FileTransferDiskThrottleLevel, 1030
FileTransferDiskThrottleLow, 1030
FileTransferDiskThrottleShortfall, 1030
FileTransferDownloadBytes, 1030
FileTransferDownloadBytesPerSecond, 1030
FileTransferFileNetReadLoad, 1031
FileTransferFileReadLoad, 1030
FileTransferFileReadSeconds, 1030
FileTransferFileWriteLoad, 1030
FileTransferFileWriteSeconds, 1031
FileTransferNetReadSeconds, 1031
FileTransferNetWriteLoad, 1031
FileTransferNetWriteSeconds, 1031
FileTransferUploadBytes, 1032
FileTransferUploadBytesPerSecond, 1032
JobQueueBirthdate, 1022
JobsAccumBadputTime, 1022
JobsAccumExceptionalBadputTime, 1022
JobsAccumRunningTime, 1022
JobsAccumTimeToStart, 1022
JobsBadputRuntimes, 1022
JobsBadputSizes, 1022
JobsCheckpointed, 1022
JobsCompleted, 1022
JobsCompletedRuntimes, 1022
JobsCompletedSizes, 1022
JobsCoredumped, 1022
JobsDebugLogError, 1022
JobsExecFailed, 1022
JobsExited, 1022
JobsExitedAndClaimClosing, 1023
JobsExitedNormally, 1023
JobsExitException, 1023
JobsKilled, 1023
JobsMissedDeferralTime, 1023
JobsNotStarted, 1023
JobsRestartReconnectsAttempting, 1023
JobsRestartReconnectsBadput, 1023
JobsRestartReconnectsFailed, 1023
JobsRestartReconnectsInterrupted, 1023
JobsRestartReconnectsLeaseExpired, 1023
JobsRestartReconnectsSucceeded, 1023
JobsRunning, 1023
JobsRunningRuntimes, 1024
JobsRunningSizes, 1024
JobsRuntimesHistogramBuckets, 1024
JobsShadowNoMemory, 1024
JobsShouldHold, 1024
JobsShouldRemove, 1024
JobsShouldRequeue, 1024
JobsSizesHistogramBuckets, 1024
JobsStarted, 1024
JobsSubmitted, 1024
Machine, 1024
MaxJobsRunning, 1024
MonitorSelfAge, 1024
MonitorSelfCPUUsage, 1024
MonitorSelfImageSize, 1024
MonitorSelfRegisteredSocketCount, 1024
MonitorSelfResidentSetSize, 1025
MonitorSelfSecuritySessions, 1025
MonitorSelfTime, 1025
MyAddress, 1025
MyCurrentTime, 1025
Name, 1025
NumJobStartsDelayed, 1025
NumPendingClaims, 1025
NumUsers, 1025
PublicNetworkIpAddr, 1025
RecentDaemonCoreDutyCycle, 1025
RecentJobsAccumBadputTime, 1025

RecentJobsAccumRunningTime, 1025
RecentJobsAccumTimeToStart, 1025
RecentJobsBadputRuntimes, 1025
RecentJobsBadputSizes, 1025
RecentJobsCheckpointed, 1025
RecentJobsCompleted, 1026
RecentJobsCompletedRuntimes, 1026
RecentJobsCompletedSizes, 1026
RecentJobsCoredumped, 1026
RecentJobsDebugLogError, 1026
RecentJobsExecFailed, 1026
RecentJobsExited, 1026
RecentJobsExitedAndClaimClosing, 1026
RecentJobsExitedNormally, 1026
RecentJobsExitException, 1026
RecentJobsKilled, 1026
RecentJobsMissedDeferralTime, 1026
RecentJobsNotStarted, 1026
RecentJobsShadowNoMemory, 1027
RecentJobsShouldHold, 1027
RecentJobsShouldRemove, 1027
RecentJobsShouldRequeue, 1027
RecentJobsStarted, 1027
RecentJobsSubmitted, 1027
RecentShadowsReconnections, 1027
RecentShadowsRecycled, 1027
RecentShadowsStarted, 1027
RecentStatsLifetime, 1027
RecentStatsTickTime, 1027
RecentWindowMax, 1027
ScheddIpAddr, 1028
ServerTime, 1028
ShadowsReconnections, 1028
ShadowsRecycled, 1028
ShadowsRunning, 1028
ShadowsRunningPeak, 1028
ShadowsStarted, 1028
StartLocalUniverse, 1028
StartSchedulerUniverse, 1028
StatsLastUpdateTime, 1028
StatsLifetime, 1028
TotalFlockedJobs, 1028
TotalHeldJobs, 1028
TotalIdleJobs, 1028
TotalJobAds, 1028
TotalLocalJobsIdle, 1029
TotalLocalJobsRunning, 1029
TotalRemovedJobs, 1029
TotalRunningJobs, 1029
TotalSchedulerJobsIdle, 1029
TotalSchedulerJobsRunning, 1029
TransferQueueMBWaitingToDownload, 1032
TransferQueueMBWaitingToUpload, 1032
TransferQueueNumWaitingToDownload, 1032
TransferQueueNumWaitingToUpload, 1032
TransferQueueUserExpr, 1029
UpdateInterval, 1029
UpdateSequenceNumber, 1029
VirtualMemory, 1029
WantResAd, 1029
ClassAd statistics attribute
DebugOuts, 1040
PipeMessages, 1040
PipeRuntime, 1040
SelectWaittime, 1040
SignalRuntime, 1040
Signals, 1040
SocketRuntime, 1040
SockMessages, 1040
TimerRuntime, 1040
TimersFired, 1040
ClassAd submitter attribute
CondorVersion, 1035
FlockedJobs, 1035
HeldJobs, 1035
IdleJobs, 1035
LocalJobsIdle, 1035
LocalJobsRunning, 1035
MyAddress, 1035
Name, 1035
RunningJobs, 1035
ScheddIpAddr, 1035
ScheddName, 1035
SchedulerJobsIdle, 1035
SchedulerJobsRunning, 1035
SubmitterTag, 1035
WeightedIdleJobs, 1035
WeightedRunningJobs, 1035
CLASSAD_LIFETIME macro, 286, 454
CLASSAD_LOG_STRICT_PARSING macro, 212
CLASSAD_USER_LIBS macro, 215, 522, 634
CLASSAD_USER_MAPDATA_<name> macro, 218

CLASSAD_USER_MAPFILE_<name> macro, 217, 218
CLASSAD_USER_PYTHON_LIB macro, 215
CLASSAD_USER_PYTHON_MODULES macro, 215
CLIENT_TIMEOUT macro, 287
clipped platform
availability, 5
definition of, 5
cluster
definition, 986
ClusterId
job ClassAd attribute, 929, 986
CM_IP_ADDR macro, 213
Cmd
job ClassAd attribute, 987
COD
attributes, 528
ClusterId, 530
ProcID, 530
authorizing users, 527
condor_cod activate command, 530
condor_cod tool, 531
condor_cod_activate command, 534
condor_cod_deactivate command, 536
condor_cod_delegate_proxy command, 537
condor_cod_release command, 537
condor_cod_renew command, 535
condor_cod_request command, 532
condor_cod_resume command, 536
condor_cod_suspend command, 535
defining an application, 527
defining applications
Job ID, 530
defining attributes by configuration, 530
introduction, 526
limitations, 537
managing claims, 531
optional attributes, 528
Args, 529
Env, 529
Err, 529
In, 529
IWD, 529
JarFiles, 529
JobUniverse, 529
KillSig, 529
Out, 529
StarterUserLog, 529
StarterUserLogUseXML, 530
overview, 527
required attributes, 528
Cmd, 528
Owner, 528
RequestCpus, 528
RequestDisk, 528
RequestMemory, 528
COD (Computing on Demand), 526–538
COLLECTOR_ADDRESS_FILE macro, 433
COLLECTOR_ADDRESS_FILE macro, 225
COLLECTOR_ALLOW_ONLY_ONE_NEGOTIATOR macro, 705
COLLECTOR_CLASS_HISTORY_SIZE macro, 289
COLLECTOR_DAEMON_HISTORY_SIZE macro, 289, 744, 955, 1039
COLLECTOR_DAEMON_STATS macro, 288, 289
COLLECTOR_DEBUG macro, 289
COLLECTOR_FORWARD_FILTERING macro, 290, 706
COLLECTOR_FORWARD_INTERVAL macro, 290
COLLECTOR_FORWARD_WATCH_LIST macro, 290
COLLECTOR_HOST macro, 207, 433, 1021
COLLECTOR_MAX_FILE_DESCRIPTORS macro, 232
COLLECTOR_NAME macro, 287
COLLECTOR_PERSISTENT_AD_LOG macro, 259, 290, 331, 455
COLLECTOR_PORT macro, 207
COLLECTOR_QUERY_WORKERS macro, 289
COLLECTOR_REQUIREMENTS macro, 287
COLLECTOR_SOCKET_BUFSIZE macro, 288
COLLECTOR_STATS_SWEEP macro, 289
COLLECTOR_SUPER_ADDRESS_FILE macro, 226
COLLECTOR_TCP_SOCKET_BUFSIZE macro, 288
COLLECTOR_UPDATE_INTERVAL macro, 288
COLLECTOR_USES_SHARED_PORT macro, 229
CommittedSlotTime
job ClassAd attribute, 987
CommittedSuspensionTime
job ClassAd attribute, 987
CommittedTime
job ClassAd attribute, 987
compilers
supported with condor_compile, 5
CompletionDate
job ClassAd attribute, 987

COMPRESS_PERIODIC_CKPT macro, 277
COMPRESS_VACATE_CKPT macro, 277
Computing On Demand
    Defining Applications
        Job ID, 530
        Optional attributes, 528
        Required attributes, 528
Computing on Demand (see COD), 526
concurrency limits, 489
CONCURRENCY_LIMIT_DEFAULT macro, 295, 490
CONCURRENCY_LIMIT_DEFAULT_<NAME> macro, 295
ConcurrencyLimits
    job ClassAd attribute, 987
Condor commands
    condor_convert_history, 771
CONDOR_ADMIN macro, 163, 211, 818
condor_advertise command, 742
condor_check_userlogs command, 746
condor_checkpoint command, 747
condor_chirp, 750
condor_ckpt_server daemon, 160, 444
condor_cod command, 754
condor_collector, 441
condor_collector daemon, 159
condor_compile command, 757
    list of supported compilers, 5
condor_config_val command, 759
condor_configure command, 168, 764, 801
condor_continue command, 769
condor_convert_history command, 771
condor_credd daemon, 160, 299, 643
condor_dagman command, 773
condor_dagman_metrics_reporter command, 779
condor_dbmsd source code contrib daemon, 656
condor_defrag daemon, 160, 390
CONDOR_DEVELOPERS macro, 7, 136, 287, 780
CONDOR_DEVELOPERS_COLLECTOR macro, 7, 287
condor_drain command, 782
condor_fetchlog command, 784
condor_findhost command, 787
CONDOR_FSYNC macro, 215
CONDOR_GAHP macro, 302, 556
condor_gangliad daemon, 340, 452
condor_gather_info command, 789
condor_gpu_discovery command, 792
condor_gridmanager daemon, 160
condor_had daemon, 160, 458
condor_hdfs daemon, 160
condor_history command, 795
condor_hold command, 798
CONDOR_HOST macro, 207
CONDOR_ID environment variable, 43
CONDOR_IDS environment variable, 43, 164, 210, 212
CONDOR_IDS macro, 164, 210, 426, 539, 540
condor_install command, 764, 801
condor_job_router daemon, 160, 583
condor_job_router_info command, 806
condor_kbdd daemon, 159, 470
condor_lease_manager daemon, 160
condor_master daemon, 158, 808
condor_negotiator daemon, 159
condor_off command, 809
condor_on command, 812
condor_ping command, 815
condor_pool_job_report command, 818
condor_power command, 819
condor_preen command, 821
condor_prio command, 823
condor_procd command, 825
condor_procd daemon, 160
condor_q command, 828
CONDOR_Q_DASH_BATCH_IS_DEFAULT macro, 265
CONDOR_Q_ONLY_MY_JOBS macro, 265, 704, 715
CONDOR_Q_USE_V3_PROTOCOL macro, 265
condor_qedit command, 843
condor_qsub command, 845
condor_quill source code contrib daemon, 656
condor_reconfig command, 850
condor_release command, 853
condor_replication daemon, 160, 458
condor_reschedule command, 855
condor_restart command, 857
condor_rm command, 860
condor_rmdir command, 863
condor_rooster daemon, 160, 501
condor_router_history, 865
condor_router_q, 867
condor_router_rm command, 869
condor_run command, 871
condor_schedd daemon, 159
condor_set_shutdown command, 874

condor_shadow, 14, 50
condor_shadow daemon, 159
condor_shared_port daemon, 160, 435
condor_sos command, 880
CONDOR_SSH_KEYGEN macro, 73
condor_ssh_to_job command, 876
CONDOR_SSHD macro, 73
condor_startd daemon, 158
condor_startd daemon, 352
condor_starter daemon, 159
condor_stats command, 882
condor_status command, 885
condor_store_cred command, 893
condor_submit command, 895
condor_submit variables, 931
condor_submit_dag command, 937
CONDOR_SUPPORT_EMAIL macro, 211
condor_suspend command, 944
condor_tail command, 946
condor_transfer_data command, 948
condor_transferer daemon, 160, 458
condor_transform_ads command, 950
condor_update_machine_ad command, 953
condor_updates_stats command, 955
condor_urlfetch command, 958
condor_userlog command, 960
condor_userprio command, 963
condor_vacate command, 968
condor_vacate_job command, 970
condor_version command, 973
CONDOR_VIEW_CLASSAD_TYPES macro, 289
CONDOR_VIEW_HOST macro, 207, 232, 473
CONDOR_VM environment variable, 42
condor_wait command, 975
condor_who command, 978
configuration
    checkpoint server configuration variables, 237
    condor_collector configuration variables, 286
    condor_credd configuration variables, 299
    condor_defrag configuration variables, 339
    condor_gangliad configuration variables, 340
    condor_gridmanager configuration variables, 299
    condor_hdfs configuration variables, 655
    condor_job_router configuration variables, 303
    condor_lease_manager configuration variables, 305
    condor_master configuration variables, 238
    condor_negotiator configuration variables, 290
    condor_preen configuration variables, 286
    condor_rooster configuration variables, 331
    condor_schedd configuration variables, 261
    condor_schedd policy, 391
    condor_shadow configuration variables, 276
    condor_shared_port configuration variables, 331
    condor_ssh_to_job configuration variables, 329
    condor_startd configuration variables, 244
    condor_startd policy, 351
    condor_starter configuration variables, 278
    condor_submit configuration variables, 283
    daemon logging configuration variables, 218
    DaemonCore configuration variables, 225
    DAGMan configuration variables, 308
    Error and warning syntax, 193
    example, 354
    for flocking, 554
    function macros, 196
    grid configuration variables, 307
    Grid Monitor configuration variables, 306
    high availability configuration variables, 324
    hook configuration variables, 333
    HTCondor-wide configuration variables, 206
    IF/ELSE syntax, 194
    INCLUDE syntax, 192
    multi-core machines, 381
    network-related configuration variables, 228
    of machines, to implement a given policy, 351
    pre-defined macros, 198
    security configuration variables, 316
    shared file system configuration variables, 233
    SMP machines, 381
    to use GPUs, 383
    USE syntax, 201
    virtual machine configuration variables, 322
    Windows platform configuration variables, 338
configuration change requiring a restart of HTCondor, 198
configuration file
    evaluation order, 185
    macro definitions, 186
    macros, 200
    pre-defined macros, 198
    subsystem names, 199
configuration files
    location, 165

configuration macro
-maxidle, 729
<DaemonName>_ENVIRONMENT, 239
<Keyword>_HOOK_EVICT_CLAIM, 334, 540
<Keyword>_HOOK_FETCH_WORK, 334, 539, 540, 542
<Keyword>_HOOK_JOB_CLEANUP, 335, 546
<Keyword>_HOOK_JOB_EXIT_TIMEOUT, 334
<Keyword>_HOOK_JOB_EXIT, 334, 541, 542
<Keyword>_HOOK_JOB_FINALIZE, 335, 546
<Keyword>_HOOK_PREPARE_JOB, 334, 540, 992
<Keyword>_HOOK_REPLY_CLAIM, 334
<Keyword>_HOOK_REPLY_FETCH, 334, 540
<Keyword>_HOOK_TRANSLATE_JOB, 335, 546, 588
<Keyword>_HOOK_UPDATE_JOB_INFO, 334, 541, 542, 546
<NAME>_LIMIT, 295
<SUBSYS>_<LEVEL>_LOG, 223
<SUBSYS>_ADDRESS_FILE, 225, 432
<SUBSYS>_ADMIN_EMAIL, 211
<SUBSYS>_ARGS, 239
<SUBSYS>_ATTRS, 226
<SUBSYS>_CLASSAD_USER_MAP_NAMES, 217
<SUBSYS>_DAEMON_AD_FILE, 226
<SUBSYS>_DEBUG, 221, 550
<SUBSYS>_ENABLE_SOAP_SSL, 328
<SUBSYS>_EXPRS, 226
<SUBSYS>_LOCK, 219, 550
<SUBSYS>_LOG_KEEP_OPEN, 219, 550
<SUBSYS>_LOG, 218, 550
<SUBSYS>_MAX_FILE_DESCRIPTORS, 229
<SUBSYS>_NOT_RESPONDING_TIMEOUT, 227
<SUBSYS>_SOAP_SSL_PORT, 328
<SUBSYS>_SUPER_ADDRESS_FILE, 226, 880
<SUBSYS>_TIMEOUT_MULTIPLIER, 232
<SUBSYS>_USERID, 239
<SUBSYS>, 239
<var>_ATTRS, 730
<var>_EXPRS, 730
ABORT_ON_EXCEPTION, 213
ABSENT_EXPIRE_ADS_AFTER, 290, 455
ABSENT_REQUIREMENTS, 290, 455
ABSENT_SUBMITTER_LIFETIME, 266
ABSENT_SUBMITTER_UPDATE_RATE, 266
ACCOUNTANT_LOCAL_DOMAIN, 291
ADD_SIGNIFICANT_ATTRIBUTES, 274
ADD_WINDOWS_FIREWALL_EXCEPTION, 243
ADVERTISE_IPV4_FIRST, 217
ADVERTISE_PSLOT_ROLLUP_INFORMATION, 245
ALIVE_INTERVAL, 248, 267, 358
ALLOW_* macros, 417
ALLOW_ADMIN_COMMANDS, 243
ALLOW_CLIENT, 317, 397
ALLOW_CONFIG, 644
ALLOW_PSLOT_PREEMPTION, 295, 725
ALLOW_SCRIPTS_TO_RUN_AS_EXECUTABLES, 214
ALLOW_VM_CRUFT, 42, 256, 1016
ALLOW_WRITE, 163
ALL_DEBUG, 223
ALWAYS_REUSEADDR, 233
ALWAYS_USE_LOCAL_CKPT_SERVER, 238
ALWAYS_VM_UNIV_USE_NOBODY, 323
APPEND_PREF_STANDARD, 284
APPEND_PREF_VANILLA, 284
APPEND_RANK_STANDARD, 284
APPEND_RANK_VANILLA, 284
APPEND_RANK, 284
APPEND_REQUIREMENTS, 284
APPEND_REQ_STANDARD, 284
APPEND_REQ_VANILLA, 284
ARCH, 200
ASSIGN_CPU_AFFINITY, 280
AUTH_SSL_CLIENT_CADIR, 320, 406
AUTH_SSL_CLIENT_CAFILE, 320, 406
AUTH_SSL_CLIENT_CERTFILE, 321, 406
AUTH_SSL_CLIENT_KEYFILE, 321, 406
AUTH_SSL_SERVER_CADIR, 320, 406
AUTH_SSL_SERVER_CAFILE, 320, 406
AUTH_SSL_SERVER_CERTFILE, 320, 406
AUTH_SSL_SERVER_KEYFILE, 321, 406
AUTO_INCLUDE_SHARED_PORT_IN_DAEMON_LIST, 229
AfterHours, 373
BACKFILL_SYSTEM, 252, 479
BASE_CGROUP, 299, 486
BATCH_GAHP_CHECK_STATUS_ATTEMPTS, 302
BATCH_GAHP, 302, 569

BENCHMARKS_<JobName>_ARGS, 338
BENCHMARKS_<JobName>_CWD, 338
BENCHMARKS_<JobName>_ENV, 338
BENCHMARKS_<JobName>_EXECUTABLE, 336
BENCHMARKS_<JobName>_JOB_LOAD, 337
BENCHMARKS_<JobName>_KILL, 338
BENCHMARKS_<JobName>_MODE, 336
BENCHMARKS_<JobName>_PERIOD, 336
BENCHMARKS_<JobName>_PREFIX, 336
BENCHMARKS_<JobName>_SLOTS, 336
BENCHMARKS_CONFIG_VAL, 335
BENCHMARKS_JOBLIST, 336
BENCHMARKS_MAX_JOB_LOAD, 337
BIND_ALL_INTERFACES, 228, 436
BIN, 207
BOINC_Arguments, 481, 484
BOINC_Environment, 481
BOINC_Error, 482
BOINC_Executable, 480, 481, 483
BOINC_GAHP, 303
BOINC_InitialDir, 480, 481, 484
BOINC_Output, 481
BOINC_Owner, 480, 481, 484
BOINC_Universe, 481
CCB_ADDRESS, 228, 439, 441
CCB_HEARTBEAT_INTERVAL, 229
CCB_POLLING_INTERVAL, 229
CCB_POLLING_MAX_INTERVAL, 229
CCB_POLLING_TIMESLICE, 229
CCB_READ_BUFFER, 229
CCB_RECONNECT_FILE, 229
CCB_SWEEP_INTERVAL, 229
CCB_WRITE_BUFFER, 229
CERTIFICATE_MAPFILE_ASSUME_HASH_KEYS, 321, 411, 705
CERTIFICATE_MAPFILE, 321, 410
CGROUP_MEMORY_LIMIT_POLICY, 280, 488
CHECKPOINT_PLATFORM, 246
CHIRP_DELAYED_UPDATE_MAX_ATTRS, 282
CHIRP_DELAYED_UPDATE_PREFIX, 282
CHOWN_JOB_SPOOL_FILES, 276, 718
CKPT_PROBE, 213
CKPT_SERVER_CHECK_PARENT_INTERVAL, 237
CKPT_SERVER_CLASSAD_FILE, 237
CKPT_SERVER_CLEAN_INTERVAL, 237
CKPT_SERVER_CLIENT_TIMEOUT_RETRY, 273
CKPT_SERVER_CLIENT_TIMEOUT, 273
CKPT_SERVER_DEBUG, 446
CKPT_SERVER_DIR, 237, 445
CKPT_SERVER_HOST, 237, 439, 446, 447
CKPT_SERVER_INTERVAL, 237
CKPT_SERVER_LOG, 446
CKPT_SERVER_MAX_PROCESSES, 238
CKPT_SERVER_MAX_RESTORE_PROCESSES, 238
CKPT_SERVER_MAX_STORE_PROCESSES, 238
CKPT_SERVER_REMOVE_STALE_CKPT_INTERVAL, 238
CKPT_SERVER_SOCKET_BUFSIZE, 238
CKPT_SERVER_STALE_CKPT_AGE_CUTOFF, 238
CLAIM_PARTITIONABLE_LEFTOVERS, 254
CLAIM_WORKLIFE, 248, 369
CLASSAD_LIFETIME, 286, 454
CLASSAD_LOG_STRICT_PARSING, 212
CLASSAD_USER_LIBS, 215, 522, 634
CLASSAD_USER_MAPDATA_<name>, 218
CLASSAD_USER_MAPFILE_<name>, 217, 218
CLASSAD_USER_PYTHON_LIB, 215
CLASSAD_USER_PYTHON_MODULES, 215
CLIENT_TIMEOUT, 287
CM_IP_ADDR, 213
COLLECTOR_ADDRESS_FILE, 433
COLLECTOR_ALLOW_ONLY_ONE_NEGOTIATOR, 705
COLLECTOR_CLASS_HISTORY_SIZE, 289
COLLECTOR_DAEMON_HISTORY_SIZE, 289, 744, 955, 1039
COLLECTOR_DAEMON_STATS, 288, 289
COLLECTOR_DEBUG, 289
COLLECTOR_FORWARD_FILTERING, 290, 706
COLLECTOR_FORWARD_INTERVAL, 290
COLLECTOR_FORWARD_WATCH_LIST, 290
COLLECTOR_HOST, 207, 433, 1021
COLLECTOR_MAX_FILE_DESCRIPTORS, 232
COLLECTOR_NAME, 287
COLLECTOR_PERSISTENT_AD_LOG, 259, 290, 331, 455
COLLECTOR_PORT, 207
COLLECTOR_QUERY_WORKERS, 289

COLLECTOR_REQUIREMENTS, 287
COLLECTOR_SOCKET_BUFSIZE, 288
COLLECTOR_STATS_SWEEP, 289
COLLECTOR_TCP_SOCKET_BUFSIZE, 288
COLLECTOR_UPDATE_INTERVAL, 288
COLLECTOR_USES_SHARED_PORT, 229
COMPRESS_PERIODIC_CKPT, 277
COMPRESS_VACATE_CKPT, 277
CONCURRENCY_LIMIT_DEFAULT_<NAME>, 295
CONCURRENCY_LIMIT_DEFAULT, 295, 490
CONDOR_ADMIN, 163, 211, 818
CONDOR_DEVELOPERS_COLLECTOR, 7, 287
CONDOR_DEVELOPERS, 7, 136, 287, 780
CONDOR_FSYNC, 215
CONDOR_GAHP, 302, 556
CONDOR_HOST, 207
CONDOR_IDS, 164, 210, 426, 539, 540
CONDOR_Q_DASH_BATCH_IS_DEFAULT, 265
CONDOR_Q_ONLY_MY_JOBS, 265, 704, 715
CONDOR_Q_USE_V3_PROTOCOL, 265
CONDOR_SSHD, 73
CONDOR_SSH_KEYGEN, 73
CONDOR_SUPPORT_EMAIL, 211
CONDOR_VIEW_CLASSAD_TYPES, 289
CONDOR_VIEW_HOST, 207, 232, 473
CONSOLE_DEVICES, 170, 248, 468
CONSUMPTION_<Resource>, 257
CONSUMPTION_POLICY, 257
CONTINUE, 245, 369
CORE_FILE_NAME, 228
COUNT_HYPERTHREAD_CPUS, 200, 250
CREAM_GAHP, 302
CREATE_CORE_FILES, 213
CREATE_LOCKS_ON_LOCAL_DISK, 209, 221
CREDD_CACHE_LOCALLY, 299
CREDD_HOST, 299
CREDD_POLLING_TIMEOUT, 299
CRED_MIN_TIME_LEFT, 285
CURB_MATCHMAKING, 263
C_GAHP_CONTACT_SCHEDD_DELAY, 302
C_GAHP_DEBUG, 223
C_GAHP_LOG, 302, 556
C_GAHP_WORKER_THREAD_LOG, 302
DAEMON_LIST, 229, 238, 435, 445, 468, 808
DAEMON_SHUTDOWN_FAST, 227
DAEMON_SHUTDOWN, 227, 1044
DAEMON_SOCKET_DIR, 331, 332, 551
DAGMAN_ABORT_DUPLICATES, 308
DAGMAN_ABORT_ON_SCARY_SUBMIT, 311
DAGMAN_ALLOW_EVENTS, 314
DAGMAN_ALLOW_LOG_ERROR, 313
DAGMAN_ALWAYS_RUN_POST, 309
DAGMAN_ALWAYS_USE_NODE_LOG, 314, 552
DAGMAN_AUTO_RESCUE, 312
DAGMAN_CONDOR_RM_EXE, 311
DAGMAN_CONDOR_SUBMIT_EXE, 311
DAGMAN_CONFIG_FILE, 308
DAGMAN_COPY_TO_SPOOL, 316
DAGMAN_DEBUG_CACHE_ENABLE, 315
DAGMAN_DEBUG_CACHE_SIZE, 315
DAGMAN_DEBUG, 315
DAGMAN_DEFAULT_NODE_LOG, 313, 315, 552
DAGMAN_DEFAULT_PRIORITY, 309
DAGMAN_GENERATE_SUBDAG_SUBMITS, 311
DAGMAN_HOLD_CLAIM_TIME, 100, 310
DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION, 314
DAGMAN_INSERT_SUB_FILE, 316
DAGMAN_LOG_ON_NFS_IS_ERROR, 313
DAGMAN_MAX_JOBS_IDLE, 86, 309, 310, 729, 774, 938
DAGMAN_MAX_JOBS_SUBMITTED, 85, 309, 774, 938
DAGMAN_MAX_JOB_HOLDS, 310
DAGMAN_MAX_POST_SCRIPTS, 86, 309, 775, 938
DAGMAN_MAX_PRE_SCRIPTS, 86, 309, 774, 938
DAGMAN_MAX_RESCUE_NUM, 123, 312
DAGMAN_MAX_SUBMITS_PER_INTERVAL, 310
DAGMAN_MAX_SUBMIT_ATTEMPTS, 310
DAGMAN_MUNGE_NODE_NAMES, 100, 311
DAGMAN_OLD_RESCUE, 312
DAGMAN_ON_EXIT_REMOVE, 316
DAGMAN_PEGASUS_REPORT_METRICS, 316
DAGMAN_PEGASUS_REPORT_TIMEOUT, 316, 780
DAGMAN_PENDING_REPORT_INTERVAL, 315, 552
DAGMAN_PROHIBIT_MULTI_JOBS, 310
DAGMAN_REMOVE_NODE_JOBS, 311
DAGMAN_RESET_RETRIES_UPON_RESCUE, 124, 312

DAGMAN_RETRY_NODE_FIRST, 309, 312
DAGMAN_RETRY_SUBMIT_FIRST, 312
DAGMAN_STARTUP_CYCLE_DETECT, 120, 308
DAGMAN_STORK_RM_EXE, 312
DAGMAN_STORK_SUBMIT_EXE, 311
DAGMAN_SUBMIT_DELAY, 310
DAGMAN_SUBMIT_DEPTH_FIRST, 309
DAGMAN_SUPPRESS_JOB_LOGS, 311
DAGMAN_SUPPRESS_NOTIFICATION, 311, 776, 937, 942
DAGMAN_USER_LOG_SCAN_INTERVAL, 127, 128, 310
DAGMAN_USE_OLD_DAG_READER, 308
DAGMAN_USE_SHARED_PORT, 308
DAGMAN_USE_STRICT, 124, 308
DAGMAN_VERBOSITY, 315, 552
DAGMAN_WRITE_PARTIAL_RESCUE, 125, 312
DC_DAEMON_LIST, 238
DEAD_COLLECTOR_MAX_AVOIDANCE_TIME, 214
DEBUG_TIME_FORMAT, 221
DEDICATED_EXECUTE_ACCOUNT_REGEXP, 234, 429, 485
DEDICATED_SCHEDULER_DELAY_FACTOR, 273
DEDICATED_SCHEDULER_USE_FIFO, 272
DEDICATED_SCHEDULER_WAIT_FOR_SPOOLER, 272
DEFAULT_DOMAIN_NAME, 213, 438
DEFAULT_IO_BUFFER_BLOCK_SIZE, 284
DEFAULT_IO_BUFFER_SIZE, 284
DEFAULT_JOB_MAX_RETRIES, 284
DEFAULT_MASTER_SHUTDOWN_SCRIPT, 241
DEFAULT_PRIO_FACTOR, 291, 343
DEFAULT_RANK_STANDARD, 284
DEFAULT_RANK_VANILLA, 284
DEFAULT_RANK, 284
DEFAULT_UNIVERSE, 283, 904
DEFRAG_CANCEL_REQUIREMENTS, 339
DEFRAG_DRAINING_MACHINES_PER_HOUR, 339
DEFRAG_INTERVAL, 339, 340
DEFRAG_LOG, 340
DEFRAG_MAX_CONCURRENT_DRAINING, 340
DEFRAG_MAX_WHOLE_MACHINES, 339
DEFRAG_NAME, 339, 1037
DEFRAG_RANK, 339
DEFRAG_REQUIREMENTS, 339
DEFRAG_SCHEDULE, 340
DEFRAG_STATE_FILE, 340
DEFRAG_UPDATE_INTERVAL, 340
DEFRAG_WHOLE_MACHINE_EXPR, 339
DELEGATE_FULL_JOB_GSI_CREDENTIALS, 318
DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME, 277, 318, 915, 988
DELEGATE_JOB_GSI_CREDENTIALS_REFRESH, 277, 318
DELEGATE_JOB_GSI_CREDENTIALS, 318, 916, 988
DENY_CLIENT, 317
DETECTED_CORES, 201, 1005
DETECTED_CPUS, 200, 249
DETECTED_MEMORY, 200, 250, 1005
DETECTED_PHYSICAL_CPUS, 200
DISCARD_SESSION_KEYRING_ON_STARTUP, 244
DISCONNECTED_KEYBOARD_IDLE_BOOST, 253, 381
DOCKER_DROP_ALL_CAPABILITIES, 261, 498, 709
DOCKER_IMAGE_CACHE_SIZE, 260, 498
DOCKER_VOLUMES, 260
DOCKER_VOLUME_DIR_XXX_MOUNT_IF, 704
DOCKER_VOLUME_DIR_xxx_MOUNT_IF, 497
DOCKER, 260, 498
DOT_NET_VERSIONS, 258
DYNAMIC_RUN_ACCOUNT_LOCAL_GROUP, 282, 647
D_COMMAND, 419
D_SECURITY, 419
DedicatedScheduler, 250, 475
EC2_GAHP_RATE_LIMIT, 301
EC2_GAHP, 302
EC2_RESOURCE_TIMEOUT, 301, 573
ECRYPTFS_ADD_PASSPHRASE, 320
EMAIL_DOMAIN, 213
EMAIL_SIGNATURE, 211
ENABLE_ADDRESS_REWRITING, 231
ENABLE_BACKFILL, 252, 479
ENABLE_CHIRP_DELAYED, 281
ENABLE_CHIRP_IO, 281

ENABLE_CHIRP_UPDATES, 281
ENABLE_CHIRP, 281
ENABLE_CLASSAD_CACHING, 215
ENABLE_DEPRECATION_WARNINGS, 285
ENABLE_GRID_MONITOR, 307
ENABLE_HISTORY_ROTATION, 212, 552
ENABLE_IPV4, 217, 442
ENABLE_IPV6, 217, 442
ENABLE_KERNEL_TUNING, 244
ENABLE_PERSISTENT_CONFIG, 225, 759
ENABLE_RUNTIME_CONFIG, 225
ENABLE_SOAP_SSL, 328
ENABLE_SOAP, 328
ENABLE_SSH_TO_JOB, 329
ENABLE_URL_TRANSFERS, 281
ENABLE_USERLOG_FSYNC, 220
ENABLE_USERLOG_LOCKING, 220
ENABLE_VERSIONED_OPSYS, 246
ENABLE_WEB_SERVER, 328
ENCRYPT_EXECUTE_DIRECTORY_FILENAMES, 320
ENCRYPT_EXECUTE_DIRECTORY, 319, 394
ENFORCE_CPU_AFFINITY, 281
ENVIRONMENT_FOR_Assigned<name>, 255
ENVIRONMENT_VALUE_FOR_UnAssigned<name>, 255
EVENT_LOG_COUNT_EVENTS, 221
EVENT_LOG_FSYNC, 224, 551
EVENT_LOG_JOB_AD_INFORMATION_ATTRS, 58, 224, 551
EVENT_LOG_LOCKING, 224, 551
EVENT_LOG_MAX_ROTATIONS, 224, 551
EVENT_LOG_MAX_SIZE, 224, 551
EVENT_LOG_ROTATION_LOCK, 224, 551
EVENT_LOG_USE_XML, 224, 551
EVENT_LOG, 224, 551
EVICT_BACKFILL, 253, 370, 479
EXECUTE_LOGIN_IS_DEDICATED, 235
EXECUTE, 208, 209, 495, 1006
EXEC_TRANSFER_ATTEMPTS, 278
EXPIRE_INVALIDATED_ADS, 290, 455
FEATURE : TESTINGMODE_POLICY_VALUES, 708
FEATURE : UWCS_DESKTOP_POLICY_VALUES, 708
FILESYSTEM_DOMAIN, 201, 235, 438
FILETRANSFER_PLUGINS, 281, 1007
FILE_LOCK_VIA_MUTEX, 220, 550
FILE_TRANSFER_DISK_LOAD_THROTTLE_LONG_HORIZO 264
FILE_TRANSFER_DISK_LOAD_THROTTLE_SHORT_HORIZ 264
FILE_TRANSFER_DISK_LOAD_THROTTLE_WAIT_BETWEE 264
FILE_TRANSFER_DISK_LOAD_THROTTLE, 264, 1030
FLOCK_COLLECTOR_HOSTS, 269, 270, 554
FLOCK_FROM, 555
FLOCK_INCREMENT, 270
FLOCK_NEGOTIATOR_HOSTS, 269, 554
FLOCK_TO, 554
FS_REMOTE_DIR, 319, 410
FULL_HOSTNAME, 199
FetchWorkDelay, 334, 539, 543
GAHP_ARGS, 301
GAHP_DEBUG_HIDE_SENSITIVE_DATA, 301
GAHP, 301
GANGLIAD_DEFAULT_CLUSTER, 341, 453
GANGLIAD_DEFAULT_IP, 342, 454
GANGLIAD_DEFAULT_MACHINE, 342, 454
GANGLIAD_INTERVAL, 340
GANGLIAD_LOG, 342
GANGLIAD_METRICS_CONFIG_DIR, 342, 452
GANGLIAD_PER_EXECUTE_NODE_METRICS, 341, 452
GANGLIAD_REQUIREMENTS, 341, 452
GANGLIAD_VERBOSITY, 341
GANGLIA_CONFIG, 341
GANGLIA_GMETRIC, 341
GANGLIA_GSTAT_COMMAND, 341, 452
GANGLIA_LIB64_PATH, 341
GANGLIA_LIB_PATH, 341
GANGLIA_LIB, 341
GANGLIA_SEND_DATA_FOR_ALL_HOSTS, 341, 452
GANGLIA_VERBOSITY, 453
GCE_GAHP, 303
GLEXEC_HOLD_ON_INITIAL_FAILURE, 307
GLEXEC_JOB, 307
GLEXEC_RETRIES, 307
GLEXEC_RETRY_DELAY, 307
GLEXEC, 307

GLITE_LOCATION, 302, 569
GLOBUS_GATEKEEPER_TIMEOUT, 301
GRACEFULLY_REMOVE_JOBS, 271
GRAM_VERSION_DETECTION, 301, 565
GRIDMANAGER_CHECKPROXY_INTERVAL, 299
GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT, 301
GRIDMANAGER_CONTACT_SCHEDD_DELAY, 300
GRIDMANAGER_EMPTY_RESOURCE_DELAY, 300
GRIDMANAGER_GAHP_CALL_TIMEOUT, 301
GRIDMANAGER_GAHP_RESPONSE_TIMEOUT, 301
GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT, 301
GRIDMANAGER_JOB_PROBE_INTERVAL, 300
GRIDMANAGER_JOB_PROBE_RATE, 300
GRIDMANAGER_LOG, 299
GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE, 301, 564
GRIDMANAGER_MAX_PENDING_REQUESTS, 301
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE, 300
GRIDMANAGER_MINIMUM_PROXY_TIME, 299
GRIDMANAGER_PROXY_REFRESH_TIME, 299
GRIDMANAGER_RESOURCE_PROBE_DELAY, 300
GRIDMANAGER_RESOURCE_PROBE_INTERVAL, 300, 573
GRIDMANAGER_SELECTION_EXPR, 273
GRIDMAP, 319, 404, 411
GRID_MONITOR_DISABLE_TIME, 307
GRID_MONITOR_HEARTBEAT_TIMEOUT, 307
GRID_MONITOR_NO_STATUS_TIMEOUT, 307
GRID_MONITOR_RETRY_DURATION, 307
GRID_MONITOR, 307, 567
GROUP_ACCEPT_SURPLUS_<groupname>, 297, 350
GROUP_ACCEPT_SURPLUS, 297, 350
GROUP_AUTOREGROUP_<groupname>, 297
GROUP_AUTOREGROUP, 296, 1001, 1017
GROUP_DYNAMIC_MACH_CONSTRAINT, 293
GROUP_NAMES, 296
GROUP_PRIO_FACTOR_<groupname>, 296
GROUP_QUOTA_<groupname>, 296
GROUP_QUOTA_DYNAMIC_<groupname>, 296
GROUP_QUOTA_MAX_ALLOCATION_ROUNDS, 297, 1034
GROUP_QUOTA_ROUND_ROBIN_RATE, 297
GROUP_SORT_EXPR, 297, 351
GSI_AUTHZ_CONF, 318
GSI_DAEMON_CERT, 317, 404
GSI_DAEMON_DIRECTORY, 317, 403, 404
GSI_DAEMON_KEY, 317, 404
GSI_DAEMON_NAME, 317
GSI_DAEMON_PROXY, 318, 404
GSI_DAEMON_TRUSTED_CA_DIR, 318, 404, 574
GSI_DELEGATION_CLOCK_SKEW_ALLOWABLE, 319
GSI_DELEGATION_KEYBITS, 318
GSI_SKIP_HOST_CHECK_CERT_REGEX, 317
GSI_SKIP_HOST_CHECK, 317
GSS_ASSIST_GRIDMAP_CACHE_EXPIRATION, 318
GT2_GAHP, 302
HAD_ARGS, 326
HAD_CONNECTION_TIMEOUT, 326
HAD_CONTROLLEE, 326
HAD_DEBUG, 327
HAD_LIST, 326
HAD_LOG, 327
HAD_UPDATE_INTERVAL, 327
HAD_USE_PRIMARY, 326
HAD_USE_REPLICATION, 327, 459
HAD, 326
HA_<SUBSYS>_LOCK_HOLD_TIME, 325
HA_<SUBSYS>_LOCK_URL, 325
HA_<SUBSYS>_POLL_PERIOD, 325
HA_LOCK_HOLD_TIME, 325
HA_LOCK_URL, 325
HA_POLL_PERIOD, 325
HDFS_ALLOW, 656
HDFS_BACKUPNODE_WEB, 656
HDFS_BACKUPNODE, 656
HDFS_DATANODE_ADDRESS, 655
HDFS_DATANODE_CLASS, 656
HDFS_DATANODE_DIR, 655
HDFS_DATANODE_WEB, 655
HDFS_DENY, 656
HDFS_HOME, 655

HDFS_LOG4J, 656
HDFS_NAMENODE_CLASS, 656
HDFS_NAMENODE_DIR, 655
HDFS_NAMENODE_ROLE, 656
HDFS_NAMENODE_WEB, 655
HDFS_NAMENODE, 655
HDFS_NODETYPE, 656
HDFS_REPLICATION, 656
HDFS_SITE_FILE, 656
HIBERNATE_CHECK_INTERVAL, 258, 500
HIBERNATE, 258, 500
HIBERNATION_OVERRIDE_WOL, 259
HIBERNATION_PLUGIN_ARGS, 259
HIBERNATION_PLUGIN, 259
HIGHPORT, 231, 433
HISTORY_HELPER_MAX_CONCURRENCY, 212
HISTORY_HELPER_MAX_HISTORY, 212
HISTORY_HELPER, 709
HISTORY, 212
HOLD_JOB_IF_CREDENTIAL_EXPIRES, 300
HOSTALLOW_ADMINISTRATOR, 178
HOSTALLOW_NEGOTIATOR_SCHEDD, 554
HOSTALLOW_READ, 178
HOSTALLOW_WRITE, 178, 181
HOSTALLOW..., 225, 759
HOSTALLOW, 225
HOSTDENY, 225
HOSTNAME, 199
HOST_ALIAS, 317
IGNORE_DNS_PROTOCOL_PREFERENCE, 217
IGNORE_LEAF_OOM, 218
IGNORE_NFS_LOCK_ERRORS, 236
IGNORE_TARGET_PROTOCOL_PREFERENCE, 217
IMMUTABLE_JOB_ATTRS, 276
INCLUDE, 208
INTERACTIVE_SUBMIT_FILE, 48, 285
INVALID_LOG_FILES, 286, 821
IN_HIGHPORT, 232, 433
IN_LOWPORT, 232, 433
IPV4_ADDRESS, 199
IPV6_ADDRESS, 199, 714
IP_ADDRESS_IS_V6, 199
IP_ADDRESS, 199
IS_OWNER, 246, 360
IS_VALID_CHECKPOINT_PLATFORM, 246
JAVA5_HOOK_PREPARE_JOB, 544
JAVA_CLASSPATH_ARGUMENT, 257
JAVA_CLASSPATH_DEFAULT, 257
JAVA_CLASSPATH_SEPARATOR, 257
JAVA_EXTRA_ARGUMENTS, 257, 493
JAVA, 257, 492
JOB_DEFAULT_NOTIFICATION, 283
JOB_DEFAULT_REQUESTCPUS, 283, 388
JOB_DEFAULT_REQUESTDISK, 283, 388
JOB_DEFAULT_REQUESTMEMORY, 283, 388, 905, 1000
JOB_EXECDIR_PERMISSIONS, 282
JOB_INHERITS_STARTER_ENVIRONMENT, 280
JOB_IS_FINISHED_COUNT, 267
JOB_IS_FINISHED_INTERVAL, 267
JOB_QUEUE_LOG, 220, 550
JOB_RENICE_INCREMENT, 278, 353
JOB_ROUTER_DEFAULTS, 303
JOB_ROUTER_ENTRIES_CMD, 303, 589
JOB_ROUTER_ENTRIES_FILE, 303
JOB_ROUTER_ENTRIES_REFRESH, 304
JOB_ROUTER_ENTRIES, 303, 589
JOB_ROUTER_HOOK_KEYWORD, 335
JOB_ROUTER_LOCK, 304
JOB_ROUTER_MAX_JOBS, 304
JOB_ROUTER_NAME, 304
JOB_ROUTER_POLLING_PERIOD, 304, 546
JOB_ROUTER_RELEASE_ON_HOLD, 304
JOB_ROUTER_SCHEDD1_NAME, 305
JOB_ROUTER_SCHEDD1_POOL, 305
JOB_ROUTER_SCHEDD1_SPOOL, 305
JOB_ROUTER_SCHEDD2_NAME, 305
JOB_ROUTER_SCHEDD2_POOL, 305
JOB_ROUTER_SCHEDD2_SPOOL, 305
JOB_ROUTER_SOURCE_JOB_CONSTRAINT, 304
JOB_SPOOL_PERMISSIONS, 276
JOB_START_COUNT, 266, 1025
JOB_START_DELAY, 266, 1025
JOB_STOP_COUNT, 266
JOB_STOP_DELAY, 266
JOB_TRANSFORM_<Name>, 275
JOB_TRANSFORM_<name>, 391
JOB_TRANSFORM_NAMES, 275, 391
KBDD_BUMP_CHECK_AFTER_IDLE_TIME, 249

KBDD_BUMP_CHECK_SIZE, 249
KEEP_POOL_HISTORY, 288, 472
KERBEROS_CLIENT_KEYTAB, 322
KERBEROS_MAP_FILE, 407, 412
KERBEROS_SERVER_KEYTAB, 321
KERBEROS_SERVER_PRINCIPAL, 321, 407
KERBEROS_SERVER_SERVICE, 322
KERBEROS_SERVER_USER, 321
KERNEL_TUNING_LOG, 244
KILLING_TIMEOUT, 247, 367, 369, 925, 996
KILL, 245–247, 369
LIBEXEC, 208
LIBVIRT_XML_SCRIPT_ARGS, 323
LIBVIRT_XML_SCRIPT, 323
LIB, 208
LINUX_HIBERNATION_METHOD, 259
LINUX_KERNEL_TUNING_SCRIPT, 244
LOCAL_CONFIG_DIR_EXCLUDE_REGEXP, 210
LOCAL_CONFIG_DIR, 186, 210
LOCAL_CONFIG_FILE, 186, 191, 209, 467, 468
LOCAL_CREDD, 644
LOCAL_DIR, 164, 166, 208
LOCAL_UNIV_EXECUTE, 261
LOCK_DEBUG_LOG_TO_APPEND, 220
LOCK_FILE_UPDATE_INTERVAL, 228
LOCK, 164, 211
LOGS_USE_TIMESTAMP, 221, 550
LOG_ON_NFS_IS_ERROR, 285
LOG, 208, 213, 249, 451
LOWPORT, 231, 433
LSF_GAHP, 302
LeaseManager.CLASSAD_LOG, 306
LeaseManager.DEBUG_ADS, 306
LeaseManager.DEFAULT_MAX_LEASE_DURATION, 306
LeaseManager.GETADS_INTERVAL, 306
LeaseManager.MAX_LEASE_DURATION, 306
LeaseManager.MAX_TOTAL_LEASE_DURATION, 306
LeaseManager.PRUNE_INTERVAL, 306
LeaseManager.QUERY_ADTYPE, 306
LeaseManager.QUERY_CONSTRAINTS, 306
LeaseManager.UPDATE_INTERVAL, 306
MACHINE_RESOURCE_<name>, 254, 255, 379
MACHINE_RESOURCE_INVENTORY_<name>, 255
MACHINE_RESOURCE_INVENTORY_GPUs, 1019
MACHINE_RESOURCE_NAMES, 254, 380
MAIL_FROM, 211
MAIL, 211, 468
MASTER_<SUBSYS>_CONTROLLER, 326
MASTER_<name>_BACKOFF_CEILING, 241
MASTER_<name>_BACKOFF_CONSTANT, 241
MASTER_<name>_BACKOFF_FACTOR, 241
MASTER_<name>_RECOVER_FACTOR, 241
MASTER_ADDRESS_FILE, 243
MASTER_ATTRS, 243
MASTER_BACKOFF_CEILING, 241
MASTER_BACKOFF_CONSTANT, 241
MASTER_BACKOFF_FACTOR, 241
MASTER_CHECK_INTERVAL, 287
MASTER_CHECK_NEW_EXEC_INTERVAL, 181, 240
MASTER_DEBUG, 243
MASTER_HAD_BACKOFF_CONSTANT, 459
MASTER_HA_LIST, 324, 456
MASTER_INSTANCE_LOCK, 243
MASTER_NAME, 207, 242, 808
MASTER_NEW_BINARY_DELAY, 240
MASTER_NEW_BINARY_RESTART, 240
MASTER_RECOVER_FACTOR, 241
MASTER_SHUTDOWN_<Name>, 241
MASTER_UPDATE_INTERVAL, 240
MATCH_TIMEOUT, 357, 363, 368
MAXJOBRETIREMENTTIME, 244, 248, 295, 369
MAX_<SUBSYS>_<LEVEL>_LOG, 224
MAX_<SUBSYS>_LOG, 218, 219, 550
MAX_ACCEPTS_PER_CYCLE, 228
MAX_ACCOUNTANT_DATABASE_SIZE, 291
MAX_CKPT_SERVER_LOG, 446
MAX_CLAIM_ALIVES_MISSED, 248, 267
MAX_CONCURRENT_DOWNLOADS, 263, 264, 1002
MAX_CONCURRENT_UPLOADS, 264, 1002
MAX_C_GAHP_LOG, 302
MAX_DAGMAN_LOG, 88, 315
MAX_DEFAULT_LOG, 218, 219
MAX_DISCARDED_RUN_TIME, 237, 444
MAX_EVENT_LOG, 224
MAX_FILE_DESCRIPTORS, 230, 440
MAX_HAD_LOG, 326
MAX_HISTORY_LOG, 212, 552

MAX_HISTORY_ROTATIONS, 212, 552
MAX_JOBS_PER_OWNER, 263
MAX_JOBS_PER_SUBMISSION, 263
MAX_JOBS_RUNNING, 50, 262, 434, 1024
MAX_JOBS_SUBMITTED, 263
MAX_JOB_MIRROR_UPDATE_LAG, 304
MAX_JOB_QUEUE_LOG_ROTATIONS, 212, 550
MAX_NEXT_JOB_START_DELAY, 266, 911, 997
MAX_NUM_<SUBSYS>_LOG, 219, 550
MAX_NUM_CPUS, 250
MAX_NUM_SCHEDD_AUDIT_LOG, 275, 551
MAX_NUM_SHADOW_LOG, 728
MAX_NUM_SHARED_PORT_AUDIT_LOG, 333, 551
MAX_PENDING_STARTD_CONTACTS, 263
MAX_PERIODIC_EXPR_INTERVAL, 270
MAX_PROCD_LOG, 298
MAX_REAPS_PER_CYCLE, 228
MAX_REPLICATION_LOG, 327
MAX_RUNNING_SCHEDULER_JOBS_PER_OWNER, 263, 706, 710
MAX_SCHEDD_AUDIT_LOG, 275, 551
MAX_SHADOW_EXCEPTIONS, 263
MAX_SHADOW_STATS_LOG, 277
MAX_SHARED_PORT_AUDIT_LOG, 333, 551
MAX_SLOT_TYPES, 254
MAX_STARTER_STATS_LOG, 283
MAX_TIME_SKIP, 227
MAX_TRACKING_GID, 299, 485
MAX_TRANSFERER_LOG, 328
MAX_TRANSFER_INPUT_MB, 264, 907, 992, 997
MAX_TRANSFER_LIFETIME, 327
MAX_TRANSFER_OUTPUT_MB, 265, 907, 993, 997
MAX_TRANSFER_QUEUE_AGE, 265
MAX_VM_GAHP_LOG, 322
MEMORY_USAGE_METRIC_VM, 282
MEMORY_USAGE_METRIC, 282
MEMORY, 250
MIN_TRACKING_GID, 298, 485
MODIFY_REQUEST_EXPR_REQUESTCPUS, 256, 388
MODIFY_REQUEST_EXPR_REQUESTDISK, 256, 388
MODIFY_REQUEST_EXPR_REQUESTMEMORY, 256, 388
MOUNT_UNDER_SCRATCH, 252
MUST_MODIFY_REQUEST_EXPRS, 256
MYPROXY_GET_DELEGATION, 328, 566
MachineMaxVacateTime, 245–247, 367, 369
NAMED_CHROOT, 280
NEGOTIATE_ALL_JOBS_IN_CLUSTER, 270, 346
NEGOTIATION_CYCLE_STATS_LENGTH, 291
NEGOTIATOR_ADDRESS_FILE, 432
NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION, 298, 349, 351
NEGOTIATOR_CONSIDER_EARLY_PREEMPTION, 248, 267, 295, 369
NEGOTIATOR_CONSIDER_PREEMPTION, 295
NEGOTIATOR_CROSS_SLOT_PRIOS, 708
NEGOTIATOR_CYCLE_DELAY, 291
NEGOTIATOR_DEBUG, 293
NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES, 291
NEGOTIATOR_HOST, 207
NEGOTIATOR_IGNORE_USER_PRIORITIES, 581
NEGOTIATOR_INFORM_STARTD, 292
NEGOTIATOR_INTERVAL, 290
NEGOTIATOR_MATCHLIST_CACHING, 295, 581
NEGOTIATOR_MATCH_EXPRS, 294
NEGOTIATOR_MATCH_LOG, 224, 552
NEGOTIATOR_MAX_TIME_PER_CYCLE, 294
NEGOTIATOR_MAX_TIME_PER_PIESPIN, 294
NEGOTIATOR_MAX_TIME_PER_SCHEDD, 294
NEGOTIATOR_MAX_TIME_PER_SUBMITTER, 293, 1034
NEGOTIATOR_POST_JOB_RANK, 292
NEGOTIATOR_PRE_JOB_RANK, 292
NEGOTIATOR_READ_CONFIG_BEFORE_CYCLE, 295
NEGOTIATOR_RESOURCE_REQUEST_LIST_SIZE, 294
NEGOTIATOR_SLOT_CONSTRAINT, 293
NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT, 293, 1032
NEGOTIATOR_SOCKET_CACHE_SIZE, 291, 434
NEGOTIATOR_TIMEOUT, 291
NEGOTIATOR_TRIM_SHUTDOWN_THRESHOLD, 293
NEGOTIATOR_UPDATE_AFTER_CYCLE, 295
NEGOTIATOR_UPDATE_INTERVAL, 291

NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT, 233
NEGOTIATOR_USE_SLOT_WEIGHTS, 297
NEGOTIATOR_USE_WEIGHTED_DEMAND, 297
NETWORK_HOSTNAME, 230
NETWORK_INTERFACE, 230, 437, 439, 443
NETWORK_MAX_PENDING_CONNECTS, 214
NICE_USER_PRIO_FACTOR, 291, 343
NONBLOCKING_COLLECTOR_UPDATE, 232
NORDUGRID_GAHP, 302
NOT_RESPONDING_TIMEOUT, 227
NOT_RESPONDING_WANT_CORE, 227
NO_DNS, 213, 444, 722
NUM_CLAIMS, 257
NUM_CPUS, 249, 256, 378
NUM_SLOTS_TYPE_<N>, 256
NUM_SLOTS, 256, 377
OBITUARY_LOG_LENGTH, 240
OFFLINE_EXPIRE_ADS_AFTER, 260, 501
OFFLINE_LOG, 260, 501
OFFLINE_MACHINE_RESOURCE_<name>, 255
OPENMPI_EXCLUDE_NETWORK_INTERFACES, 74, 261
OPENMPI_INSTALL_PATH, 73, 261
OPEN_VERB_FOR_<EXT>_FILES, 214
OPSYS_AND_VER, 200
OPSYS_VER, 200
OPSYS, 200
OUT_HIGHPORT, 232, 433
OUT_LOWPORT, 232, 433
PASSWD_CACHE_REFRESH, 214
PBS_GAHP, 302
PERIODIC_CHECKPOINT, 245, 523
PERIODIC_EXPR_INTERVAL, 270
PERIODIC_EXPR_TIMESLICE, 270
PERIODIC_MEMORY_SYNC, 277
PERSISTENT_CONFIG_DIR, 225
PER_JOB_HISTORY_DIR, 272, 726
PER_JOB_NAMESPACES, 282
PID, 201
PIPE_BUFFER_MAX, 228
POLLING_INTERVAL, 247, 364
POOL_HISTORY_DIR, 288, 472
POOL_HISTORY_MAX_STORAGE, 288, 472
POOL_HISTORY_SAMPLING_INTERVAL, 288
PPID, 201
PREEMPTION_RANK_STABLE, 293, 344
PREEMPTION_RANK, 293
PREEMPTION_REQUIREMENTS_STABLE, 292, 344
PREEMPTION_REQUIREMENTS, 60, 292, 295, 343, 832
PREEMPT, 244, 369, 541
PREEN_ADMIN, 286, 821
PREEN_ARGS, 240
PREEN_INTERVAL, 240
PREEN, 239
PREFER_IPV4, 217, 443
PREFER_OUTBOUND_IPV4, 217
PRIORITY_HALFLIFE, 60, 291, 342, 345
PRIVATE_NETWORK_INTERFACE, 231, 437
PRIVATE_NETWORK_NAME, 228, 230, 437
PROCD_ADDRESS, 298
PROCD_LOG, 298
PROCD_MAX_SNAPSHOT_INTERVAL, 298
PROPORTIONAL_SWAP_ASSSIGNMENT, 489
PROTECTED_JOB_ATTRS, 276
PUBLISH_OBITUARIES, 240
ParallelSchedulingGroup, 272, 478
QUERY_TIMEOUT, 287
QUEUE_ALL_USERS_TRUSTED, 268
QUEUE_CLEAN_INTERVAL, 268, 550
QUEUE_SUPER_USERS, 268
QUEUE_SUPER_USER_MAY_IMPERSONATE, 268, 305
Q_QUERY_TIMEOUT, 213
RANK_FACTOR, 477
RANK, 245, 370, 476, 477
RELEASE_DIR, 165, 207, 468
REMOTE_GROUP_RESOURCES_IN_USE, 725
REMOTE_PRIO_FACTOR, 291, 343
REMOVE_SIGNIFICANT_ATTRIBUTES, 274
REPLICATION_ARGS, 327
REPLICATION_DEBUG, 327
REPLICATION_INTERVAL, 327
REPLICATION_LIST, 327
REPLICATION_LOG, 327
REPLICATION, 327
REQUEST_CLAIM_TIMEOUT, 267
REQUIRE_LOCAL_CONFIG_FILE, 210
RESERVED_DISK, 211, 1006
RESERVED_MEMORY, 250

RESERVED_SWAP, 55, 211
RESERVE_AFS_CACHE, 235
ROOSTER_INTERVAL, 331
ROOSTER_MAX_UNHIBERNATE, 331
ROOSTER_UNHIBERNATE_RANK, 331
ROOSTER_UNHIBERNATE, 331
ROOSTER_WAKEUP_CMD, 331
ROTATE_HISTORY_DAILY, 274, 550
ROTATE_HISTORY_MONTHLY, 274, 550
RUNBENCHMARKS, 250, 363, 368
RUN_FILETRANSFER_PLUGINS_WITH_ROOT, 281
RUN, 208
RemoteSysCpu, 712
RemoteUserCpu, 712
Requirements, 261, 262
SBIN, 208
SCHEDD_ADDRESS_FILE, 269
SCHEDD_ASSUME_NEGOTIATOR_GONE, 271
SCHEDD_ATTRS, 269
SCHEDD_AUDIT_LOG, 274, 551
SCHEDD_BACKUP_SPOOL, 272, 550
SCHEDD_CLUSTER_INCREMENT_VALUE, 273
SCHEDD_CLUSTER_INITIAL_VALUE, 273
SCHEDD_CLUSTER_MAXIMUM_VALUE, 273
SCHEDD_COLLECT_STATS_BY_<Name>, 274
SCHEDD_COLLECT_STATS_FOR_<Name>, 274
SCHEDD_CRON_<JobName>_ARGS, 338
SCHEDD_CRON_<JobName>_CWD, 338
SCHEDD_CRON_<JobName>_ENV, 338
SCHEDD_CRON_<JobName>_EXECUTABLE, 336
SCHEDD_CRON_<JobName>_JOB_LOAD, 337
SCHEDD_CRON_<JobName>_KILL, 338
SCHEDD_CRON_<JobName>_MODE, 336
SCHEDD_CRON_<JobName>_PERIOD, 336
SCHEDD_CRON_<JobName>_PREFIX, 336
SCHEDD_CRON_<JobName>_RECONFIG_RERUN, 337
SCHEDD_CRON_<JobName>_RECONFIG, 337
SCHEDD_CRON_CONFIG_VAL, 335
SCHEDD_CRON_JOBLIST, 336
SCHEDD_CRON_MAX_JOB_LOAD, 337
SCHEDD_CRON_NAME, 335
SCHEDD_DEBUG, 269
SCHEDD_ENABLE_SSH_TO_JOB, 329
SCHEDD_EXECUTE, 269
SCHEDD_EXPIRE_STATS_BY_<Name>, 274
SCHEDD_HOST, 207, 710, 796
SCHEDD_INTERVAL_TIMESLICE, 266
SCHEDD_INTERVAL, 154, 266
SCHEDD_JOB_QUEUE_LOG_FLUSH_DELAY, 273
SCHEDD_LOCK, 269
SCHEDD_MIN_INTERVAL, 266
SCHEDD_NAME, 207, 243, 269, 457
SCHEDD_PREEMPTION_RANK, 272, 477
SCHEDD_PREEMPTION_REQUIREMENTS, 272, 477
SCHEDD_QUERY_WORKERS, 265
SCHEDD_RESTART_REPORT, 275
SCHEDD_ROUND_ATTR_<xxxx>, 271
SCHEDD_SEND_VACATE_VIA_TCP, 273
SCHEDD_USES_STARTD_FOR_LOCAL_UNIVERSE, 262
SCHEDD_USE_SLOT_WEIGHT, 275, 731
SCHED_UNIV_RENICE_INCREMENT, 268
SEC_*_AUTHENTICATION_METHODS, 317
SEC_*_AUTHENTICATION, 316
SEC_*_CRYPTO_METHODS, 317
SEC_*_ENCRYPTION, 316
SEC_*_INTEGRITY, 316
SEC_*_NEGOTIATION, 316
SEC_<access-level>_SESSION_DURATION, 319
SEC_<access-level>_SESSION_LEASE, 319
SEC_DEFAULT_AUTHENTICATION_TIMEOUT, 320
SEC_DEFAULT_SESSION_DURATION, 319
SEC_DEFAULT_SESSION_LEASE, 319
SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION, 321, 418
SEC_INVALIDATE_SESSIONS_VIA_TCP, 319
SEC_PASSWORD_FILE, 320, 408
SEC_TCP_SESSION_DEADLINE, 320
SEC_TCP_SESSION_TIMEOUT, 320
SENDMAIL, 211
SETTABLE_ATTRS_<PERMISSION-LEVEL>, 225, 424, 425, 759
SETTABLE_ATTRS_ADMINISTRATOR, 424
SETTABLE_ATTRS_CONFIG, 202, 225, 424

SETTABLE_ATTRS_OWNER, 424
SETTABLE_ATTRS_WRITE, 424
SGE_GAHP, 303
SHADOW_CHECKPROXY_INTERVAL, 277, 318
SHADOW_DEBUG, 276
SHADOW_JOB_CLEANUP_RETRY_DELAY, 277
SHADOW_LAZY_QUEUE_UPDATE, 276
SHADOW_LOCK, 276
SHADOW_MAX_JOB_CLEANUP_RETRIES, 277
SHADOW_QUEUE_UPDATE_INTERVAL, 276
SHADOW_RENICE_INCREMENT, 267
SHADOW_RUN_UNKNOWN_USER_JOBS, 277
SHADOW_SIZE_ESTIMATE, 211, 267
SHADOW_STATS_LOG, 223, 277
SHADOW_WORKLIFE, 277
SHADOW, 261
SHARED_PORT_ARGS, 332
SHARED_PORT_AUDIT_LOG, 332, 333, 551
SHARED_PORT_DAEMON_AD_FILE, 332
SHARED_PORT_DEFAULT_ID, 229
SHARED_PORT_MAX_WORKERS, 332
SHARED_PORT_PORT, 331
SHARED_PORT, 229, 435
SHELL, 873
SHUTDOWN_FAST_TIMEOUT, 240
SHUTDOWN_GRACEFUL_TIMEOUT, 225, 248
SIGNIFICANT_ATTRIBUTES, 274, 346
SINGULARITY_BIND_EXPR, 283
SINGULARITY_IMAGE_EXPR, 283
SINGULARITY_JOB, 283
SINGULARITY_TARGET_DIR, 283
SINGULARITY, 283
SKIP_WINDOWS_LOGON_NETWORK, 299
SLOT<N>_CPU_AFFINITY, 281
SLOT<N>_EXECUTE, 209, 378
SLOT<N>_JOB_HOOK_KEYWORD, 333, 542
SLOT<N>_USER, 234, 429
SLOTS_CONNECTED_TO_CONSOLE, 253, 381, 1005
SLOTS_CONNECTED_TO_KEYBOARD, 253, 381, 1008
SLOT_TYPE_<N>_PARTITIONABLE, 254, 386
SLOT_TYPE_<N>, 254, 378, 380
SLOT_TYPE_<n>_STARTD_ATTRS, 730
SLOT_WEIGHT, 257, 389, 704
SLOW_CKPT_SPEED, 277
SMTP_SERVER, 211
SOAP_LEAVE_IN_QUEUE, 328, 592
SOAP_SSL_CA_DIR, 329, 574
SOAP_SSL_CA_FILE, 329, 574
SOAP_SSL_DH_FILE, 329
SOAP_SSL_SERVER_KEYFILE_PASSWORD, 329
SOAP_SSL_SERVER_KEYFILE, 329
SOAP_SSL_SKIP_HOST_CHECK, 329
SOCKET_LISTEN_BACKLOG, 228
SOFT_UID_DOMAIN, 234, 426
SPOOL, 208
SSH_TO_JOB_<SSH-CLIENT>_CMD, 329
SSH_TO_JOB_SSHD_ARGS, 330
SSH_TO_JOB_SSHD_CONFIG_TEMPLATE, 330
SSH_TO_JOB_SSHD, 330
SSH_TO_JOB_SSH_KEYGEN_ARGS, 330
SSH_TO_JOB_SSH_KEYGEN, 330
STARTD_ADDRESS_FILE, 249
STARTD_AD_REEVAL_EXPR, 295
STARTD_ATTRS, 249, 384, 425, 478, 730
STARTD_AVAIL_CONFIDENCE, 260
STARTD_CLAIM_ID_FILE, 249
STARTD_COMPUTE_AVAIL_STATS, 260
STARTD_CRON_<JobName>_ARGS, 338
STARTD_CRON_<JobName>_CWD, 338
STARTD_CRON_<JobName>_ENV, 338
STARTD_CRON_<JobName>_EXECUTABLE, 336
STARTD_CRON_<JobName>_JOB_LOAD, 337
STARTD_CRON_<JobName>_KILL, 338
STARTD_CRON_<JobName>_MODE, 336
STARTD_CRON_<JobName>_PERIOD, 336
STARTD_CRON_<JobName>_PREFIX, 336
STARTD_CRON_<JobName>_RECONFIG_RERUN, 337
STARTD_CRON_<JobName>_RECONFIG, 337
STARTD_CRON_<JobName>_SLOTS, 336
STARTD_CRON_AUTOPUBLISH, 335
STARTD_CRON_CONFIG_VAL, 335
STARTD_CRON_JOBLIST, 336
STARTD_CRON_MAX_JOB_LOAD, 337
STARTD_CRON_NAME, 335
STARTD_DEBUG, 249
STARTD_EXPRS, 730
STARTD_HAS_BAD_UTMP, 248

STARTD_HISTORY, 247, 731
STARTD_JOB_ATTRS, 249, 730
STARTD_JOB_EXPRS, 292
STARTD_JOB_HOOK_KEYWORD, 333, 542
STARTD_MAX_AVAIL_PERIOD_SAMPLES, 260
STARTD_NAME, 250
STARTD_NOCLAIM_SHUTDOWN, 250
STARTD_PARTITIONABLE_SLOT_ATTRS, 246
STARTD_PUBLISH_DOTNET, 258
STARTD_PUBLISH_WINREG, 251
STARTD_RESOURCE_PREFIX, 253
STARTD_SENDS_ALIVES, 267
STARTD_SHOULD_WRITE_CLAIM_ID_FILE, 249
STARTD_SLOT_ATTRS, 253
STARTD_VM_ATTRS, 253
STARTD_VM_EXPRS, 253
STARTER_ALLOW_RUNAS_OWNER, 234, 428, 429, 484
STARTER_CHOOSES_CKPT_SERVER, 237, 446
STARTER_DEBUG, 278
STARTER_INITIAL_UPDATE_INTERVAL, 541
STARTER_JOB_ENVIRONMENT, 280
STARTER_JOB_HOOK_KEYWORD, 542
STARTER_LOCAL_LOGGING, 278
STARTER_LOCAL, 261
STARTER_LOG_NAME_APPEND, 278
STARTER_RLIMIT_AS, 282
STARTER_STATS_LOG, 223, 282
STARTER_UPDATE_INTERVAL_TIMESLICE, 278
STARTER_UPDATE_INTERVAL, 278, 541
STARTER_UPLOAD_TIMEOUT, 280
STARTER, 247
START_BACKFILL, 252, 363, 370, 479
START_DAEMONS, 240
START_LOCAL_UNIVERSE, 261, 1028
START_MASTER, 240
START_SCHEDULER_UNIVERSE, 262, 1028
START, 244, 252, 352, 368, 476
STATE_FILE, 327
STATISTICS_TO_PUBLISH_LIST, 216
STATISTICS_TO_PUBLISH, 215, 341, 1030–1032
STATISTICS_WINDOW_QUANTUM_<collection>, 217
STATISTICS_WINDOW_QUANTUM, 217
STATISTICS_WINDOW_SECONDS_<collection>, 216
STATISTICS_WINDOW_SECONDS, 216, 1028
STRICT_CLASSAD_EVALUATION, 215, 505
SUBMIT_ATTRS, 285, 429, 488, 723, 730
SUBMIT_EXPRS, 488
SUBMIT_MAX_PROCS_IN_CLUSTER, 285
SUBMIT_PUBLISH_WINDOWS_OSVERSIONINFO, 708
SUBMIT_REQUIREMENT_<Name>_REASON, 275, 392
SUBMIT_REQUIREMENT_<Name>, 275, 392
SUBMIT_REQUIREMENT_NAMES, 275, 392
SUBMIT_SEND_RESCHEDULE, 285
SUBMIT_SKIP_FILECHECKS, 285
SUBSYSTEM, 199
SUSPEND, 244, 369
SYSAPI_GET_LOADAVG, 214
SYSTEM_IMMUTABLE_JOB_ATTRS, 276
SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH, 269
SYSTEM_JOB_MACHINE_ATTRS, 268, 269, 925, 987
SYSTEM_PERIODIC_HOLD_REASON, 271
SYSTEM_PERIODIC_HOLD_SUBCODE, 271
SYSTEM_PERIODIC_HOLD, 270, 992
SYSTEM_PERIODIC_RELEASE, 271
SYSTEM_PERIODIC_REMOVE, 271
SYSTEM_PROTECTED_JOB_ATTRS, 276
SYSTEM_VALID_SPOOL_FILES, 286, 821
SlotWeight, 297
TCP_FORWARDING_HOST, 230, 231
TCP_KEEPALIVE_INTERVAL, 217
TCP_UPDATE_COLLECTORS, 232, 442
TEMP_DIR, 209
TILDE, 199
TMP_DIR, 209
TOOL_DEBUG, 223
TOUCH_LOG_INTERVAL, 221, 550
TRANSFERER_DEBUG, 328
TRANSFERER_LOG, 328
TRANSFERER, 328
TRANSFER_IO_REPORT_INTERVAL, 265
TRANSFER_IO_REPORT_TIMESPANS, 264, 265, 1029–1032

TRANSFER_QUEUE_USER_EXPR, 264, 1029–1032
TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN, 224
TRUNC_<SUBSYS>_LOG_ON_OPEN, 219, 224, 550
TRUST_UID_DOMAIN, 234
UDP_LOOPBACK_FRAGMENT_SIZE, 233
UDP_NETWORK_FRAGMENT_SIZE, 233
UID_DOMAIN, 201, 233, 426, 427, 438, 902
UNAME_ARCH, 200
UNAME_OPSYS, 200
UNHIBERNATE, 259, 331, 501
UNICORE_GAHP, 302
UPDATE_COLLECTOR_WITH_TCP, 232, 442
UPDATE_INTERVAL, 247, 335, 362, 454
UPDATE_OFFSET, 247
UPDATE_VIEW_COLLECTOR_WITH_TCP, 232, 442
USERLOG_FILE_CACHE_CLEAR_INTERVAL, 220
USERLOG_FILE_CACHE_MAX, 220
USERNAME, 201
USER_CONFIG_FILE, 185, 210
USER_JOB_WRAPPER, 279, 487
USE_AFS, 236
USE_CKPT_SERVER, 237, 446
USE_CLONE_TO_CREATE_PROCESSES, 227
USE_GID_PROCESS_TRACKING, 298, 485
USE_NFS, 235
USE_PID_NAMESPACES, 282
USE_PROCD, 298, 486
USE_PROCESS_GROUPS, 244
USE_PSS, 282
USE_RESOURCE_REQUEST_COUNTS, 294
USE_SHARED_PORT, 229, 331
USE_VISIBLE_DESKTOP, 280, 647
VALID_COD_USERS, 527
VALID_SPOOL_FILES, 286, 325, 456, 821
VMP_HOST_MACHINE, 324, 474
VMP_VM_LIST, 324, 474
VMWARE_BRIDGE_NETWORKING_TYPE, 324
VMWARE_LOCAL_SETTINGS_FILE, 324
VMWARE_NAT_NETWORKING_TYPE, 324
VMWARE_NETWORKING_TYPE, 324
VMWARE_PERL, 323
VMWARE_SCRIPT, 323
VM_GAHP_LOG, 322
VM_GAHP_REQ_TIMEOUT, 322
VM_GAHP_SERVER, 322
VM_MAX_NUMBER, 322, 1016
VM_MEMORY, 322, 1016
VM_NETWORKING_BRIDGE_INTERFACE, 323
VM_NETWORKING_DEFAULT_TYPE, 323
VM_NETWORKING_TYPE, 323
VM_NETWORKING, 323
VM_RECHECK_INTERVAL, 322
VM_SOFT_SUSPEND, 322
VM_STATUS_INTERVAL, 322
VM_TYPE, 322, 494, 1017
VM_UNIV_NOBODY_USER, 323
WALL_CLOCK_CKPT_INTERVAL, 268
WANT_HOLD_REASON, 245
WANT_HOLD_SUBCODE, 245
WANT_HOLD, 244, 992
WANT_SUSPEND, 246, 368
WANT_UDP_COMMAND_SOCKET, 214, 292
WANT_VACATE, 246, 247, 369
WARN_ON_UNUSED_SUBMIT_FILE_MACROS, 285, 896
WEB_ROOT_DIR, 328
WEIGHTED_JOBS_RUNNING, 725
WINDOWED_STAT_WIDTH, 266
WINDOWS_FIREWALL_FAILURE_RETRY, 243
WINDOWS_RMDIR_OPTIONS, 338
WINDOWS_RMDIR, 338
WeightedJobsRunning, 726
WorkHours, 373
XEN_BOOTLOADER, 324
include command, 192
ALLOW_ADMINISTRATOR, 415
ALLOW_ADVERTISE_MASTER, 415
ALLOW_ADVERTISE_SCHEDD, 415
ALLOW_ADVERTISE_STARTD, 415
ALLOW_CLIENT, 415
ALLOW_CONFIG, 415
ALLOW_DAEMON, 415
ALLOW_NEGOTIATOR, 415
ALLOW_OWNER, 415
ALLOW_READ, 415
ALLOW_SOAP, 415
ALLOW_WRITE, 415

COLLECTOR_ADDRESS_FILE, 225
COLLECTOR_ARGS, 239
COLLECTOR_SUPER_ADDRESS_FILE, 226
DENY_ADMINISTRATOR, 415
DENY_ADVERTISE_MASTER, 415
DENY_ADVERTISE_SCHEDD, 415
DENY_ADVERTISE_STARTD, 415
DENY_CLIENT, 415
DENY_CONFIG, 415
DENY_DAEMON, 415
DENY_NEGOTIATOR, 415
DENY_OWNER, 415
DENY_READ, 415
DENY_SOAP, 415
DENY_WRITE, 415
IS_VALID_CHECKPOINT_PLATFORM, 353
NEGOTIATOR_ADDRESS_FILE, 225
NEGOTIATOR_ARGS, 239
RANK, 354
SCHEDD_ARGS, 239
SCHEDD_SUPER_ADDRESS_FILE, 226
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS, 401
SEC_ADMINISTRATOR_AUTHENTICATION, 401
SEC_ADMINISTRATOR_CRYPTO_METHODS, 413
SEC_ADMINISTRATOR_ENCRYPTION, 412
SEC_ADMINISTRATOR_INTEGRITY, 414
SEC_ADVERTISE_MASTER_AUTHENTICATION_METHODS, 401
SEC_ADVERTISE_MASTER_AUTHENTICATION, 401
SEC_ADVERTISE_MASTER_CRYPTO_METHODS, 413
SEC_ADVERTISE_MASTER_ENCRYPTION, 412
SEC_ADVERTISE_MASTER_INTEGRITY, 414
SEC_ADVERTISE_SCHEDD_AUTHENTICATION_METHODS, 401
SEC_ADVERTISE_SCHEDD_AUTHENTICATION, 401
SEC_ADVERTISE_SCHEDD_CRYPTO_METHODS, 413
SEC_ADVERTISE_SCHEDD_ENCRYPTION, 412
SEC_ADVERTISE_SCHEDD_INTEGRITY, 414
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS, 401
SEC_ADVERTISE_STARTD_AUTHENTICATION, 401
SEC_ADVERTISE_STARTD_CRYPTO_METHODS, 413
SEC_ADVERTISE_STARTD_ENCRYPTION, 412
SEC_ADVERTISE_STARTD_INTEGRITY, 414
SEC_CLIENT_AUTHENTICATION_METHODS, 401
SEC_CLIENT_AUTHENTICATION, 401
SEC_CLIENT_CRYPTO_METHODS, 413
SEC_CLIENT_ENCRYPTION, 412
SEC_CLIENT_INTEGRITY, 413
SEC_CONFIG_AUTHENTICATION_METHODS, 401
SEC_CONFIG_AUTHENTICATION, 401
SEC_CONFIG_CRYPTO_METHODS, 413
SEC_CONFIG_ENCRYPTION, 412
SEC_CONFIG_INTEGRITY, 414
SEC_DAEMON_AUTHENTICATION_METHODS, 401
SEC_DAEMON_AUTHENTICATION, 401
SEC_DAEMON_CRYPTO_METHODS, 413
SEC_DAEMON_ENCRYPTION, 412
SEC_DAEMON_INTEGRITY, 414
SEC_DEFAULT_AUTHENTICATION_METHODS, 401
SEC_DEFAULT_AUTHENTICATION, 401
SEC_DEFAULT_CRYPTO_METHODS, 413
SEC_DEFAULT_ENCRYPTION, 412
SEC_DEFAULT_INTEGRITY, 413, 414
SEC_NEGOTIATOR_AUTHENTICATION_METHODS, 401
SEC_NEGOTIATOR_AUTHENTICATION, 401
SEC_NEGOTIATOR_CRYPTO_METHODS, 413
SEC_NEGOTIATOR_INTEGRITY, 414
SEC_OWNER_AUTHENTICATION_METHODS, 401
SEC_OWNER_AUTHENTICATION, 401
SEC_OWNER_CRYPTO_METHODS, 413
SEC_OWNER_ENCRYPTION, 412
SEC_OWNER_INTEGRITY, 414
SEC_READ_AUTHENTICATION_METHODS, 401
SEC_READ_AUTHENTICATION, 401
SEC_READ_CRYPTO_METHODS, 413

SEC_READ_ENCRYPTION, 412
SEC_READ_INTEGRITY, 414
SEC_WRITE_AUTHENTICATION_METHODS, 401
SEC_WRITE_AUTHENTICATION, 401
SEC_WRITE_CRYPTO_METHODS, 413
SEC_WRITE_ENCRYPTION, 412
SEC_WRITE_INTEGRITY, 414
SHARED_PORT_MAX_FILE_DESCRIPTORS, 229
STARTD_ARGS, 239
STARTD_ATTRS, 226
STARTD_EXPRS, 226
configuration of source code contrib
Quill configuration variables, 660
configuration: introduction, 184
configuration: macros, 206
configuration: templates, 201
CONSOLE_DEVICES macro, 170, 248, 468
consumption policy, 388
CONSUMPTION_<Resource> macro, 257
CONSUMPTION_POLICY macro, 257
CONTINUE macro, 245, 369
contrib module
HTCondorView client, 683
CORE_FILE_NAME macro, 228
COUNT_HYPERTHREAD_CPUS macro, 200, 250
CpusProvisioned
job ClassAd attribute, 1004
cream, 576
CREAM_GAHP macro, 302
CREATE_CORE_FILES macro, 213
CREATE_LOCKS_ON_LOCAL_DISK macro, 209, 221
CRED_MIN_TIME_LEFT macro, 285
CREDD_CACHE_LOCALLY macro, 299
CREDD_HOST macro, 299
CREDD_POLLING_TIMEOUT macro, 299
Crondor, 150
CronTab job scheduling, 150
crontab program, 685
CumulativeRemoteSysCpu
job ClassAd attribute, 1000
CumulativeRemoteUserCpu
job ClassAd attribute, 1000
CumulativeSlotTime
job ClassAd attribute, 987
CumulativeSuspensionTime
job ClassAd attribute, 987
CumulativeTransferTime
job ClassAd attribute, 987
CURB_MATCHMAKING macro, 263
current working directory, 430
CurrentHosts
job ClassAd attribute, 987
cwd
of jobs, 430

D_COMMAND macro, 419
D_SECURITY macro, 419
daemon
condor_ckpt_server, 160, 444
condor_collector, 159
condor_credd, 160, 299, 643
condor_defrag, 160, 390
condor_gangliad, 340, 452
condor_gridmanager, 160
condor_had, 160, 458
condor_hdfs, 160
condor_job_router, 160, 583
condor_kbdd, 159, 470
condor_lease_manager, 160
condor_master, 158, 808
condor_negotiator, 159
condor_procd, 160
condor_replication, 160, 458
condor_rooster, 160, 501
condor_schedd, 159
condor_shadow, 159
condor_shared_port, 160, 435
condor_startd, 158, 351, 352
condor_starter, 159
condor_transferer, 160, 458
descriptions, 158
running as root, 155
Daemon ClassAd Hooks, 547
DAEMON_LIST macro, 229, 238, 435, 445, 468, 808
DAEMON_SHUTDOWN macro, 227, 1044
DAEMON_SHUTDOWN_FAST macro, 227
DAEMON_SOCKET_DIR macro, 331, 332, 551
daemoncore, 448–451
command line arguments, 450
Unix signals, 450

DAG input file
ABORT-DAG-ON command, 90
ALL_NODES option, 121
CATEGORY command, 97
command order, 83
Composing workflows, 102
CONFIG command, 98
CONNECT command, 117
DATA command, 78
DOT command, 126
FINAL command, 119
INCLUDE command, 101
JOB command, 77
JOBSTATE_LOG command, 129
MAXJOBS command, 97
NODE_STATUS_FILE command, 127
PARENT . . . CHILD command, 78
PIN_IN command, 117
PIN_OUT command, 117
PRE_SKIP command, 83
PRIORITY command, 95
RETRY command, 89
SCRIPT command, 79
SET_JOB_ATTR command, 99
SPLICE command, 107
SUBDAG command, 103
VARS command, 91
DAG_InRecovery
job ClassAd attribute, 1003
DAG_NodesDone
job ClassAd attribute, 1003
DAG_NodesFailed
job ClassAd attribute, 1003
DAG_NodesPostrun
job ClassAd attribute, 1003
DAG_NodesPrerun
job ClassAd attribute, 1003
DAG_NodesQueued
job ClassAd attribute, 1003
DAG_NodesReady
job ClassAd attribute, 1003
DAG_NodesTotal
job ClassAd attribute, 1003
DAG_NodesUnready
job ClassAd attribute, 1003
DAG_Status
job ClassAd attribute, 1003
DAGMan, 75–139
aborting a DAG, 90
accounting groups, 139
command order, 83
Composing workflows, 102
configuration specific to a DAG, 98
connecting DAG splices, 117
DAG INCLUDE command, 101
DAG input file, 77
DAG monitoring, 88
DAG recovery, 125
DAG removal, 88
DAG status in a job ClassAd, 133
DAG submission, 84
$DAG_STATUS value, 82
DAGs within DAGs, 103
describing dependencies, 78
difference between Rescue DAG and DAG recovery, 125
example submit description file, 84
$FAILED_COUNT value, 82
file paths in DAGs, 86
FINAL node, 119
$JOB value, 81
$JOBID value, 81
jobstate.log file, 129
large numbers of jobs, 133
machine-readable event history, 129
$MAX_RETRIES value, 81
node job submit description file, 84
node priorities, 95
node status file, 127
optimization of submit time, 99
POST script, 79
PRE and POST scripts, 79
PRE script, 79
$PRE_SCRIPT_RETURN value, 81
rescue DAG, 122
$RETRY value, 81
retrying failed nodes, 89
$RETURN value, 81
setting ClassAd attributes in a DAG, 99
single submission of multiple, independent DAGs, 100
skipping node execution, 83

splicing DAGs, 107
suspending a running DAG, 88
terminology, 76
throttling, 85
throttling nodes by category, 97
VARS (macro for submit description file), 91
VARS (use of special characters), 93
visualizing DAGs, 126
workflow metrics, 136
DAGMan configuration: debug output, 315
DAGMan configuration: general, 308
DAGMan configuration: HTCondor attributes, 316
DAGMan configuration: log files, 313
DAGMan configuration: metrics, 316
DAGMan configuration: priority, node semantics, 309
DAGMan configuration: rescue/retry, 312
DAGMan configuration: submission/removal, 310
DAGMan configuration: throttling, 309
DAGMAN_ABORT_DUPLICATES macro, 308
DAGMAN_ABORT_ON_SCARY_SUBMIT macro, 311
DAGMAN_ALLOW_EVENTS macro, 314
DAGMAN_ALLOW_LOG_ERROR macro, 313
DAGMAN_ALWAYS_RUN_POST macro, 309
DAGMAN_ALWAYS_USE_NODE_LOG macro, 314, 552
DAGMAN_AUTO_RESCUE macro, 312
DAGMAN_CONDOR_RM_EXE macro, 311
DAGMAN_CONDOR_SUBMIT_EXE macro, 311
DAGMAN_CONFIG_FILE macro, 308
DAGMAN_COPY_TO_SPOOL macro, 316
DAGMAN_DEBUG macro, 315
DAGMAN_DEBUG_CACHE_ENABLE macro, 315
DAGMAN_DEBUG_CACHE_SIZE macro, 315
DAGMAN_DEFAULT_NODE_LOG macro, 313, 315, 552
DAGMAN_DEFAULT_PRIORITY macro, 309
DAGMAN_GENERATE_SUBDAG_SUBMITS macro, 311
DAGMAN_HOLD_CLAIM_TIME macro, 100, 310
DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION macro, 314
DAGMAN_INSERT_SUB_FILE macro, 316
DAGMAN_LOG_ON_NFS_IS_ERROR macro, 313
DAGMAN_MAX_JOB_HOLDS macro, 310
DAGMAN_MAX_JOBS_IDLE macro, 86, 309, 310, 729, 774, 938
DAGMAN_MAX_JOBS_SUBMITTED macro, 85, 309, 774, 938
DAGMAN_MAX_POST_SCRIPTS macro, 86, 309, 775, 938
DAGMAN_MAX_PRE_SCRIPTS macro, 86, 309, 774, 938
DAGMAN_MAX_RESCUE_NUM macro, 123, 312
DAGMAN_MAX_SUBMIT_ATTEMPTS macro, 310
DAGMAN_MAX_SUBMITS_PER_INTERVAL macro, 310
DAGMAN_MUNGE_NODE_NAMES macro, 100, 311
DAGMAN_OLD_RESCUE macro, 312
DAGMAN_ON_EXIT_REMOVE macro, 316
DAGMAN_PEGASUS_REPORT_METRICS macro, 316
DAGMAN_PEGASUS_REPORT_TIMEOUT macro, 316, 780
DAGMAN_PENDING_REPORT_INTERVAL macro, 315, 552
DAGMAN_PROHIBIT_MULTI_JOBS macro, 310
DAGMAN_REMOVE_NODE_JOBS macro, 311
DAGMAN_RESET_RETRIES_UPON_RESCUE macro, 124, 312
DAGMAN_RETRY_NODE_FIRST macro, 309, 312
DAGMAN_RETRY_SUBMIT_FIRST macro, 312
DAGMAN_STARTUP_CYCLE_DETECT macro, 120, 308
DAGMAN_STORK_RM_EXE macro, 312
DAGMAN_STORK_SUBMIT_EXE macro, 311
DAGMAN_SUBMIT_DELAY macro, 310
DAGMAN_SUBMIT_DEPTH_FIRST macro, 309
DAGMAN_SUPPRESS_JOB_LOGS macro, 311
DAGMAN_SUPPRESS_NOTIFICATION macro, 311, 776, 937, 942
DAGMAN_USE_OLD_DAG_READER macro, 308
DAGMAN_USE_SHARED_PORT macro, 308
DAGMAN_USE_STRICT macro, 124, 308
DAGMAN_USER_LOG_SCAN_INTERVAL macro, 127, 128, 310
DAGMAN_VERBOSITY macro, 315, 552
DAGMAN_WRITE_PARTIAL_RESCUE macro, 125, 312
DAGManJobId
job ClassAd attribute, 987
DAGManNodesLog
job ClassAd attribute, 987
DAGManNodesMask
job ClassAd attribute, 988
DAGParentNodeNames
job ClassAd attribute, 84, 987
DC_DAEMON_LIST macro, 238

DEAD_COLLECTOR_MAX_AVOIDANCE_TIME macro, 214
Debian installation with Debian packages, 167
DEBUG_TIME_FORMAT macro, 221
dedicated scheduling, 475
DEDICATED_EXECUTE_ACCOUNT_REGEXP macro, 234, 429, 485
DEDICATED_SCHEDULER_DELAY_FACTOR macro, 273
DEDICATED_SCHEDULER_USE_FIFO macro, 272
DEDICATED_SCHEDULER_WAIT_FOR_SPOOLER macro, 272
DedicatedScheduler macro, 250, 475
DEFAULT_DOMAIN_NAME macro, 213, 438
DEFAULT_IO_BUFFER_BLOCK_SIZE macro, 284
DEFAULT_IO_BUFFER_SIZE macro, 284
DEFAULT_JOB_MAX_RETRIES macro, 284
DEFAULT_MASTER_SHUTDOWN_SCRIPT macro, 241
DEFAULT_PRIO_FACTOR macro, 291, 343
DEFAULT_RANK macro, 284
DEFAULT_RANK_STANDARD macro, 284
DEFAULT_RANK_VANILLA macro, 284
DEFAULT_UNIVERSE macro, 283, 904
deferral time
    of a job, 148
DEFRAG_CANCEL_REQUIREMENTS macro, 339
DEFRAG_DRAINING_MACHINES_PER_HOUR macro, 339
DEFRAG_INTERVAL macro, 339, 340
DEFRAG_LOG macro, 340
DEFRAG_MAX_CONCURRENT_DRAINING macro, 340
DEFRAG_MAX_WHOLE_MACHINES macro, 339
DEFRAG_NAME macro, 339, 1037
DEFRAG_RANK macro, 339
DEFRAG_REQUIREMENTS macro, 339
DEFRAG_SCHEDULE macro, 340
DEFRAG_STATE_FILE macro, 340
DEFRAG_UPDATE_INTERVAL macro, 340
DEFRAG_WHOLE_MACHINE_EXPR macro, 339
DELEGATE_FULL_JOB_GSI_CREDENTIALS macro, 318
DELEGATE_JOB_GSI_CREDENTIALS macro, 318, 916, 988
DELEGATE_JOB_GSI_CREDENTIALS_LIFETIME macro, 277, 318, 915, 988
DELEGATE_JOB_GSI_CREDENTIALS_REFRESH macro, 277, 318
DelegateJobGSICredentialsLifetime
    job ClassAd attribute, 988
DENY_ADMINISTRATOR macro, 415
DENY_ADVERTISE_MASTER macro, 415
DENY_ADVERTISE_SCHEDD macro, 415
DENY_ADVERTISE_STARTD macro, 415
DENY_CLIENT macro, 317
DENY_CLIENT macro, 415
DENY_CONFIG macro, 415
DENY_DAEMON macro, 415
DENY_NEGOTIATOR macro, 415
DENY_OWNER macro, 415
DENY_READ macro, 415
DENY_SOAP macro, 415
DENY_WRITE macro, 415
DETECTED_CORES macro, 201, 1005
DETECTED_CPUS macro, 200, 249
DETECTED_MEMORY macro, 200, 250, 1005
DETECTED_PHYSICAL_CPUS macro, 200
directed acyclic graph (DAG), 75
Directed Acyclic Graph Manager (DAGMan), 75
DISCARD_SESSION_KEYRING_ON_STARTUP macro, 244
DISCONNECTED_KEYBOARD_IDLE_BOOST macro, 253, 381
disk space requirement
    execute directory, 164
    log directory, 164
    spool directory, 164
    all versions, 166
    HTCondor files, 165
DiskProvisioned
    job ClassAd attribute, 1004
DiskUsage
    job ClassAd attribute, 988
distributed ownership
    of machines, 1, 2
Distributed Resource Management Application API (DRMAA), 602
dividing resources in multi-core machines, 377
DOCKER macro, 260, 498
docker universe, 16, 146–147
    set up, 497

DOCKER_DROP_ALL_CAPABILITIES macro, 261, 498, 709
DOCKER_IMAGE_CACHE_SIZE macro, 260, 498
DOCKER_VOLUME_DIR_XXX_MOUNT_IF macro, 704
DOCKER_VOLUME_DIR_xxx_MOUNT_IF macro, 497
DOCKER_VOLUMES macro, 260
DOT_NET_VERSIONS macro, 258
download, 161
drained state, 356, 368
DRMAA (Distributed Resource Management Application API), 602
dynamic *condor_startd* provisioning, 385
dynamic slots, 385
DYNAMIC_RUN_ACCOUNT_LOCAL_GROUP macro, 282, 647

EC2 GAHP Statistics
  NumDistinctRequests, 574
  NumExpiredSignatures, 574
  NumRequests, 574
  NumRequestsExceedingLimit, 574
EC2 grid jobs, 570
EC2_GAHP macro, 302
EC2_GAHP_RATE_LIMIT macro, 301
EC2_RESOURCE_TIMEOUT macro, 301, 573
EC2AccessKeyId
  job ClassAd attribute, 989
EC2AmiID
  job ClassAd attribute, 989
EC2BlockDeviceMapping
  job ClassAd attribute, 989
EC2ElasticIp
  job ClassAd attribute, 989
EC2IamProfileArn
  job ClassAd attribute, 989
EC2IamProfileName
  job ClassAd attribute, 989
EC2InstanceName
  job ClassAd attribute, 989
EC2InstanceType
  job ClassAd attribute, 989
EC2KeyPair
  job ClassAd attribute, 989
EC2KeyPairFile
  job ClassAd attribute, 989
EC2ParameterNames
  job ClassAd attribute, 989
EC2RemoteVirtualMachineName
  job ClassAd attribute, 990
EC2SecretAccessKey
  job ClassAd attribute, 990
EC2SecurityGroups
  job ClassAd attribute, 990
EC2SecurityIDs
  job ClassAd attribute, 990
EC2SpotPrice
  job ClassAd attribute, 989
EC2SpotRequestID
  job ClassAd attribute, 989
EC2StatusReasonCode
  job ClassAd attribute, 989
EC2TagNames
  job ClassAd attribute, 989
EC2UserData
  job ClassAd attribute, 990
EC2UserDataFile
  job ClassAd attribute, 990
ECRYPTFS_ADD_PASSPHRASE macro, 320
effective user priority (EUP), 343
email notification
  in DAGs, 937
  submit command, 902
EMAIL_DOMAIN macro, 213
EMAIL_SIGNATURE macro, 211
EmailAttributes
  job ClassAd attribute, 990
ENABLE_ADDRESS_REWRITING macro, 231
ENABLE_BACKFILL macro, 252, 479
ENABLE_CHIRP macro, 281
ENABLE_CHIRP_DELAYED macro, 281
ENABLE_CHIRP_IO macro, 281
ENABLE_CHIRP_UPDATES macro, 281
ENABLE_CLASSAD_CACHING macro, 215
ENABLE_DEPRECATION_WARNINGS macro, 285
ENABLE_GRID_MONITOR macro, 307
ENABLE_HISTORY_ROTATION macro, 212, 552
ENABLE_IPV4 macro, 217, 442
ENABLE_IPV6 macro, 217, 442
ENABLE_KERNEL_TUNING macro, 244
ENABLE_PERSISTENT_CONFIG macro, 225, 759
ENABLE_RUNTIME_CONFIG macro, 225
ENABLE_SOAP macro, 328

ENABLE_SOAP_SSL macro, 328
ENABLE_SSH_TO_JOB macro, 329
ENABLE_URL_TRANSFERS macro, 281
ENABLE_USERLOG_FSYNC macro, 220
ENABLE_USERLOG_LOCKING macro, 220
ENABLE_VERSIONED_OPSYS macro, 246
ENABLE_WEB_SERVER macro, 328
ENCRYPT_EXECUTE_DIRECTORY macro, 319, 394
ENCRYPT_EXECUTE_DIRECTORY_FILENAMES macro, 320
EncryptExecuteDirectory
job ClassAd attribute, 990
ENFORCE_CPU_AFFINITY macro, 281
EnteredCurrentStatus
job ClassAd attribute, 990
Env
job ClassAd attribute, 990
Environment
job ClassAd attribute, 990
environment variables, 42
_CONDOR_JOB_AD, 42, 488
_CONDOR_JOB_IWD, 42
_CONDOR_MACHINE_AD, 42, 488
_CONDOR_SCRATCH_DIR, 42
_CONDOR_SLOT, 42
_CONDOR_WRAPPER_ERROR_FILE, 43, 488
CONDOR_ID, 43
CONDOR_IDS, 43, 164, 210, 212
CONDOR_VM, 42
copying current environment, 901
in submit description file, 931
setting, for a job, 900
X509_USER_PROXY, 42
ENVIRONMENT_FOR_Assigned<name> macro, 255
ENVIRONMENT_VALUE_FOR_UnAssigned<name> macro, 255
Error and warning configuration syntax, 193
Event Log Reader API, 603
EVENT_LOG macro, 224, 551
EVENT_LOG_COUNT_EVENTS macro, 221
EVENT_LOG_FSYNC macro, 224, 551
EVENT_LOG_JOB_AD_INFORMATION_ATTRS macro, 58, 224, 551
EVENT_LOG_LOCKING macro, 224, 551
EVENT_LOG_MAX_ROTATIONS macro, 224, 551
EVENT_LOG_MAX_SIZE macro, 224, 551
EVENT_LOG_ROTATION_LOCK macro, 224, 551
EVENT_LOG_USE_XML macro, 224, 551
EVICT_BACKFILL macro, 253, 370, 479
EXEC_TRANSFER_ATTEMPTS macro, 278
ExecutableSize
job ClassAd attribute, 990
execute machine, 158
EXECUTE macro, 208, 209, 495, 1006
EXECUTE_LOGIN_IS_DEDICATED macro, 235
execution environment, 42
exit codes
of condor_shadow, 1041
ExitBySignal
job ClassAd attribute, 990
ExitCode
job ClassAd attribute, 990
ExitSignal
job ClassAd attribute, 991
ExitStatus
job ClassAd attribute, 991
EXPIRE_INVALIDATED_ADS macro, 290, 455

FAQ, 653
FEATURE : TESTINGMODE_POLICY_VALUES macro, 708
FEATURE : UWCS_DESKTOP_POLICY_VALUES macro, 708
FetchWorkDelay macro, 334, 539, 543
file
locking, 4, 14
memory-mapped, 4, 14
read only, 4, 14
submit description, 16
write only, 4, 14
file system
AFS, 154, 463
NFS, 154
file transfer mechanism, 32
input file specified by URL, 40, 464
input file(s) encryption, 906
output file(s) encryption, 906
output file(s) specified by URL, 40, 464, 907
submit command should_transfer_files, 907
FILE_LOCK_VIA_MUTEX macro, 220, 550
FILE_TRANSFER_DISK_LOAD_THROTTLE macro, 264, 1030

FILE_TRANSFER_DISK_LOAD_THROTTLE_LONG_HORIZON macro, 264
FILE_TRANSFER_DISK_LOAD_THROTTLE_SHORT_HORIZON macro, 264
FILE_TRANSFER_DISK_LOAD_THROTTLE_WAIT_BETWEEN_INCREMENTS macro, 264
FILESYSTEM_DOMAIN macro, 201, 235, 438
FILETRANSFER_PLUGINS macro, 281, 1007
FLOCK_COLLECTOR_HOSTS macro, 269, 270, 554
FLOCK_FROM macro, 555
FLOCK_INCREMENT macro, 270
FLOCK_NEGOTIATOR_HOSTS macro, 269, 554
FLOCK_TO macro, 554
FlockedJobs
submitter ClassAd attribute, 1035
flocking, 554
Frequently Asked Questions, 653
FS_REMOTE_DIR macro, 319, 410
FULL_HOSTNAME macro, 199

GAHP (Grid ASCII Helper Protocol), 556
GAHP macro, 301
GAHP_ARGS macro, 301
GAHP_DEBUG_HIDE_SENSITIVE_DATA macro, 301
Ganglia monitoring, 452
GANGLIA_CONFIG macro, 341
GANGLIA_GMETRIC macro, 341
GANGLIA_GSTAT_COMMAND macro, 341, 452
GANGLIA_LIB macro, 341
GANGLIA_LIB64_PATH macro, 341
GANGLIA_LIB_PATH macro, 341
GANGLIA_SEND_DATA_FOR_ALL_HOSTS macro, 341, 452
GANGLIA_VERBOSITY macro, 453
GANGLIAD_DEFAULT_CLUSTER macro, 341, 453
GANGLIAD_DEFAULT_IP macro, 342, 454
GANGLIAD_DEFAULT_MACHINE macro, 342, 454
GANGLIAD_INTERVAL macro, 340
GANGLIAD_LOG macro, 342
GANGLIAD_METRICS_CONFIG_DIR macro, 342, 452
GANGLIAD_PER_EXECUTE_NODE_METRICS macro, 341, 452
GANGLIAD_REQUIREMENTS macro, 341, 452
GANGLIAD_VERBOSITY macro, 341
GASS (Global Access to Secondary Storage), 559
GCE grid jobs, 574
GCE_GAHP macro, 303
GceAuthFile
job ClassAd attribute, 991
GceImage
job ClassAd attribute, 991
GceJsonFile
job ClassAd attribute, 991
GceMachineType
job ClassAd attribute, 991
GceMetadata
job ClassAd attribute, 991
GceMetadataFile
job ClassAd attribute, 991
GcePreemptible
job ClassAd attribute, 991
gidd_alloc command, 982
GLEXEC macro, 307
GLEXEC_HOLD_ON_INITIAL_FAILURE macro, 307
GLEXEC_JOB macro, 307
GLEXEC_RETRIES macro, 307
GLEXEC_RETRY_DELAY macro, 307
glidein, 554
GLITE_LOCATION macro, 302, 569
GlobalJobId
job ClassAd attribute, 991
GLOBUS_GATEKEEPER_TIMEOUT macro, 301
Google Compute Engine, 574
GPUs
configuration, 383
requesting GPUs for a job, 47, 905
GRACEFULLY_REMOVE_JOBS macro, 271
GRAM (Grid Resource Allocation and Management), 559
GRAM_VERSION_DETECTION macro, 301, 565
green computing, 500–502
grid computing
glidein, 554
Grid Monitor, 567
HTCondor-C, 555
matchmaking, 578
submitting jobs to BOINC, 577
submitting jobs to cream, 576
submitting jobs to GCE, 574
submitting jobs to gt2, 560
submitting jobs to gt5, 564
submitting jobs to NorduGrid, 567
submitting jobs to PBS, 569

submitting jobs to Platform LSF, 569
submitting jobs to SGE, 569
submitting jobs to Unicore, 568
submitting jobs using the EC2 Query API, 570
Grid Monitor, 567
grid type
boinc, 577
ec2, 570
authentication methods, 570
gce, 574
GRID_MONITOR macro, 307, 567
GRID_MONITOR_DISABLE_TIME macro, 307
GRID_MONITOR_HEARTBEAT_TIMEOUT macro, 307
GRID_MONITOR_NO_STATUS_TIMEOUT macro, 307
GRID_MONITOR_RETRY_DURATION macro, 307
GridJobStatus
job ClassAd attribute, 991
GRIDMANAGER_CHECKPROXY_INTERVAL macro, 299
GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT macro, 301
GRIDMANAGER_CONTACT_SCHEDD_DELAY macro, 300
GRIDMANAGER_EMPTY_RESOURCE_DELAY macro, 300
GRIDMANAGER_GAHP_CALL_TIMEOUT macro, 301
GRIDMANAGER_GAHP_RESPONSE_TIMEOUT macro, 301
GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT macro, 301
GRIDMANAGER_JOB_PROBE_INTERVAL macro, 300
GRIDMANAGER_JOB_PROBE_RATE macro, 300
GRIDMANAGER_LOG macro, 299
GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE macro, 301, 564
GRIDMANAGER_MAX_PENDING_REQUESTS macro, 301
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE macro, 300
GRIDMANAGER_MINIMUM_PROXY_TIME macro, 299
GRIDMANAGER_PROXY_REFRESH_TIME macro, 299
GRIDMANAGER_RESOURCE_PROBE_DELAY macro, 300
GRIDMANAGER_RESOURCE_PROBE_INTERVAL macro, 300, 573
GRIDMANAGER_SELECTION_EXPR macro, 273
GRIDMAP macro, 319, 404, 411
GridResource
job ClassAd attribute, 991
group accounting
<none> group, 348
GROUP_ACCEPT_SURPLUS macro, 297, 350
GROUP_ACCEPT_SURPLUS_<groupname> macro, 297, 350
GROUP_AUTOREGROUP macro, 296, 1001, 1017
GROUP_AUTOREGROUP_<groupname> macro, 297
GROUP_DYNAMIC_MACH_CONSTRAINT macro, 293
GROUP_NAMES macro, 296
GROUP_PRIO_FACTOR_<groupname> macro, 296
GROUP_QUOTA_<groupname> macro, 296
GROUP_QUOTA_DYNAMIC_<groupname> macro, 296
GROUP_QUOTA_MAX_ALLOCATION_ROUNDS macro, 297, 1034
GROUP_QUOTA_ROUND_ROBIN_RATE macro, 297
GROUP_SORT_EXPR macro, 297, 351
groups
accounting, 347
quotas, 348
GSI (Grid Security Infrastructure), 559
GSI_AUTHZ_CONF macro, 318
GSI_DAEMON_CERT macro, 317, 404
GSI_DAEMON_DIRECTORY macro, 317, 403, 404
GSI_DAEMON_KEY macro, 317, 404
GSI_DAEMON_NAME macro, 317
GSI_DAEMON_PROXY macro, 318, 404
GSI_DAEMON_TRUSTED_CA_DIR macro, 318, 404, 574
GSI_DELEGATION_CLOCK_SKEW_ALLOWABLE macro, 319
GSI_DELEGATION_KEYBITS macro, 318
GSI_SKIP_HOST_CHECK macro, 317
GSI_SKIP_HOST_CHECK_CERT_REGEX macro, 317
GSS_ASSIST_GRIDMAP_CACHE_EXPIRATION macro, 318
GT2_GAHP macro, 302

HA_<SUBSYS>_LOCK_HOLD_TIME macro, 325
HA_<SUBSYS>_LOCK_URL macro, 325
HA_<SUBSYS>_POLL_PERIOD macro, 325
HA_LOCK_HOLD_TIME macro, 325
HA_LOCK_URL macro, 325

HA_POLL_PERIOD macro, 325
HAD macro, 326
HAD_ARGS macro, 326
HAD_CONNECTION_TIMEOUT macro, 326
HAD_CONTROLLEE macro, 326
HAD_DEBUG macro, 327
HAD_LIST macro, 326
HAD_LOG macro, 327
HAD_UPDATE_INTERVAL macro, 327
HAD_USE_PRIMARY macro, 326
HAD_USE_REPLICATION macro, 327, 459
Hadoop Distributed File System (HDFS)
integrated with HTCondor, 654
Hawkeye
see Daemon ClassAd Hooks, 547
HDFS_ALLOW macro, 656
HDFS_BACKUPNODE macro, 656
HDFS_BACKUPNODE_WEB macro, 656
HDFS_DATANODE_ADDRESS macro, 655
HDFS_DATANODE_CLASS macro, 656
HDFS_DATANODE_DIR macro, 655
HDFS_DATANODE_WEB macro, 655
HDFS_DENY macro, 656
HDFS_HOME macro, 655
HDFS_LOG4J macro, 656
HDFS_NAMENODE macro, 655
HDFS_NAMENODE_CLASS macro, 656
HDFS_NAMENODE_DIR macro, 655
HDFS_NAMENODE_ROLE macro, 656
HDFS_NAMENODE_WEB macro, 655
HDFS_NODETYPE macro, 656
HDFS_REPLICATION macro, 656
HDFS_SITE_FILE macro, 656
HeldJobs
submitter ClassAd attribute, 1035
heterogeneous pool
submitting a job to, 43
HIBERNATE macro, 258, 500
HIBERNATE_CHECK_INTERVAL macro, 258, 500
HIBERNATION_OVERRIDE_WOL macro, 259
HIBERNATION_PLUGIN macro, 259
HIBERNATION_PLUGIN_ARGS macro, 259
hierarchical group quotas, 348
High Availability, 455
of central manager, 457
of job queue, 455
of job queue, with remote job submission, 457
sample configuration, 460
High-Performance Computing (HPC), 1
High-Throughput Computing (HTC), 1
HIGHPORT macro, 231, 433
HISTORY macro, 212
HISTORY_HELPER macro, 709
HISTORY_HELPER_MAX_CONCURRENCY macro, 212
HISTORY_HELPER_MAX_HISTORY macro, 212
HOLD_JOB_IF_CREDENTIAL_EXPIRES macro, 300
HoldKillSig
job ClassAd attribute, 991
HoldReason
job ClassAd attribute, 991
HoldReasonCode
job ClassAd attribute, 991
HoldReasonSubCode
job ClassAd attribute, 993
Hooks, 538–549
Daemon ClassAd Hooks, 547
job hooks that fetch work, 538
Job Router hooks, 545
host certificate, 403
HOST_ALIAS macro, 317
HOSTALLOW macro, 225
HOSTALLOW_ADMINISTRATOR macro, 178
HOSTALLOW_NEGOTIATOR_SCHEDD macro, 554
HOSTALLOW_READ macro, 178
HOSTALLOW_WRITE macro, 178, 181
HOSTALLOW... macro, 225, 759
HOSTDENY macro, 225
HOSTNAME macro, 199
HPC (High-Performance Computing), 1
HTC (High-Throughput Computing), 1
HTCondor
acknowledgments, 6
configuration-intro, 184
configuration-macros, 206
configuration-templates, 201
contact information, 7
contributions, 5
default policy, 370
FAQ, 653
flocking, 554
Frequently Asked Questions, 653
limitations, under UNIX, 4

mailing lists, 7
overview, 1–3
platforms available, 5
pool, 157
resource allocation, 10
resource management, 2
shared functionality in daemons, 448
universe, 13
Unix administrator, 163
user manual, 9–156
HTCondor commands
condor_advertise, 742
condor_check_userlogs, 746
condor_checkpoint, 747
condor_chirp, 750
condor_cod, 754
condor_compile, 61, 757
list of supported compilers, 5
condor_config_val, 759
condor_configure, 764, 801
condor_continue, 769
condor_dagman, 773
condor_dagman_metrics_reporter, 779
condor_drain, 782
condor_fetchlog, 784
condor_findhost, 787
condor_gather_info, 789
condor_gpu_discovery, 792
condor_history, 795
condor_hold, 52, 798
condor_install, 764, 801
condor_job_router_info, 806
condor_master, 808
condor_off, 809
condor_on, 812
condor_ping, 815
condor_pool_job_report, 818
condor_power, 819
condor_preen, 821
condor_prio, 52, 60, 823
condor_procd, 825
condor_q, 12, 49, 53, 828
condor_qedit, 843
condor_qsub, 845
condor_reconfig, 850
condor_release, 52, 853
condor_reschedule, 855
condor_restart, 857
condor_rm, 13, 51, 860
condor_rmdir, 863
condor_router_rm, 869
condor_run, 871
condor_set_shutdown, 874
condor_sos, 880
condor_ssh_to_job, 876
condor_stats, 882
condor_status, 10, 12, 29, 49, 50, 885
condor_store_cred, 893
condor_submit, 12, 16, 895
condor_submit_dag, 937
condor_suspend, 944
condor_tail, 946
condor_transfer_data, 948
condor_transform_ads, 950
condor_update_machine_ad, 953
condor_updates_stats, 955
condor_urlfetch, 958
condor_userlog, 960
condor_userprio, 60, 963
condor_vacate, 968
condor_vacate_job, 970
condor_version, 973
condor_wait, 975
condor_who, 978
gidd_alloc, 982
procd_ctl, 983
HTCondor daemon
command line arguments, 450
condor_ckpt_server, 160, 444
condor_collector, 159
condor_credd, 160, 299, 643
condor_defrag, 160, 390
condor_gridmanager, 160
condor_had, 160, 458
condor_hdfs, 160
condor_job_router, 160, 583
condor_kbdd, 159, 470
condor_lease_manager, 160
condor_master, 158, 808
condor_negotiator, 159
condor_procd, 160
condor_replication, 160, 458

condor_rooster, 160, 501
condor_schedd, 159
condor_shadow, 14, 155
condor_shadow, 159
condor_shared_port, 160, 435
condor_startd, 158, 352
condor_starter, 159
condor_transferer, 160, 458
descriptions, 158
HTCondor daemon, source code contrib
condor_dbmsd, 656
condor_quill, 656
HTCondor GAHP, 556
HTCondor-C, 555–558
configuration, 556
job submission, 557
HTCondor-G, 558–567
GASS, 559
GRAM, 559
GSI, 559
job submission, 560
limitations, 567
proxy, 560
X.509 certificate, 560
HTCondorView
Client, 683
Client installation, 683
configuration, 472
Server, 471
use of *crontab* program, 685

IdleJobs
submitter ClassAd attribute, 1035
IF/ELSE configuration syntax, 194
IF/ELSE submit commands syntax, 23
IGNORE_DNS_PROTOCOL_PREFERENCE macro, 217
IGNORE_LEAF_OOM macro, 218
IGNORE_NFS_LOCK_ERRORS macro, 236
IGNORE_TARGET_PROTOCOL_PREFERENCE macro, 217
ImageSize
job ClassAd attribute, 993
IMMUTABLE_JOB_ATTRS macro, 276
IN_HIGHPORT macro, 232, 433
IN_LOWPORT macro, 232, 433
include command macro, 192
INCLUDE configuration syntax, 192
INCLUDE macro, 208
installation
checkpoint server, 444
download, 161
for the docker universe, 497
for the vm universe, 493
HTCondorView Client, 683
installing a new version on an existing pool, 181
Java, 492
running as root, 163
Singularity, 499
using Debian packages, 167
using Red Hat RPMs, 166
Windows, 172–180
with *condor_configure*, 168
interactive jobs, 48
INTERACTIVE_SUBMIT_FILE macro, 48, 285
INVALID_LOG_FILES macro, 286, 821
IP_ADDRESS macro, 199
IP_ADDRESS_IS_V6 macro, 199
IPv4 port specification, 431
IPV4_ADDRESS macro, 199
IPv6, 442–444
IPV6_ADDRESS macro, 199, 714
IS_OWNER macro, 246, 360
IS_VALID_CHECKPOINT_PLATFORM macro, 246
IS_VALID_CHECKPOINT_PLATFORM macro, 353
IwdFlushNFSCache
job ClassAd attribute, 994

Java, 13, 61, 492
job example, 62
multiple class files, 63
using JAR files, 64
using packages, 65
JAVA macro, 257, 492
Java Virtual Machine, 13, 61, 492
JAVA5_HOOK_PREPARE_JOB macro, 544
JAVA_CLASSPATH_ARGUMENT macro, 257
JAVA_CLASSPATH_DEFAULT macro, 257
JAVA_CLASSPATH_SEPARATOR macro, 257
JAVA_EXTRA_ARGUMENTS macro, 257, 493
job
analysis, 53
batch ready, 12

completion, 59
dependencies within, 75
event log file, 55
heterogeneous submit, 43
interactive, 48
job ID
defined for a DAGMan node job, 81
multiple data sets, 2
not running, 53
not running, on hold, 55
preparation, 12
priority, 52, 60
state, 50, 52, 995
submission using a shared file system, 31
submission without a shared file system, 32
submitting, 16
universe, 995
who the job runs as, 428
job deferral time, 148
job execution
at a specific time, 147
Job hooks, 538
Fetch Hooks
Job exit, 541
Update job info, 541
Evict a claim, 540
Fetch work, 539
Prepare job, 540
Reply to fetched work, 539
FetchWorkDelay, 543
Hooks invoked by HTCondor, 539
Java example, 543
Job Router Hooks
Job Cleanup, 546
Job Finalize, 546
Translate Job, 545
Update Job Info, 546
keywords, 542
job ID
cluster identifier, 929, 986
defined for a DAGMan node job, 81
process identifier, 999
use in *condor_wait*, 976
job lease, 156
Job Log Reader API, 603
Job monitor, 685
Job Router, 333, 545, 583
Job Router commands
condor_router_history, 865
condor_router_q, 867
Job Router Routing Table ClassAd attribute
Copy_<ATTR>, 588
Delete_<ATTR>, 588
EditJobInPlace, 588
Eval_Set_<ATTR>, 588
FailureRateThreshold, 587
GridResource, 587
JobFailureTest, 587
JobShouldBeSandboxed, 588
MaxIdleJobs, 587
MaxJobs, 587
Name, 587
OverrideRoutingEntry, 588
Requirements, 587
Set_<ATTR>, 588
SharedX509UserProxy, 588
TargetUniverse, 588
UseSharedX509UserProxy, 588
job scheduling
periodic, 150
job transforms, 391
JOB_DEFAULT_NOTIFICATION macro, 283
JOB_DEFAULT_REQUESTCPUS macro, 283, 388
JOB_DEFAULT_REQUESTDISK macro, 283, 388
JOB_DEFAULT_REQUESTMEMORY macro, 283, 388, 905, 1000
JOB_EXECDIR_PERMISSIONS macro, 282
JOB_INHERITS_STARTER_ENVIRONMENT macro, 280
JOB_IS_FINISHED_COUNT macro, 267
JOB_IS_FINISHED_INTERVAL macro, 267
job_max_vacate_time, 926
JOB_QUEUE_LOG macro, 220, 550
JOB_RENICE_INCREMENT macro, 278, 353
JOB_ROUTER_DEFAULTS macro, 303
JOB_ROUTER_ENTRIES macro, 303, 589
JOB_ROUTER_ENTRIES_CMD macro, 303, 589
JOB_ROUTER_ENTRIES_FILE macro, 303
JOB_ROUTER_ENTRIES_REFRESH macro, 304
JOB_ROUTER_HOOK_KEYWORD macro, 335
JOB_ROUTER_LOCK macro, 304
JOB_ROUTER_MAX_JOBS macro, 304

JOB_ROUTER_NAME macro, 304
JOB_ROUTER_POLLING_PERIOD macro, 304, 546
JOB_ROUTER_RELEASE_ON_HOLD macro, 304
JOB_ROUTER_SCHEDD1_NAME macro, 305
JOB_ROUTER_SCHEDD1_POOL macro, 305
JOB_ROUTER_SCHEDD1_SPOOL macro, 305
JOB_ROUTER_SCHEDD2_NAME macro, 305
JOB_ROUTER_SCHEDD2_POOL macro, 305
JOB_ROUTER_SCHEDD2_SPOOL macro, 305
JOB_ROUTER_SOURCE_JOB_CONSTRAINT macro, 304
JOB_SPOOL_PERMISSIONS macro, 276
JOB_START_COUNT macro, 266, 1025
JOB_START_DELAY macro, 266, 1025
JOB_STOP_COUNT macro, 266
JOB_STOP_DELAY macro, 266
JOB_TRANSFORM_<Name> macro, 275
JOB_TRANSFORM_<name> macro, 391
JOB_TRANSFORM_NAMES macro, 275, 391
JobAdInformationAttrs
job ClassAd attribute, 994
JobCurrentStartDate
job ClassAd attribute, 994
JobCurrentStartExecutingDate
job ClassAd attribute, 994
JobCurrentStartTransferOutputDate
job ClassAd attribute, 994
JobDescription
job ClassAd attribute, 994
JobLeaseDuration
job ClassAd attribute, 156, 994
JobMaxVacateTime
job ClassAd attribute, 994
JobNotification
job ClassAd attribute, 994
JobPrio
job ClassAd attribute, 994
JobRunCount
job ClassAd attribute, 994
JobStartDate
job ClassAd attribute, 995
JobStatus
job ClassAd attribute, 995
JobUniverse
job ClassAd attribute, 995
JVM, 13, 61, 492
KBDD_BUMP_CHECK_AFTER_IDLE_TIME macro, 249
KBDD_BUMP_CHECK_SIZE macro, 249
KEEP_POOL_HISTORY macro, 288, 472
KeepClaimIdle
job ClassAd attribute, 995
Kerberos authentication, 406
KERBEROS_CLIENT_KEYTAB macro, 322
KERBEROS_MAP_FILE macro, 407, 412
KERBEROS_SERVER_KEYTAB macro, 321
KERBEROS_SERVER_PRINCIPAL macro, 321, 407
KERBEROS_SERVER_SERVICE macro, 322
KERBEROS_SERVER_USER macro, 321
KERNEL_TUNING_LOG macro, 244
KILL macro, 245–247, 369
KILLING_TIMEOUT macro, 247, 367, 369, 925, 996
KillSig
job ClassAd attribute, 995
KillSigTimeout
job ClassAd attribute, 996

LastCheckpointPlatform
job ClassAd attribute, 996
LastCkptServer
job ClassAd attribute, 996
LastCkptTime
job ClassAd attribute, 996
LastMatchTime
job ClassAd attribute, 996
LastRejMatchReason
job ClassAd attribute, 996
LastRejMatchTime
job ClassAd attribute, 996
LastRemotePool
job ClassAd attribute, 996
LastSuspensionTime
job ClassAd attribute, 996
LastVacateTime
job ClassAd attribute, 996
LeaseManager.CLASSAD_LOG macro, 306
LeaseManager.DEBUG_ADS macro, 306
LeaseManager.DEFAULT_MAX_LEASE_DURATION macro, 306
LeaseManager.GETADS_INTERVAL macro, 306
LeaseManager.MAX_LEASE_DURATION macro, 306

LeaseManager.MAX_TOTAL_LEASE_DURATION macro, 306
LeaseManager.PRUNE_INTERVAL macro, 306
LeaseManager.QUERY_ADTYPE macro, 306
LeaseManager.QUERY_CONSTRAINTS macro, 306
LeaseManager.UPDATE_INTERVAL macro, 306
LeaveJobInQueue
job ClassAd attribute, 996
LIB macro, 208
LIBEXEC macro, 208
LIBVIRT_XML_SCRIPT macro, 323
LIBVIRT_XML_SCRIPT_ARGS macro, 323
limits
on resource usage, 487
on resource usage with cgroup, 488
linking
dynamic, 4, 15
static, 4, 15
Linux kernel
per job PID namespaces, 484
LINUX_HIBERNATION_METHOD macro, 259
LINUX_KERNEL_TUNING_SCRIPT macro, 244
local universe, 16
LOCAL_CONFIG_DIR macro, 186, 210
LOCAL_CONFIG_DIR_EXCLUDE_REGEXP macro, 210
LOCAL_CONFIG_FILE macro, 186, 191, 209, 467, 468
LOCAL_CREDD macro, 644
LOCAL_DIR macro, 164, 166, 208
LOCAL_UNIV_EXECUTE macro, 261
LocalJobsIdle
submitter ClassAd attribute, 1035
LocalJobsRunning
submitter ClassAd attribute, 1035
LocalSysCpu
job ClassAd attribute, 996
LocalUserCpu
job ClassAd attribute, 996
LOCK macro, 164, 211
LOCK_DEBUG_LOG_TO_APPEND macro, 220
LOCK_FILE_UPDATE_INTERVAL macro, 228
log files
event codes for jobs, 1041
job event codes and descriptions, 55
LOG macro, 208, 213, 249, 451
LOG_ON_NFS_IS_ERROR macro, 285
logging, 549–552
LOGS_USE_TIMESTAMP macro, 221, 550
LOWPORT macro, 231, 433
LSF, 569
LSF_GAHP macro, 302

machine
central manager, 157
checkpoint server, 158
execute, 158
owner, 157
submit, 158
machine activity, 359
Backfill, 359
Benchmarking, 359
Busy, 359
Drained, 360
Idle, 359
Killing, 359
Retiring, 359
Suspended, 359
transitions, 360–370
transitions summary, 368
Unclaimed, 359
Vacating, 359
machine ClassAd, 11
machine state, 355
Backfill, 356, 367
Claimed, 355, 364
claimed, the claim lease, 358
Drained, 356, 368
Matched, 355, 363
Owner, 355, 360
Preempting, 355, 366
transitions, 360–370
transitions summary, 368
Unclaimed, 355, 362
machine state and activities figure, 360
MACHINE_RESOURCE_<name> macro, 254, 255, 379
MACHINE_RESOURCE_INVENTORY_<name> macro, 255
MACHINE_RESOURCE_INVENTORY_GPUs macro, 1019
MACHINE_RESOURCE_NAMES macro, 254, 380
MachineAttr<X><N>
job ClassAd attribute, 996

MachineMaxVacateTime macro, 245–247, 367, 369
macro
    in configuration file, 186
    in submit description file, 929
    subsystem names, 199
MAIL macro, 211, 468
MAIL_FROM macro, 211
mailing lists, 7
MASTER_<name>_BACKOFF_CEILING macro, 241
MASTER_<name>_BACKOFF_CONSTANT macro, 241
MASTER_<name>_BACKOFF_FACTOR macro, 241
MASTER_<name>_RECOVER_FACTOR macro, 241
MASTER_<SUBSYS>_CONTROLLER macro, 326
MASTER_ADDRESS_FILE macro, 243
MASTER_ATTRS macro, 243
MASTER_BACKOFF_CEILING macro, 241
MASTER_BACKOFF_CONSTANT macro, 241
MASTER_BACKOFF_FACTOR macro, 241
MASTER_CHECK_INTERVAL macro, 287
MASTER_CHECK_NEW_EXEC_INTERVAL macro, 181, 240
MASTER_DEBUG macro, 243
MASTER_HA_LIST macro, 324, 456
MASTER_HAD_BACKOFF_CONSTANT macro, 459
MASTER_INSTANCE_LOCK macro, 243
MASTER_NAME macro, 207, 242, 808
MASTER_NEW_BINARY_DELAY macro, 240
MASTER_NEW_BINARY_RESTART macro, 240
MASTER_RECOVER_FACTOR macro, 241
MASTER_SHUTDOWN_<Name> macro, 241
MASTER_UPDATE_INTERVAL macro, 240
MATCH_TIMEOUT macro, 357, 363, 368
matched state, 355, 363
matchmaking, 2
    negotiation algorithm, 345
    on the Grid, 578
    priority, 343
MAX_<SUBSYS>_<LEVEL>_LOG macro, 224
MAX_<SUBSYS>_LOG macro, 218, 219, 550
MAX_ACCEPTS_PER_CYCLE macro, 228
MAX_ACCOUNTANT_DATABASE_SIZE macro, 291
MAX_C_GAHP_LOG macro, 302
MAX_CKPT_SERVER_LOG macro, 446
MAX_CLAIM_ALIVES_MISSED macro, 248, 267
MAX_CONCURRENT_DOWNLOADS macro, 263, 264, 1002
MAX_CONCURRENT_UPLOADS macro, 264, 1002
MAX_DAGMAN_LOG macro, 88, 315
MAX_DEFAULT_LOG macro, 218, 219
MAX_DISCARDED_RUN_TIME macro, 237, 444
MAX_EVENT_LOG macro, 224
MAX_FILE_DESCRIPTORS macro, 230, 440
MAX_HAD_LOG macro, 326
MAX_HISTORY_LOG macro, 212, 552
MAX_HISTORY_ROTATIONS macro, 212, 552
MAX_JOB_MIRROR_UPDATE_LAG macro, 304
MAX_JOB_QUEUE_LOG_ROTATIONS macro, 212, 550
max_job_retirement_time, 926
MAX_JOBS_PER_OWNER macro, 263
MAX_JOBS_PER_SUBMISSION macro, 263
MAX_JOBS_RUNNING macro, 50, 262, 434, 1024
MAX_JOBS_SUBMITTED macro, 263
MAX_NEXT_JOB_START_DELAY macro, 266, 911, 997
MAX_NUM_<SUBSYS>_LOG macro, 219, 550
MAX_NUM_CPUS macro, 250
MAX_NUM_SCHEDD_AUDIT_LOG macro, 275, 551
MAX_NUM_SHADOW_LOG macro, 728
MAX_NUM_SHARED_PORT_AUDIT_LOG macro, 333, 551
MAX_PENDING_STARTD_CONTACTS macro, 263
MAX_PERIODIC_EXPR_INTERVAL macro, 270
MAX_PROCD_LOG macro, 298
MAX_REAPS_PER_CYCLE macro, 228
MAX_REPLICATION_LOG macro, 327
MAX_RUNNING_SCHEDULER_JOBS_PER_OWNER macro, 263, 706, 710
MAX_SCHEDD_AUDIT_LOG macro, 275, 551
MAX_SHADOW_EXCEPTIONS macro, 263
MAX_SHADOW_STATS_LOG macro, 277
MAX_SHARED_PORT_AUDIT_LOG macro, 333, 551
MAX_SLOT_TYPES macro, 254
MAX_STARTER_STATS_LOG macro, 283
MAX_TIME_SKIP macro, 227
MAX_TRACKING_GID macro, 299, 485
MAX_TRANSFER_INPUT_MB macro, 264, 907, 992, 997
MAX_TRANSFER_LIFETIME macro, 327
MAX_TRANSFER_OUTPUT_MB macro, 265, 907, 993, 997
MAX_TRANSFER_QUEUE_AGE macro, 265
MAX_TRANSFERER_LOG macro, 328
MAX_VM_GAHP_LOG macro, 322
MaxHosts

job ClassAd attribute, 996
MaxJobRetirementTime
job ClassAd attribute, 996
MAXJOBRETIREMENTTIME macro, 244, 248, 295, 369
MaxTransferInputMB
job ClassAd attribute, 997
MaxTransferOutputMB
job ClassAd attribute, 997
MEMORY macro, 250
MEMORY_USAGE_METRIC macro, 282
MEMORY_USAGE_METRIC_VM macro, 282
MemoryProvisioned
job ClassAd attribute, 1004
MemoryUsage
job ClassAd attribute, 997
migration, 2, 3
MIN_TRACKING_GID macro, 298, 485
MinHosts
job ClassAd attribute, 997
MODIFY_REQUEST_EXPR_REQUESTCPUS macro, 256, 388
MODIFY_REQUEST_EXPR_REQUESTDISK macro, 256, 388
MODIFY_REQUEST_EXPR_REQUESTMEMORY macro, 256, 388
Monitoring
with Ganglia, 452
monitoring pools, 452
MOUNT_UNDER_SCRATCH macro, 252
MPI application, 68, 72
under the dedicated scheduler, 475
multi-core machines
configuration, 377–391
multiple network interfaces, 436
MUST_MODIFY_REQUEST_EXPRS macro, 256
MY., ClassAd scope resolution prefix, 515
MyAddress
submitter ClassAd attribute, 1035
MYPROXY_GET_DELEGATION macro, 328, 566

Name
submitter ClassAd attribute, 1035
NAMED_CHROOT macro, 280
namespaces
per job PID namespaces, 484
NEGOTIATE_ALL_JOBS_IN_CLUSTER macro, 270, 346
negotiation, 345
by group, 348
priority, 343
NEGOTIATION_CYCLE_STATS_LENGTH macro, 291
NEGOTIATOR_ADDRESS_FILE macro, 432
NEGOTIATOR_ADDRESS_FILE macro, 225
NEGOTIATOR_ALLOW_QUOTA_OVERSUBSCRIPTION macro, 298, 349, 351
NEGOTIATOR_CONSIDER_EARLY_PREEMPTION macro, 248, 267, 295, 369
NEGOTIATOR_CONSIDER_PREEMPTION macro, 295
NEGOTIATOR_CROSS_SLOT_PRIOS macro, 708
NEGOTIATOR_CYCLE_DELAY macro, 291
NEGOTIATOR_DEBUG macro, 293
NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES macro, 291
NEGOTIATOR_HOST macro, 207
NEGOTIATOR_IGNORE_USER_PRIORITIES macro, 581
NEGOTIATOR_INFORM_STARTD macro, 292
NEGOTIATOR_INTERVAL macro, 290
NEGOTIATOR_MATCH_EXPRS macro, 294
NEGOTIATOR_MATCH_LOG macro, 224, 552
NEGOTIATOR_MATCHLIST_CACHING macro, 295, 581
NEGOTIATOR_MAX_TIME_PER_CYCLE macro, 294
NEGOTIATOR_MAX_TIME_PER_PIESPIN macro, 294
NEGOTIATOR_MAX_TIME_PER_SCHEDD macro, 294
NEGOTIATOR_MAX_TIME_PER_SUBMITTER macro, 293, 1034
NEGOTIATOR_POST_JOB_RANK macro, 292
NEGOTIATOR_PRE_JOB_RANK macro, 292
NEGOTIATOR_READ_CONFIG_BEFORE_CYCLE macro, 295
NEGOTIATOR_RESOURCE_REQUEST_LIST_SIZE macro, 294
NEGOTIATOR_SLOT_CONSTRAINT macro, 293
NEGOTIATOR_SLOT_POOLSIZE_CONSTRAINT macro, 293, 1032
NEGOTIATOR_SOCKET_CACHE_SIZE macro, 291, 434
NEGOTIATOR_TIMEOUT macro, 291

NEGOTIATOR_TRIM_SHUTDOWN_THRESHOLD macro, 293
NEGOTIATOR_UPDATE_AFTER_CYCLE macro, 295
NEGOTIATOR_UPDATE_INTERVAL macro, 291
NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT macro, 233
NEGOTIATOR_USE_SLOT_WEIGHTS macro, 297
NEGOTIATOR_USE_WEIGHTED_DEMAND macro, 297
network, 4, 14, 430
network interfaces
multiple, 436
NETWORK_HOSTNAME macro, 230
NETWORK_INTERFACE macro, 230, 437, 439, 443
NETWORK_MAX_PENDING_CONNECTS macro, 214
NextJobStartDelay
job ClassAd attribute, 997
NFS
cache flush on submit machine, 155
interaction with, 154
nice job, 61
NICE_USER_PRIO_FACTOR macro, 291, 343
NiceUser
job ClassAd attribute, 997
NICs, 436
NO_DNS macro, 213, 444, 722
NONBLOCKING_COLLECTOR_UPDATE macro, 232
Nonessential
job ClassAd attribute, 997
NorduGrid, 567
NORDUGRID_GAHP macro, 302
NOT_RESPONDING_TIMEOUT macro, 227
NOT_RESPONDING_WANT_CORE macro, 227
notification
e-mail in DAGs, 937
e-mail related to a job, 902
NTDomain
job ClassAd attribute, 997
NUM_CLAIMS macro, 257
NUM_CPUS macro, 249, 256, 378
NUM_SLOTS macro, 256, 377
NUM_SLOTS_TYPE_<N> macro, 256
NumCkpts
job ClassAd attribute, 997
NumDistinctRequests
EC2 GAHP Statistics, 574
NumExpiredSignatures
EC2 GAHP Statistics, 574
NumGlobusSubmits
job ClassAd attribute, 997
NumJobCompletions
job ClassAd attribute, 997
NumJobMatches
job ClassAd attribute, 997
NumJobReconnects
job ClassAd attribute, 998
NumJobStarts
job ClassAd attribute, 998
NumPids
job ClassAd attribute, 998
NumRequests
EC2 GAHP Statistics, 574
NumRequestsExceedingLimit
EC2 GAHP Statistics, 574
NumRestarts
job ClassAd attribute, 998
NumShadowExceptions
job ClassAd attribute, 998
NumShadowStarts
job ClassAd attribute, 998
NumSystemHolds
job ClassAd attribute, 998

OBITUARY_LOG_LENGTH macro, 240
offline ClassAd, 1018
offline machine, 500
OFFLINE_EXPIRE_ADS_AFTER macro, 260, 501
OFFLINE_LOG macro, 260, 501
OFFLINE_MACHINE_RESOURCE_<name> macro, 255
OPEN_VERB_FOR_<EXT>_FILES macro, 214
OPENMPI_EXCLUDE_NETWORK_INTERFACES macro, 74, 261
OPENMPI_INSTALL_PATH macro, 73, 261
OPSYS macro, 200
OPSYS_AND_VER macro, 200
OPSYS_VER macro, 200
OtherJobRemoveRequirements
job ClassAd attribute, 998
OUT_HIGHPORT macro, 232, 433
OUT_LOWPORT macro, 232, 433
OutputDestination
job ClassAd attribute, 998

overview, 1–3
Owner
job ClassAd attribute, 998
owner
of directories, 164
owner state, 355, 360

parallel scheduling groups, 477
parallel universe, 16, 68–74
running MPI applications, 72
ParallelSchedulingGroup macro, 272, 478
ParallelShutdownPolicy
job ClassAd attribute, 998
partitionable slot preemption, 387
partitionable slots, 385
negotiator-side resource consumption policy, 388
PASSWD_CACHE_REFRESH macro, 214
PBS (Portable Batch System), 569
PBS_GAHP macro, 302
PER_JOB_HISTORY_DIR macro, 272, 726
PER_JOB_NAMESPACES macro, 282
PERIODIC_CHECKPOINT macro, 245, 523
PERIODIC_EXPR_INTERVAL macro, 270
PERIODIC_EXPR_TIMESLICE macro, 270
PERIODIC_MEMORY_SYNC macro, 277
Perl module, 613
examples, 616
PERSISTENT_CONFIG_DIR macro, 225
PID macro, 201
PID namespaces
per job, 484
pie slice, 346
pie spin, 346
PIPE_BUFFER_MAX macro, 228
platform-specific information
address space randomization, 641
Linux, 640
Macintosh OS X, 652
Windows, 641–652
starting and stopping a job, 647
platforms supported, 5
policy
at UW-Madison, 372
default with HTCondor, 370
desktop/non-desktop, 373
disabling preemption, 375
enabling preemption, 375
suspending jobs instead of evicting them, 375
test job, 372
time of day, 372
utilizing interactive jobs, 376
POLLING_INTERVAL macro, 247, 364
pool management
absent ClassAds, 454
installing a new version on an existing pool, 181
monitoring, 452
reconfiguration, 183
restarting HTCondor, 182
shutting down HTCondor, 182
pool monitoring, 452
pool of machines, 157
POOL_HISTORY_DIR macro, 288, 472
POOL_HISTORY_MAX_STORAGE macro, 288, 472
POOL_HISTORY_SAMPLING_INTERVAL macro, 288
port usage, 431
conflicts, 434
firewalls, 433
IPv4 port specification, 431
multiple collectors, 434
nonstandard ports for central managers, 432
PostJobPrio1
job ClassAd attribute, 999
PostJobPrio2
job ClassAd attribute, 999
power management, 500–502
entering a low power state, 500
leaving a low power state, 501
Linux platform details, 501
Windows platform troubleshooting, 502
PPID macro, 201
PREEMPT macro, 244, 369, 541
preempting state, 355, 366
preemption
desktop/non-desktop, 373
disabling and enabling, 375
priority, 60, 343
vacate, 61
PREEMPTION_RANK macro, 293
PREEMPTION_RANK_STABLE macro, 293, 344
PREEMPTION_REQUIREMENTS macro, 60, 292, 295, 343, 832

PREEMPTION_REQUIREMENTS_STABLE macro, 292, 344
PREEN macro, 239
PREEN_ADMIN macro, 286, 821
PREEN_ARGS macro, 240
PREEN_INTERVAL macro, 240
PREFER_IPV4 macro, 217, 443
PREFER_OUTBOUND_IPV4 macro, 217
PreJobPrio1
    job ClassAd attribute, 999
PreJobPrio2
    job ClassAd attribute, 999
PreserveRelativeExecutable
    job ClassAd attribute, 999
priority
    by group, 347
    in machine allocation, 342
    nice job, 61
    of a job, 52, 60
    of a user, 60
PRIORITY_HALFLIFE macro, 60, 291, 342, 345
PRIVATE_NETWORK_INTERFACE macro, 231, 437
PRIVATE_NETWORK_NAME macro, 228, 230, 437
PROCD_ADDRESS macro, 298
procd_ctl command, 983
PROCD_LOG macro, 298
PROCD_MAX_SNAPSHOT_INTERVAL macro, 298
process
    definition for a submitted job, 999
ProcId
    job ClassAd attribute, 999
PROPORTIONAL_SWAP_ASSSIGNMENT macro, 489
ProportionalSetSizeKb
    job ClassAd attribute, 999
PROTECTED_JOB_ATTRS macro, 276
proxy, 560
    renewal with *MyProxy*, 565
pslot preemption, 387
PUBLISH_OBITUARIES macro, 240
Python bindings, 621

Q_QUERY_TIMEOUT macro, 213
QDate
    job ClassAd attribute, 999
QUERY_TIMEOUT macro, 287
QUEUE_ALL_USERS_TRUSTED macro, 268
QUEUE_CLEAN_INTERVAL macro, 268, 550
QUEUE_SUPER_USER_MAY_IMPERSONATE macro, 268, 305
QUEUE_SUPER_USERS macro, 268
Quill contrib, 656–683
Quill source code contrib configuration macro
    DATABASE_PURGE_INTERVAL, 661
    DATABASE_REINDEX_INTERVAL, 662
    DBMSD_ARGS, 662
    DBMSD_LOG, 662
    DBMSD_NOT_RESPONDING_TIMEOUT, 662
    DBMSD, 662
    QUILL_ADDRESS_FILE, 662
    QUILL_ARGS, 660
    QUILL_DBSIZE_LIMIT, 662
    QUILL_DB_IP_ADDR, 659, 661
    QUILL_DB_NAME, 661
    QUILL_DB_QUERY_PASSWORD, 662
    QUILL_DB_TYPE, 661
    QUILL_DB_USER, 661
    QUILL_ENABLED, 660
    QUILL_IS_REMOTELY_QUERYABLE, 662
    QUILL_JOB_HISTORY_DURATION, 661
    QUILL_LOG, 660
    QUILL_MAINTAIN_DB_CONN, 661
    QUILL_MANAGE_VACUUM, 662
    QUILL_NAME, 660
    QUILL_NOT_RESPONDING_TIMEOUT, 661
    QUILL_POLLING_PERIOD, 661
    QUILL_RESOURCE_HISTORY_DURATION, 662
    QUILL_RUN_HISTORY_DURATION, 661
    QUILL_SHOULD_REINDEX, 662
    QUILL_USE_SQL_LOG, 661
    QUILL, 660
quotas
    hierarchical quotas for a group, 348

RANDOM_CHOICE() macro
    use in submit description file, 931
rank attribute, 29
    examples, 29, 519
RANK expression, 354
RANK macro, 245, 370, 476, 477
RANK_FACTOR macro, 477
ReadUserLog class, 603
real user priority (RUP), 342

RecentBlockReadKbytes
job ClassAd attribute, 999
RecentBlockReads
job ClassAd attribute, 999
RecentBlockWriteKbytes
job ClassAd attribute, 999
RecentBlockWrites
job ClassAd attribute, 1000
RELEASE_DIR macro, 165, 207, 468
ReleaseReason
job ClassAd attribute, 1000
remote system call, 2, 3, 14
condor_shadow, 14, 50, 155
REMOTE_GROUP_RESOURCES_IN_USE macro, 725
REMOTE_PRIO_FACTOR macro, 291, 343
RemoteIwd
job ClassAd attribute, 1000
RemotePool
job ClassAd attribute, 1000
RemoteSysCpu
job ClassAd attribute, 1000
RemoteSysCpu macro, 712
RemoteUserCpu
job ClassAd attribute, 1000
RemoteUserCpu macro, 712
RemoteWallClockTime
job ClassAd attribute, 1000
REMOVE_SIGNIFICANT_ATTRIBUTES macro, 274
RemoveKillSig
job ClassAd attribute, 1000
REPLICATION macro, 327
REPLICATION_ARGS macro, 327
REPLICATION_DEBUG macro, 327
REPLICATION_INTERVAL macro, 327
REPLICATION_LIST macro, 327
REPLICATION_LOG macro, 327
REQUEST_CLAIM_TIMEOUT macro, 267
RequestCpus
job ClassAd attribute, 1000
RequestDisk
job ClassAd attribute, 1000
RequestedChroot
job ClassAd attribute, 1000
RequestMemory
job ClassAd attribute, 1000
REQUIRE_LOCAL_CONFIG_FILE macro, 210
requirements attribute, 29, 519
Requirements macro, 261, 262
RESERVE_AFS_CACHE macro, 235
RESERVED_DISK macro, 211, 1006
RESERVED_MEMORY macro, 250
RESERVED_SWAP macro, 55, 211
ResidentSetSize
job ClassAd attribute, 1000
resource
management, 2
offer, 2
owner, 157
request, 2
resource limits, 487
resource limits with cgroups, 488
ROOSTER_INTERVAL macro, 331
ROOSTER_MAX_UNHIBERNATE macro, 331
ROOSTER_UNHIBERNATE macro, 331
ROOSTER_UNHIBERNATE_RANK macro, 331
ROOSTER_WAKEUP_CMD macro, 331
ROTATE_HISTORY_DAILY macro, 274, 550
ROTATE_HISTORY_MONTHLY macro, 274, 550
RPM installation on Red Hat, 166
RUN macro, 208
RUN_FILETRANSFER_PLUGINS_WITH_ROOT
macro, 281
RunAsOwner, 428
RUNBENCHMARKS macro, 250, 363, 368
running a job
on a different architecture, 43
running as root, 155
running multiple programs, 19
RunningJobs
submitter ClassAd attribute, 1035

SBIN macro, 208
scalability
using the Grid Monitor, 567
SCHED_UNIV_RENICE_INCREMENT macro, 268
Schedd Cron functionality
see Daemon ClassAd Hooks, 547
SCHEDD_ADDRESS_FILE macro, 269
SCHEDD_ASSUME_NEGOTIATOR_GONE macro, 271
SCHEDD_ATTRS macro, 269
SCHEDD_AUDIT_LOG macro, 274, 551
SCHEDD_BACKUP_SPOOL macro, 272, 550

SCHEDD_CLUSTER_INCREMENT_VALUE macro, 273
SCHEDD_CLUSTER_INITIAL_VALUE macro, 273
SCHEDD_CLUSTER_MAXIMUM_VALUE macro, 273
SCHEDD_COLLECT_STATS_BY_<Name> macro, 274
SCHEDD_COLLECT_STATS_FOR_<Name> macro, 274
SCHEDD_CRON_<JobName>_ARGS macro, 338
SCHEDD_CRON_<JobName>_CWD macro, 338
SCHEDD_CRON_<JobName>_ENV macro, 338
SCHEDD_CRON_<JobName>_EXECUTABLE macro, 336
SCHEDD_CRON_<JobName>_JOB_LOAD macro, 337
SCHEDD_CRON_<JobName>_KILL macro, 338
SCHEDD_CRON_<JobName>_MODE macro, 336
SCHEDD_CRON_<JobName>_PERIOD macro, 336
SCHEDD_CRON_<JobName>_PREFIX macro, 336
SCHEDD_CRON_<JobName>_RECONFIG macro, 337
SCHEDD_CRON_<JobName>_RECONFIG_RERUN macro, 337
SCHEDD_CRON_CONFIG_VAL macro, 335
SCHEDD_CRON_JOBLIST macro, 336
SCHEDD_CRON_MAX_JOB_LOAD macro, 337
SCHEDD_CRON_NAME macro, 335
SCHEDD_DEBUG macro, 269
SCHEDD_ENABLE_SSH_TO_JOB macro, 329
SCHEDD_EXECUTE macro, 269
SCHEDD_EXPIRE_STATS_BY_<Name> macro, 274
SCHEDD_HOST macro, 207, 710, 796
SCHEDD_INTERVAL macro, 154, 266
SCHEDD_INTERVAL_TIMESLICE macro, 266
SCHEDD_JOB_QUEUE_LOG_FLUSH_DELAY macro, 273
SCHEDD_LOCK macro, 269
SCHEDD_MIN_INTERVAL macro, 266
SCHEDD_NAME macro, 207, 243, 269, 457
SCHEDD_PREEMPTION_RANK macro, 272, 477
SCHEDD_PREEMPTION_REQUIREMENTS macro, 272, 477
SCHEDD_QUERY_WORKERS macro, 265
SCHEDD_RESTART_REPORT macro, 275
SCHEDD_ROUND_ATTR_<xxxx> macro, 271
SCHEDD_SEND_VACATE_VIA_TCP macro, 273
SCHEDD_SUPER_ADDRESS_FILE macro, 226
SCHEDD_USE_SLOT_WEIGHT macro, 275, 731
SCHEDD_USES_STARTD_FOR_LOCAL_UNIVERSE macro, 262
ScheddIpAddr
    submitter ClassAd attribute, 1035
ScheddName
    submitter ClassAd attribute, 1035
scheduler universe, 16
SchedulerJobsIdle
    submitter ClassAd attribute, 1035
SchedulerJobsRunning
    submitter ClassAd attribute, 1035
scheduling
    dedicated, 69
    pie slice, 346
    pie spin, 346
scheduling jobs
    to execute at a specific time, 147
    to execute periodically, 150
SDK
    Chirp, 66
SEC_*_AUTHENTICATION macro, 316
SEC_*_AUTHENTICATION_METHODS macro, 317
SEC_*_CRYPTO_METHODS macro, 317
SEC_*_ENCRYPTION macro, 316
SEC_*_INTEGRITY macro, 316
SEC_*_NEGOTIATION macro, 316
SEC_<access-level>_SESSION_DURATION macro, 319
SEC_<access-level>_SESSION_LEASE macro, 319
SEC_ADMINISTRATOR_AUTHENTICATION macro, 401
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS macro, 401
SEC_ADMINISTRATOR_CRYPTO_METHODS macro, 413
SEC_ADMINISTRATOR_ENCRYPTION macro, 412
SEC_ADMINISTRATOR_INTEGRITY macro, 414
SEC_ADVERTISE_MASTER_AUTHENTICATION macro, 401
SEC_ADVERTISE_MASTER_AUTHENTICATION_METHODS macro, 401
SEC_ADVERTISE_MASTER_CRYPTO_METHODS macro, 413
SEC_ADVERTISE_MASTER_ENCRYPTION macro, 412
SEC_ADVERTISE_MASTER_INTEGRITY macro, 414

SEC_ADVERTISE_SCHEDD_AUTHENTICATION macro, 401
SEC_ADVERTISE_SCHEDD_AUTHENTICATION_METHODS macro, 401
SEC_ADVERTISE_SCHEDD_CRYPTO_METHODS macro, 413
SEC_ADVERTISE_SCHEDD_ENCRYPTION macro, 412
SEC_ADVERTISE_SCHEDD_INTEGRITY macro, 414
SEC_ADVERTISE_STARTD_AUTHENTICATION macro, 401
SEC_ADVERTISE_STARTD_AUTHENTICATION_METHODS macro, 401
SEC_ADVERTISE_STARTD_CRYPTO_METHODS macro, 413
SEC_ADVERTISE_STARTD_ENCRYPTION macro, 412
SEC_ADVERTISE_STARTD_INTEGRITY macro, 414
SEC_CLIENT_AUTHENTICATION macro, 401
SEC_CLIENT_AUTHENTICATION_METHODS macro, 401
SEC_CLIENT_CRYPTO_METHODS macro, 413
SEC_CLIENT_ENCRYPTION macro, 412
SEC_CLIENT_INTEGRITY macro, 413
SEC_CONFIG_AUTHENTICATION macro, 401
SEC_CONFIG_AUTHENTICATION_METHODS macro, 401
SEC_CONFIG_CRYPTO_METHODS macro, 413
SEC_CONFIG_ENCRYPTION macro, 412
SEC_CONFIG_INTEGRITY macro, 414
SEC_DAEMON_AUTHENTICATION macro, 401
SEC_DAEMON_AUTHENTICATION_METHODS macro, 401
SEC_DAEMON_CRYPTO_METHODS macro, 413
SEC_DAEMON_ENCRYPTION macro, 412
SEC_DAEMON_INTEGRITY macro, 414
SEC_DEFAULT_AUTHENTICATION macro, 401
SEC_DEFAULT_AUTHENTICATION_METHODS macro, 401
SEC_DEFAULT_AUTHENTICATION_TIMEOUT macro, 320
SEC_DEFAULT_CRYPTO_METHODS macro, 413
SEC_DEFAULT_ENCRYPTION macro, 412
SEC_DEFAULT_INTEGRITY macro, 413, 414
SEC_DEFAULT_SESSION_DURATION macro, 319
SEC_DEFAULT_SESSION_LEASE macro, 319
SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION macro, 321, 418
SEC_INVALIDATE_SESSIONS_VIA_TCP macro, 319
SEC_NEGOTIATOR_AUTHENTICATION macro, 401
SEC_NEGOTIATOR_AUTHENTICATION_METHODS macro, 401
SEC_NEGOTIATOR_CRYPTO_METHODS macro, 413
SEC_NEGOTIATOR_ENCRYPTION macro, 412
SEC_NEGOTIATOR_INTEGRITY macro, 414
SEC_OWNER_AUTHENTICATION macro, 401
SEC_OWNER_AUTHENTICATION_METHODS macro, 401
SEC_OWNER_CRYPTO_METHODS macro, 413
SEC_OWNER_ENCRYPTION macro, 412
SEC_OWNER_INTEGRITY macro, 414
SEC_PASSWORD_FILE macro, 320, 408
SEC_READ_AUTHENTICATION macro, 401
SEC_READ_AUTHENTICATION_METHODS macro, 401
SEC_READ_CRYPTO_METHODS macro, 413
SEC_READ_ENCRYPTION macro, 412
SEC_READ_INTEGRITY macro, 414
SEC_TCP_SESSION_DEADLINE macro, 320
SEC_TCP_SESSION_TIMEOUT macro, 320
SEC_WRITE_AUTHENTICATION macro, 401
SEC_WRITE_AUTHENTICATION_METHODS macro, 401
SEC_WRITE_CRYPTO_METHODS macro, 413
SEC_WRITE_ENCRYPTION macro, 412
SEC_WRITE_INTEGRITY macro, 414
security
  access levels, 394
  authentication, 401
  authorization, 414
  based on user authorization, 414
  changing the configuration, 424
  configuration examples, 422
  encryption, 412
  host-based, 420
  in HTCondor, 393–430
  integrity, 413
  running jobs as user nobody, 428
  sample configuration using pool password, 409
  sample configuration using pool password for startd advertisement, 409

sessions, 419
unified map file, 410
SENDMAIL macro, 211
sessions, 419
SETTABLE_ATTRS_<PERMISSION-LEVEL> macro, 225, 424, 425, 759
SETTABLE_ATTRS_ADMINISTRATOR macro, 424
SETTABLE_ATTRS_CONFIG macro, 202, 225, 424
SETTABLE_ATTRS_OWNER macro, 424
SETTABLE_ATTRS_WRITE macro, 424
SGE (Sun Grid Engine), 569
SGE_GAHP macro, 303
shadow, 14
SHADOW macro, 261
SHADOW_CHECKPROXY_INTERVAL macro, 277, 318
SHADOW_DEBUG macro, 276
SHADOW_JOB_CLEANUP_RETRY_DELAY macro, 277
SHADOW_LAZY_QUEUE_UPDATE macro, 276
SHADOW_LOCK macro, 276
SHADOW_MAX_JOB_CLEANUP_RETRIES macro, 277
SHADOW_QUEUE_UPDATE_INTERVAL macro, 276
SHADOW_RENICE_INCREMENT macro, 267
SHADOW_RUN_UNKNOWN_USER_JOBS macro, 277
SHADOW_SIZE_ESTIMATE macro, 211, 267
SHADOW_STATS_LOG macro, 223, 277
SHADOW_WORKLIFE macro, 277
shared file system
submission of jobs, 31
submission of jobs without one, 32
SHARED_PORT macro, 229, 435
SHARED_PORT_ARGS macro, 332
SHARED_PORT_AUDIT_LOG macro, 332, 333, 551
SHARED_PORT_DAEMON_AD_FILE macro, 332
SHARED_PORT_DEFAULT_ID macro, 229
SHARED_PORT_MAX_WORKERS macro, 332
SHARED_PORT_PORT macro, 331
SHELL macro, 873
SHUTDOWN_FAST_TIMEOUT macro, 240
SHUTDOWN_GRACEFUL_TIMEOUT macro, 225, 248
signal, 4, 14
SIGTSTP, 4, 14
SIGUSR2, 4, 14
SIGNIFICANT_ATTRIBUTES macro, 274, 346
Simple Object Access Protocol(SOAP), 590
Singularity, 499
SINGULARITY macro, 283
SINGULARITY_BIND_EXPR macro, 283
SINGULARITY_IMAGE_EXPR macro, 283
SINGULARITY_JOB macro, 283
SINGULARITY_TARGET_DIR macro, 283
SKIP_WINDOWS_LOGON_NETWORK macro, 299
SLOT<N>_CPU_AFFINITY macro, 281
SLOT<N>_EXECUTE macro, 209, 378
SLOT<N>_JOB_HOOK_KEYWORD macro, 333, 542
SLOT<N>_USER macro, 234, 429
SLOT_TYPE_<N> macro, 254, 378, 380
SLOT_TYPE_<N>_PARTITIONABLE macro, 254, 386
SLOT_TYPE_<n>_STARTD_ATTRS macro, 730
SLOT_WEIGHT macro, 257, 389, 704
slots
dynamic *condor_startd* provisioning, 385
subdividing slots, 385
SLOTS_CONNECTED_TO_CONSOLE macro, 253, 381, 1005
SLOTS_CONNECTED_TO_KEYBOARD macro, 253, 381, 1008
SlotWeight macro, 297
SLOW_CKPT_SPEED macro, 277
SMP machines
configuration, 377–391
SMTP_SERVER macro, 211
SOAP
Web Service API, 590
SOAP_LEAVE_IN_QUEUE macro, 328, 592
SOAP_SSL_CA_DIR macro, 329, 574
SOAP_SSL_CA_FILE macro, 329, 574
SOAP_SSL_DH_FILE macro, 329
SOAP_SSL_SERVER_KEYFILE macro, 329
SOAP_SSL_SERVER_KEYFILE_PASSWORD macro, 329
SOAP_SSL_SKIP_HOST_CHECK macro, 329
SOCKET_LISTEN_BACKLOG macro, 228
SOFT_UID_DOMAIN macro, 234, 426
Software Developer's Kit
Chirp, 66
SPOOL macro, 208
SSH_TO_JOB_<SSH-CLIENT>_CMD macro, 329
SSH_TO_JOB_SSH_KEYGEN macro, 330
SSH_TO_JOB_SSH_KEYGEN_ARGS macro, 330
SSH_TO_JOB_SSHD macro, 330
SSH_TO_JOB_SSHD_ARGS macro, 330

SSH_TO_JOB_SSHD_CONFIG_TEMPLATE macro, 330
StackSize
job ClassAd attribute, 1000
StageOutFinish
job ClassAd attribute, 1001
StageOutStart
job ClassAd attribute, 1001
START macro, 244, 252, 352, 368, 476
START_BACKFILL macro, 252, 363, 370, 479
START_DAEMONS macro, 240
START_LOCAL_UNIVERSE macro, 261, 1028
START_MASTER macro, 240
START_SCHEDULER_UNIVERSE macro, 262, 1028
startd
configuration, 351
Startd Cron functionality
see Daemon ClassAd Hooks, 547
STARTD_AD_REEVAL_EXPR macro, 295
STARTD_ADDRESS_FILE macro, 249
STARTD_ATTRS macro, 249, 384, 425, 478, 730
STARTD_AVAIL_CONFIDENCE macro, 260
STARTD_CLAIM_ID_FILE macro, 249
STARTD_COMPUTE_AVAIL_STATS macro, 260
STARTD_CRON_<JobName>_ARGS macro, 338
STARTD_CRON_<JobName>_CWD macro, 338
STARTD_CRON_<JobName>_ENV macro, 338
STARTD_CRON_<JobName>_EXECUTABLE macro, 336
STARTD_CRON_<JobName>_JOB_LOAD macro, 337
STARTD_CRON_<JobName>_KILL macro, 338
STARTD_CRON_<JobName>_MODE macro, 336
STARTD_CRON_<JobName>_PERIOD macro, 336
STARTD_CRON_<JobName>_PREFIX macro, 336
STARTD_CRON_<JobName>_RECONFIG macro, 337
STARTD_CRON_<JobName>_RECONFIG_RERUN macro, 337
STARTD_CRON_<JobName>_SLOTS macro, 336
STARTD_CRON_AUTOPUBLISH macro, 335
STARTD_CRON_CONFIG_VAL macro, 335
STARTD_CRON_JOBLIST macro, 336
STARTD_CRON_MAX_JOB_LOAD macro, 337
STARTD_CRON_NAME macro, 335
STARTD_DEBUG macro, 249
STARTD_EXPRS macro, 730
STARTD_HAS_BAD_UTMP macro, 248
STARTD_HISTORY macro, 247, 731
STARTD_JOB_ATTRS macro, 249, 730
STARTD_JOB_EXPRS macro, 292
STARTD_JOB_HOOK_KEYWORD macro, 333, 542
STARTD_MAX_AVAIL_PERIOD_SAMPLES macro, 260
STARTD_NAME macro, 250
STARTD_NOCLAIM_SHUTDOWN macro, 250
STARTD_PARTITIONABLE_SLOT_ATTRS macro, 246
STARTD_PUBLISH_DOTNET macro, 258
STARTD_PUBLISH_WINREG macro, 251
STARTD_RESOURCE_PREFIX macro, 253
STARTD_SENDS_ALIVES macro, 267
STARTD_SHOULD_WRITE_CLAIM_ID_FILE macro, 249
STARTD_SLOT_ATTRS macro, 253
STARTD_VM_ATTRS macro, 253
STARTD_VM_EXPRS macro, 253
STARTER macro, 247
Starter pre and post scripts, 927
STARTER_ALLOW_RUNAS_OWNER macro, 234, 428, 429, 484
STARTER_CHOOSES_CKPT_SERVER macro, 237, 446
STARTER_DEBUG macro, 278
STARTER_INITIAL_UPDATE_INTERVAL macro, 541
STARTER_JOB_ENVIRONMENT macro, 280
STARTER_JOB_HOOK_KEYWORD macro, 542
STARTER_LOCAL macro, 261
STARTER_LOCAL_LOGGING macro, 278
STARTER_LOG_NAME_APPEND macro, 278
STARTER_RLIMIT_AS macro, 282
STARTER_STATS_LOG macro, 223, 282
STARTER_UPDATE_INTERVAL macro, 278, 541
STARTER_UPDATE_INTERVAL_TIMESLICE macro, 278
STARTER_UPLOAD_TIMEOUT macro, 280
starting HTCondor
Unix platforms, 170
Windows platforms, 180
state
of a machine, 355
transitions, 360–370
transitions summary, 368
state and activities figure, 360

STATE_FILE macro, 327
STATISTICS_TO_PUBLISH macro, 215, 341, 1030–1032
STATISTICS_TO_PUBLISH_LIST macro, 216
STATISTICS_WINDOW_QUANTUM macro, 217
STATISTICS_WINDOW_QUANTUM_<collection> macro, 217
STATISTICS_WINDOW_SECONDS macro, 216, 1028
STATISTICS_WINDOW_SECONDS_<collection> macro, 216
status
    of DAG nodes, 127
    of queued jobs, 50
StreamErr
    job ClassAd attribute, 1001
StreamOut
    job ClassAd attribute, 1001
STRICT_CLASSAD_EVALUATION macro, 215, 505
submit commands, 898
    +PostArgs, 928
    +PostArguments, 928
    +PostCmd, 928
    +PostEnv, 928
    +PostEnvironment, 928
    +PreArgs, 927
    +PreArguments, 927
    +PreCmd, 927
    +PreEnv, 928
    +PreEnvironment, 928
    $ENV macro, 931
    $RANDOM_CHOICE() macro, 931
    accounting_group, 139, 349, 710, 922, 986
    accounting_group_user, 139, 349, 710, 922
    allow_startup_script, 913
    append_files, 913
    arguments, 37, 38, 48, 64, 68, 73, 84, 331, 898, 902, 904
    batch_queue, 915
    boinc_authenticator_file, 577, 915, 986
    buffer_block_size, 913
    buffer_files, 913
    buffer_size, 913
    compress_files, 914
    concurrency_limits, 491, 922
    concurrency_limits_expr, 491, 922
    copy_to_spool, 922, 923
    coresize, 923
    cream_attributes, 577, 915
    cron_day_of_month, 150, 923
    cron_day_of_week, 150, 923
    cron_hour, 150, 923
    cron_minute, 150, 923
    cron_month, 150, 923
    cron_prep_time, 152, 923
    cron_window, 152, 923
    dagman_log, 923
    deferral_prep_time, 149, 152, 923
    deferral_time, 148, 149, 923, 924
    deferral_window, 148, 152, 923, 924
    delegate_job_GSI_credentials_lifetime, 318, 915
    description, 924, 994
    docker_image, 146, 497, 922
    dont_encrypt_input_files, 906
    dont_encrypt_output_files, 906
    ec2_access_key_id, 570, 916, 989
    ec2_ami_id, 570, 916, 989
    ec2_availability_zone, 916
    ec2_block_device_mapping, 571, 916, 989
    ec2_ebs_volumes, 916
    ec2_elastic_ip, 916, 989
    ec2_iam_profile_arn, 571, 916, 989
    ec2_iam_profile_name, 571, 916, 989
    ec2_instance_type, 571, 916, 989
    ec2_keypair, 916, 989
    ec2_keypair_file, 571, 877, 916, 990
    ec2_parameter_<name>, 916
    ec2_parameter_names, 572, 916, 989
    ec2_secret_access_key, 570, 916, 990
    ec2_security_groups, 571, 916, 990
    ec2_security_ids, 571, 917, 990
    ec2_spot_price, 572, 917, 989
    ec2_tag_<name>, 917
    ec2_tag_names, 917, 989
    ec2_user_data, 571, 917, 990
    ec2_user_data_file, 571, 917, 990
    ec2_vpc_id, 571
    ec2_vpc_ip, 917
    ec2_vpc_subnet, 571, 917
    email_attributes, 924
    encrypt_execute_directory, 319, 320, 906, 990
    encrypt_input_files, 906
    encrypt_output_files, 906

environment, 42, 68, 848, 900, 901
error, 18, 37, 39, 54, 139, 846, 901, 904, 908, 919
executable, 34, 48, 63, 64, 68, 73, 140, 147, 214, 560, 570, 575, 577, 897, 901, 908, 913
fetch_files, 914
file_remaps, 914
gce_auth_file, 575, 917, 991
gce_image, 575, 917, 991
gce_json_file, 576, 917, 991
gce_machine_type, 575, 917, 991
gce_metadata, 575, 917, 991
gce_metadata_file, 575, 917, 991
gce_preemptible, 708, 918, 991
getenv, 42, 901
globus_rematch, 918
globus_resubmit, 918
globus_rsl, 557, 564, 918
globus_xml, 557
grid_resource, 554, 557, 558, 560, 561, 567, 568, 570, 575–578, 580, 904, 918, 991
hold, 910
hold_kill_sig, 920
IF/ELSE syntax, 23
image_size, 924
included, 922
initialdir, 18, 34, 36, 37, 136, 155, 430, 647, 846, 904, 924
input, 17, 18, 33, 34, 36, 37, 139, 285, 560, 847, 896, 901, 902, 904, 907, 908, 920
jar_files, 34, 64, 920
java_vm_args, 493, 920
job_ad_information_attrs, 924
job_lease_duration, 156, 925
job_machine_attrs, 269, 925
job_machine_attrs_history_length, 269, 925
job_max_vacate_time, 925, 926
JobBatchName, 924
keep_claim_idle, 100, 910
keystore_alias, 568, 919
keystore_file, 568, 919
keystore_passphrase_file, 568, 919
kill_sig, 920, 921, 925
kill_sig_timeout, 925
leave_in_queue, 910
load_profile, 645, 646, 925, 927
local_files, 915
Log, 529, 603
log, 54, 70, 285, 550, 896, 902, 907, 923
log_xml, 902
machine_count, 68, 70, 920
match_list_length, 925
max_job_retirement_time, 926
max_retries, 909
max_transfer_input_mb, 907
max_transfer_output_mb, 907
MyProxyCredentialName, 919
MyProxyHost, 919
MyProxyNewProxyLifetime, 919
MyProxyPassword, 919
MyProxyRefreshThreshold, 919
MyProxyServerDN, 919
next_job_start_delay, 911
nice_user, 343, 926
noop_job, 926, 927
noop_job_exit_code, 927
noop_job_exit_signal, 927
nordugrid_rsl, 568, 919
notification, 59, 902, 939, 994
notify_user, 902
on_exit_hold, 911
on_exit_hold_reason, 911
on_exit_hold_subcode, 911
on_exit_remove, 150, 153, 911, 912
output, 17, 18, 33, 37, 39, 139, 285, 584, 847, 896, 902–904, 907, 908, 920
output_destination, 40, 465, 466, 907
periodic_hold, 912
periodic_hold_reason, 912
periodic_hold_subcode, 912
periodic_release, 244, 271, 912
periodic_remove, 244, 912
priority, 97, 848, 903
queue, 20, 28, 48, 68, 71, 316, 895, 903, 904, 931, 932
rank, 19, 29, 41, 579, 904
remote_initialdir, 927
remove_kill_sig, 920
rendezvousdir, 927
request_<name>, 380, 905
request_cpus, 74, 283, 904
request_disk, 283, 904
request_GPUs, 905

request_memory, 18, 283, 691, 905, 997
Requirements, 47, 54, 71, 651
requirements, 29, 40, 70, 545, 578, 897, 904–906
retry_until, 910
run_as_owner, 643, 645, 893, 925, 927
should_transfer_files, 32, 40, 73, 907
signal-number, 920, 921, 925
skip_filechecks, 907
stack_size, 927
stream_error, 907
stream_input, 908
stream_output, 908
submit_event_notes, 927
success_exit_code, 910
transfer_error, 919
transfer_executable, 48, 908
transfer_input, 919
transfer_input_files, 34, 36–38, 40, 64, 73, 75, 142, 285, 464, 465, 655, 896, 906–908, 920, 921, 927, 928
transfer_output, 920
transfer_output_files, 33, 35, 36, 285, 465, 577, 584, 648, 896, 906–908, 920
transfer_output_remaps, 36, 909
universe, 13, 48, 136, 553, 557, 897, 904, 1029
use_x509userproxy, 920
vm_checkpoint, 245, 921
vm_disk, 140–142, 921
vm_macaddr, 141, 921
vm_memory, 140, 141, 905, 921
vm_networking, 141, 921
vm_networking_type, 140, 141, 921
vm_no_output_vm, 921
vm_type, 921
vmware_dir, 141, 495, 921, 922
vmware_should_transfer_files, 141, 921
vmware_should_transfer_files = true, 495
vmware_snapshot_disk, 142, 495, 922
want_graceful_removal, 271, 925
want_remote_io, 915
WantNameTag, 917
when_to_transfer_output, 32, 33, 40, 648, 909
x509userproxy, 42, 564, 568, 576, 920, 1002, 1003
xen_initrd, 922
xen_kernel, 142, 922
xen_kernel_params, 922
xen_root, 142, 922
submit description file, 16
automatic variables, 22
contents of, 17
examples, 17–20
function macros, 26
grid universe, 560
including commands from elsewhere, 23
submit host
policy configuration, 391
submit machine, 158
submit requirements, 392
SUBMIT_ATTRS macro, 285, 429, 488, 723, 730
SUBMIT_EXPRS macro, 488
SUBMIT_MAX_PROCS_IN_CLUSTER macro, 285
SUBMIT_PUBLISH_WINDOWS_OSVERSIONINFO macro, 708
SUBMIT_REQUIREMENT_<Name> macro, 275, 392
SUBMIT_REQUIREMENT_<Name>_REASON macro, 275, 392
SUBMIT_REQUIREMENT_NAMES macro, 275, 392
SUBMIT_SEND_RESCHEDULE macro, 285
SUBMIT_SKIP_FILECHECKS macro, 285
SubmitterAutoregroup
job ClassAd attribute, 1001
SubmitterGlobalJobId
job ClassAd attribute, 1001
SubmitterGroup
job ClassAd attribute, 1001
SubmitterNegotiatingGroup
job ClassAd attribute, 1001
SubmitterTag
submitter ClassAd attribute, 1035
substitution macro
in submit description file, 930
<SUBSYS> macro, 239
<SUBSYS>_ADDRESS_FILE macro, 225
<SUBSYS>_ADMIN_EMAIL macro, 211
<SUBSYS>_ARGS macro, 239
<SUBSYS>_ATTRS macro, 226
<SUBSYS>_DAEMON_AD_FILE macro, 226
<SUBSYS>_DEBUG macro, 221
<SUBSYS>_DEBUG macro levels
D_ACCOUNTANT, 222
D_ALL, 221
D_CATEGORY, 223

D_CKPT, 222
D_COMMAND, 222
D_DAEMONCORE, 221
D_FDS, 223
D_FULLDEBUG, 221
D_HOSTNAME, 222
D_JOB, 222
D_KEYBOARD, 222
D_LOAD, 222
D_MACHINE, 222
D_MATCH, 222
D_NETWORK, 222
D_PID, 223
D_PRIV, 222
D_PROCFAMILY, 222
D_PROTOCOL, 223
D_SECURITY, 222
D_STATS, 223
D_SUB_SECOND, 223
D_SYSCALLS, 222
D_TIMESTAMP, 223
<SUBSYS>_ENABLE_SOAP_SSL macro, 328
<SUBSYS>_EXPRS macro, 226
<SUBSYS>_<LEVEL>_LOG macro, 223
<SUBSYS>_LOCK macro, 219
<SUBSYS>_LOG macro, 218
<SUBSYS>_LOG_KEEP_OPEN macro, 219
<SUBSYS>_MAX_FILE_DESCRIPTORS, 229
<SUBSYS>_SOAP_SSL_PORT macro, 328
<SUBSYS>_SUPER_ADDRESS_FILE macro, 226
<SUBSYS>_TIMEOUT_MULTIPLIER macro, 232
<SUBSYS>_USERID macro, 239
SUBSYSTEM macro, 199
subsystem names, 199
supported platforms, 5
SUSPEND macro, 244, 369
SYSAPI_GET_LOADAVG macro, 214
SYSTEM_IMMUTABLE_JOB_ATTRS macro, 276
SYSTEM_JOB_MACHINE_ATTRS macro, 268, 269, 925, 987
SYSTEM_JOB_MACHINE_ATTRS_HISTORY_LENGTH macro, 269
SYSTEM_PERIODIC_HOLD macro, 270, 992
SYSTEM_PERIODIC_HOLD_REASON macro, 271
SYSTEM_PERIODIC_HOLD_SUBCODE macro, 271
SYSTEM_PERIODIC_RELEASE macro, 271
SYSTEM_PERIODIC_REMOVE macro, 271
SYSTEM_PROTECTED_JOB_ATTRS macro, 276
SYSTEM_VALID_SPOOL_FILES macro, 286, 821

TARGET., ClassAd scope resolution prefix, 515
TCP, 441
sending updates, 441
TCP_FORWARDING_HOST macro, 230, 231
TCP_KEEPALIVE_INTERVAL macro, 217
TCP_UPDATE_COLLECTORS macro, 232, 442
TEMP_DIR macro, 209
thread
kernel-level, 4, 14
user-level, 4, 14
TILDE macro, 199
TMP_DIR macro, 209
TOOL_DEBUG macro, 223
TotalSuspensions
job ClassAd attribute, 1001
TOUCH_LOG_INTERVAL macro, 221, 550
TRANSFER_IO_REPORT_INTERVAL macro, 265
TRANSFER_IO_REPORT_TIMESPANS macro, 264, 265, 1029–1032
TRANSFER_QUEUE_USER_EXPR macro, 264, 1029–1032
TRANSFERER macro, 328
TRANSFERER_DEBUG macro, 328
TRANSFERER_LOG macro, 328
TransferErr
job ClassAd attribute, 1001
TransferExecutable
job ClassAd attribute, 1001
TransferIn
job ClassAd attribute, 1001
TransferInputSizeMB
job ClassAd attribute, 1001
TransferOut
job ClassAd attribute, 1002
TransferQueued
job ClassAd attribute, 1002
transferring files, 32
TransferringInput
job ClassAd attribute, 1002
TransferringOutput
job ClassAd attribute, 1002

TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN macro, 224
TRUNC_<SUBSYS>_LOG_ON_OPEN macro, 219, 224, 550
TRUST_UID_DOMAIN macro, 234

UDP, 441
  lost datagrams, 441
UDP_LOOPBACK_FRAGMENT_SIZE macro, 233
UDP_NETWORK_FRAGMENT_SIZE macro, 233
UID
  effective, 426
  potential risk running jobs as user nobody, 428
  real, 425
UID_DOMAIN macro, 201, 233, 426, 427, 438, 902
UIDs in HTCondor, 425–430
UNAME_ARCH macro, 200
UNAME_OPSYS macro, 200
unauthenticated, 412, 417
unclaimed state, 355, 362
UNHIBERNATE macro, 259, 331, 501
Unicore, 568
UNICORE_GAHP macro, 302
universe, 13
  docker, 16, 146, 497
  Grid, 13, 15
  grid, 557
  grid, grid type gt2, 560
  grid, grid type gt5, 564
  Java, 15
  java, 13
  job ClassAd attribute definitions
    grid = 9, 995
    java = 11, 995
    linda = 3 (no longer used), 995
    local = 12, 995
    mpi = 8, 995
    parallel = 10, 995
    pipe = 2 (no longer used), 995
    pvm = 4 (no longer used), 995
    pvmd = 6 (no longer used), 995
    scheduler = 7, 995
    standard = 1, 995
    vanilla = 5, docker = 5, 995
    vm = 13, 995
  local, 16
  parallel, 13, 16
  scheduler, 16
  set up for the docker universe, 497
  set up for the vm universe, 493
  standard, 13
  vanilla, 13, 15
  vm, 13, 16, 139
Unix
  alarm, 4, 14
  exec, 4, 14
  flock, 4, 14
  fork, 4, 14
  large files, 4, 15
  lockf, 4, 14
  mmap, 4, 14
  pipe, 4, 14
  semaphore, 4, 14
  shared memory, 4, 14
  sleep, 4, 14
  socket, 4, 14
  system, 4, 14
  timer, 4, 14
Unix administrator, 163
Unix directory
  execute, 164
  lock, 164
  log, 164
  spool, 164
Unix installation
  download, 161
Unix user
  condor, 163
  root, 163
unmapped, 412
UPDATE_COLLECTOR_WITH_TCP macro, 232, 442
UPDATE_INTERVAL macro, 247, 335, 362, 454
UPDATE_OFFSET macro, 247
UPDATE_VIEW_COLLECTOR_WITH_TCP macro, 232, 442
upgrading
  items to be aware of, 689
URL file transfer, 40, 464
USE configuration syntax, 201
USE_AFS macro, 236
USE_CKPT_SERVER macro, 237, 446
USE_CLONE_TO_CREATE_PROCESSES macro, 227

USE_GID_PROCESS_TRACKING macro, 298, 485
USE_NFS macro, 235
USE_PID_NAMESPACES macro, 282
USE_PROCD macro, 298, 486
USE_PROCESS_GROUPS macro, 244
USE_PSS macro, 282
USE_RESOURCE_REQUEST_COUNTS macro, 294
USE_SHARED_PORT macro, 229, 331
USE_VISIBLE_DESKTOP macro, 280, 647
user
    priority, 60
User Log Reader API, 603
user manual, 9–156
user nobody
    potential security risk with jobs, 428
user priority, 342
    effective (EUP), 343
    real (RUP), 342
USER_CONFIG_FILE macro, 185, 210
USER_JOB_WRAPPER macro, 279, 487
UserLog
    job ClassAd attribute, 1002
USERLOG_FILE_CACHE_CLEAR_INTERVAL macro, 220
USERLOG_FILE_CACHE_MAX macro, 220
USERNAME macro, 201

vacate, 61
VALID_COD_USERS macro, 527
VALID_SPOOL_FILES macro, 286, 325, 456, 821
viewing
    log files, 685
virtual machine
    configuration, 474
    running HTCondor jobs under, 473
virtual machine universe, 139–146
virtual machines, 493
vm universe, 16, 139
    checkpoints, 142
    ftl, 144
    submit commands specific to VMware, 141
    submit commands specific to Xen, 142
VM_GAHP_LOG macro, 322
VM_GAHP_REQ_TIMEOUT macro, 322
VM_GAHP_SERVER macro, 322
VM_MAX_NUMBER macro, 322, 1016
VM_MEMORY macro, 322, 1016
VM_NETWORKING macro, 323
VM_NETWORKING_BRIDGE_INTERFACE macro, 323
VM_NETWORKING_DEFAULT_TYPE macro, 323
VM_NETWORKING_TYPE macro, 323
VM_RECHECK_INTERVAL macro, 322
VM_SOFT_SUSPEND macro, 322
VM_STATUS_INTERVAL macro, 322
VM_TYPE macro, 322, 494, 1017
VM_UNIV_NOBODY_USER macro, 323
VMP_HOST_MACHINE macro, 324, 474
VMP_VM_LIST macro, 324, 474
VMWARE_BRIDGE_NETWORKING_TYPE macro, 324
VMWARE_LOCAL_SETTINGS_FILE macro, 324
VMWARE_NAT_NETWORKING_TYPE macro, 324
VMWARE_NETWORKING_TYPE macro, 324
VMWARE_PERL macro, 323
VMWARE_SCRIPT macro, 323

WALL_CLOCK_CKPT_INTERVAL macro, 268
WANT_HOLD macro, 244, 992
WANT_HOLD_REASON macro, 245
WANT_HOLD_SUBCODE macro, 245
WANT_SUSPEND macro, 246, 368
WANT_UDP_COMMAND_SOCKET macro, 214, 292
WANT_VACATE macro, 246, 247, 369
WARN_ON_UNUSED_SUBMIT_FILE_MACROS macro, 285, 896
Web Service API, 590
    condor_schedd daemon command port, 593
    file transfer, 592
    job submission, 591
    transactions, 590
WEB_ROOT_DIR macro, 328
WEIGHTED_JOBS_RUNNING macro, 725
WeightedIdleJobs
    submitter ClassAd attribute, 1035
WeightedJobsRunning macro, 726
WeightedRunningJobs
    submitter ClassAd attribute, 1035
WINDOWED_STAT_WIDTH macro, 266
Windows
    HTCondor daemon names, 180
    installation, 172–180
        initial file size, 172
        location of files, 176

preparation, 173
required disk space, 173
unattended install, 176
loading account profile, 645
manual install, 179
release notes, 641
starting the HTCondor service, 180
WINDOWS_FIREWALL_FAILURE_RETRY macro, 243
WINDOWS_RMDIR macro, 338
WINDOWS_RMDIR_OPTIONS macro, 338
WorkHours macro, 373

X509_USER_PROXY environment variable, 42
X509UserProxy
job ClassAd attribute, 1002
X509UserProxyEmail
job ClassAd attribute, 1002
X509UserProxyExpiration
job ClassAd attribute, 1002
X509UserProxyFirstFQAN
job ClassAd attribute, 1002
X509UserProxyFQAN
job ClassAd attribute, 1002
X509UserProxySubject
job ClassAd attribute, 1003
X509UserProxyVOName
job ClassAd attribute, 1003
XEN_BOOTLOADER macro, 324