Journal of Physics: Conference Series **664** (2015) 062014 doi:10.1088/1742-6596/664/6/062014

How much higher can HTCondor fly?

E M Fajardo[1], J M Dost[1], B Holzman[2], T Tannenbaum[3], J Letts[1], A Tiradani[2], B Bockelman[4], J Frey[3], D Mason[2]

[1] University of California San Diego, La Jolla, CA, USA
[2] Fermi National Accelerator Laboratory, USA
[3] University of Wisconsin, Madison, WI, USA
[4] University of Nebraska, Lincoln, NE, USA

E-mail: efajardo@physics.ucsd.edu

Abstract. The HTCondor high throughput computing system is heavily used in the high energy physics (HEP) community as the batch system for several WorldWide LHC Computing Grid (WLCG) resources. Moreover, it is the backbone of GlideInWMS, the pilot system used by the computing organization of the Compact Muon Solenoid (CMS) experiment. To prepare for LHC Run 2, we probed the scalability limits of new versions and configurations of HTCondor with a goal of reaching 200,000 simultaneous running jobs in a single internationally distributed dynamic pool.

In this paper, we first describe how we created an opportunistic distributed testbed capable of exercising runs with 200,000 simultaneous jobs without impacting production. This testbed methodology is appropriate not only for scale testing HTCondor, but potentially for many other services. In addition to the test conditions and the testbed topology, we include the suggested configuration options used to obtain the scaling results, and describe some of the changes to HTCondor inspired by our testing that enabled sustained operations at scales well beyond previous limits.

1. Introduction

HTCondor is a distributed high throughput computing system, providing batch services and other features [1]. The High Energy Physics (HEP) community has broadly adopted it on numerous Worldwide LHC Computing Grid (WLCG) computing clusters. Furthermore HTCondor is the central piece of GlideinWMS [2], the resource provisioning layer for the Compact Muon Solenoid (CMS) experiment as well as other physics and non-physics Virtual Organizations (VOs).

CMS anticipates that, during LHC Run II (2015-2017), up to 200,000 cores would be available for it to utilize. The proper and efficient use of these resources will be essential to achieve its physics research goals. However, as any other distributed system, HTCondor does not infinitely scale with the number of resources to manage. In Section 2, we describe the requirements and the challenge of scaling HTCondor to those limits. Afterwards, Section 4 explains the different improvements the HTCondor development team made to meet CMS scalability requirements. For the purpose of better understanding of the enhancements done we also included overview Sections 1.1 and 1.2 for the readers not familiar with HTCondor and GlideinWMS respectively.

Although CMS plans to have 200,000 cores available at peak times, having such resources available for a testing setup was too costly. Therefore a novel approach for the testing was


Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

needed. In Section 3 we show the solutions to this problem as well as the final setup for testing that allowed us to reach the proposed goal. Finally, while this work was motivated by CMS requirements, in Section 5 we discuss lessons learned and how any other VO or computing cluster will benefit from the improvements in HTCondor and GlideinWMS software that resulted from this work.

1.1. Overview of HTCondor Components

An HTCondor pool consists of three pieces: a central manager running both a *Collector* and a *Negotiator* daemon, one or more execute nodes running a *Startd* daemon, and one or more submit nodes running a *Schedd* daemon [1]. Users typically login to a submit node and use command-line tools that communicate with the local Schedd to submit jobs, query the state of a job, cancel jobs, etc. The Schedd manages the job queue and serves as a scheduler, mapping jobs onto execute nodes which it has claimed for its own use. If a Schedd does not have have enough claimed execute node resources, it sends requests for more resources from the central manager. The Negotiator receives these resource requests and attempts to locate additional execute nodes that can be assigned to the requesting Schedd via a *matchmaking* process [3]. On the execute node, the Startd daemon manages the compute resources and instances of jobs running on that node. All daemons in the HTCondor system also send regular status information to the Collector daemon running on the central manager. The Collector serves as a database of semi-structured data and can provide a centralized view of the status of all execute nodes in the compute pool.

1.2. Overview of GlideinWMS Components

GlideinWMS is a system for creating a dynamically growing HTCondor pool of execute nodes from a set of grid *entry points*, growing or shrinking the available resources based on job workload demand. To achieve this goal, it separates the grid submission and node validation tasks to a component called the *Factory* (serving multiple VOs) [4] and demand computation and resource requests to the *GlideIn Frontend* (serving a single VO). The frontend consists of a running daemon and is meant to be matched with a single HTCondor pool. The fronted queries *Schedd*s to derive resource demands based upon enqueued jobs and passes this information to the *factory*.

The *factory* as directed by the *fronted* can submit pilots (also referred to as *glideins* within the HTCondor ecosystem) to grid, cloud and opportunistic resources. Once a resource has been acquired and validated - a pilot job is successfully running on a computing resource - the pilot launches a properly configured HTCondor *Startd* binary on the execute node. In terms of functionality, the Startd run by GlideinWMS is identical to a Startd run in a "regular" pool.

2. The Scaling Challenge

During the LHC Run I, CMS collected and processed data taken from proton-proton collisions at a center-of-mass energy of 7 TeV in 2011 and at 8 TeV in 2012. However for Run II, it plans to increase the collision energy up to 13.5 TeV along with increased luminosity [5]. Thus the amount of data to be processed will increase as well as the pressure on the computing infrastructure to operate efficiently. To successfully achieve its physics research goals, WLCG sites have pledged CMS approximately 110,000 cores; adding in the resources at CERN, these may peak at 130,000 cores. Taking into account growth over the next three years, cloud, and opportunistic resources, we expect the core count may reach 200,000 during LHC Run II. Unlike Run I, during Run II CMS will have all of its resources provisioned by a single HTCondor pool [6] to maximize flexibility. Hence, we set out to demonstrate a single HTCondor pool, provisioned by GlideinWMS, which can run stably at that level.

Given that all resources will join a single pool, the central manager resiliency becomes essential. Fortunately HTCondor already provides resiliency for the central manager in the

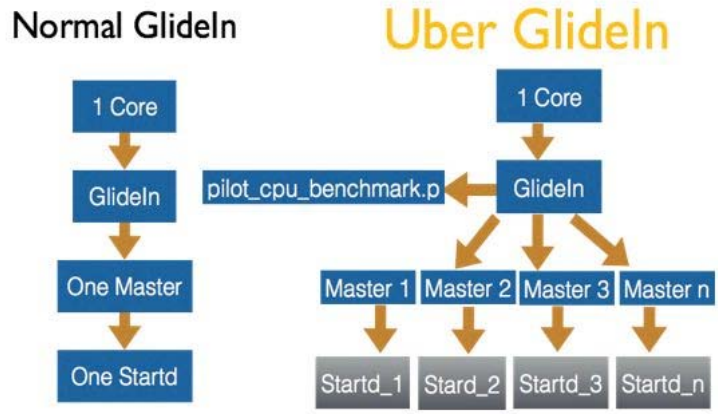


Figure 1. The comparison between a standard GlideIn and the Uber GlideIn job. As some sites insisted all jobs must consume CPU (WLCG accounting only includes CPU usage), the *pilot_cpu_benchmark.p* script was optionally run alongside the glidein.

form of the high availability (**HA**) mode [7].

Although it is usual to have several *Schedds* serving a single pool, the previous best result for parallel running jobs in a single scheduler is 41,000 jobs [8]. Thus to fill a 200,000 core pool (accounting for room for upgrades/downtimes) around seven *Schedds* would be needed. As each host requires a marginal operational effort to maintain and operate, we aimed to leverage recent scalability improvements to increase the per-host results. These are explained in Section 4.5; we doubled prior results.

Even though the original challenge was to reach 200,000 parallel running jobs in an HTCondor pool, obtaining that amount of available cores for testing purposes proved a challenge on its own. Such resources could be purchased from commercial clouds such as Amazon Web Services (AWS). The bill - based on current AWS "spot" pricing [9] - would have been approximately:

$$\frac{\$0.013}{\text{hour} * \text{core}} * \frac{24 \text{ hours}}{\text{day}} * \frac{30 \text{ days}}{\text{month}} * 200\text{k cores} = \frac{\$1,872,000}{\text{month}} \quad (1)$$

As the cycles of test-report-fix-test for HTCondor took several months, the bill would have been in the orders of millions of US dollars. We decided to performed this taskon CMS resources during the Long Shutdown 1 (LS1). However, CMS does not currently have 200,000 cores and additional kept resources busy with user analysis and simulation workflows. Hence the need for a novel approach to obtain the required resources: the *Uber GlideIn*.

3. Uber Glidein

In a typical GlideinWMS *glidein*, a single pilot acquires one core and then runs several payloads sequentially through its lifetime. Since we were not interested on the payload results, decided to run *sleep jobs* (payloads which use close to no CPU). Our objective was on scaling the number of *startd*s a single HTCondor pool can handle - hence we modified the pilot to start n instances of *startd* per core. In most of the tests, we used $n = 64$. We term such a pilot an *Uber GlideIn*. Figure 1 illustrates the difference between a normal *GlideIn* and an *Uber glidein*. The *Uber glidein* is a natural extension of a sleeper slot (usually provided by a site) [10] into a pilot, and might as well change the way Internet-facing grid services are scale tested in the future.

In addition to idle CMS resources, we ran *Uber Glideins* in some Open Science Grid (OSG)[11] and European Grid Initiative (EGI)[12] opportunistic resources. The driver was not necessarily an economic one (versus cloud resources), but that the complexity also made the testing environment closer to the production one. In production, the network latency of running executing nodes over the WAN spread all over the world are known to cause problems[13]. This



Figure 2. Uber GlideIn providing 150k sleep slots while only using 20k real cores.

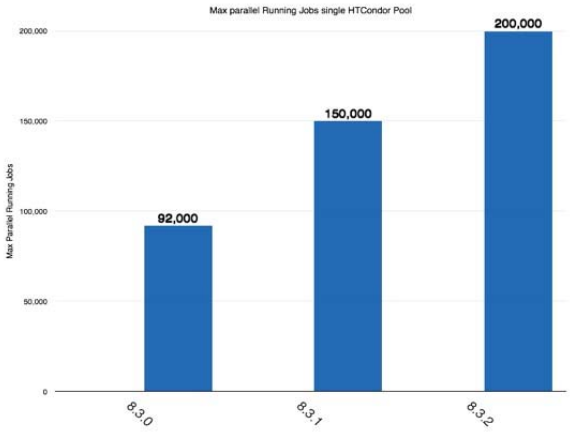


Figure 3. The maximum number of running jobs achieved in a single HTCondor pool through several development versions; the work covered in this paper started with HTCondor 8.3.0.

approached turned out to be successful as can be seen in Figure 2, in which 150,000 HTCondor execution nodes were reached only using 20,000 real grid cores.

4. Scaling Results

Once we devised a mechanism for acquiring the required testbed resources, we focused on the "testing, reporting, fixing" cycle for scaling HTCondor. The results of the various improvements are shown in Figure 3. Not all improvements came from new HTCondor releases; some were also a result of configuration changes in HTCondor, GlideinWMS, or the host/kernel parameters.

4.1. Central Manager

As a highly-available and centralized component in *HTCondor*'s architecture, the central manager is sensitive to scaling issues. In a standard setup, a single *Collector* runs in the same host as the *Negotiator*. The *Collector*, which receives status updates from all executing nodes, had scaling problems when the number of Startds was larger 2,000 for nodes located on the local-area network, or just 200 for remote Startds communicating over a high-latency wide-area network largely due to the number of network round-trips required to securely authenticate a Startd joining the compute pool. Accordingly, a two-tier *Collector* architecture has been in use by GlideinWMS case [13] since 2010. In the two-tier architecture, a set of "child" collectors receives updates from a subset of the remote *Startd*s, and then the children forward the updates a top-level *Collector* over the local network interface. This hides the network latency from the parent collector, improving overall scalability. Prior to HTCondor 8.3.0, the recommended ratio between Startds and child collectors was 200 to one [8, 13]. To achieve 200,000 running jobs, we would need 1,000 child-collectors, which proved difficult to manage. Working with the HTCondor team, the Grid Security Infrastructure (GSI) [14] authentication implementation was changed from blocking to non-blocking. Because a single GSI authentication requires multiple network round-trips, with execute nodes are distributed across continents a single synchronous GSI authentication would take 500+ milliseconds with the *Collector* process blocked on network I/O for most of that time. The asynchronous re-implementation allows the *Collector* to authenticate many Startds in parallel. In addition we performed some kernel tweaking and HTCondor and



Figure 4. The final recommended HTCondor topology for deployments in order to run more than 100k parallel running jobs.

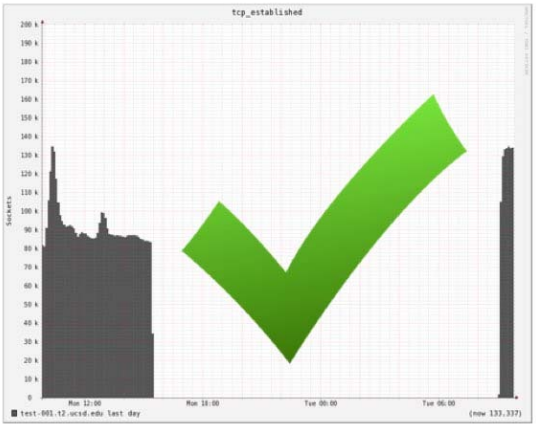


Figure 5. Scenario of bringing a single CCB down for more than 12 hours and having all jobs reconnecting back.

GlideinWMS configuration changes: the value of the *UDP Buffer size* was increased the order of $10MB$ and "renicing" the parent Collector process. The configuration changes modified the periodicity in which the *startd* reported back to the *Secondary Collector* from the default average of every 5 minutes to every 10 miuntes. Additionally, we decreased the frequency the *VO Frontent* queried the parent *Collector*. The reasoning behind those changes was to reduce the load on the parent *Collector* to allow it to process more updates from the children. All together, these changes allowed us to change our ratio of Startds per child collector to 1,000 to one.

4.2. Condor Connection Broker

Standard HTCondor deployments have the *Scheduler* initiate connections to the *Startd* directly; this assumes that the Startd is either on the public Internet or the submit and execute hosts are on the same private network. This is not true in the *GlideinWMS* use case as the worker nodes may be behind restrictive firewalls or NAT. To establish connections in such a case, HTCondor utilizes the *Condor Connection Broker* (CCB) and *Shared Port* [15]. The CCB acts as a trusted "middle man" between the executing node and the scheduler. The Startd maintains an authenticated outgoing TCP connection to the CCB; instead of the *Schedd* connecting to the Startd directly, it will first connect to the CCB and request the CCB to forward a connection request to the Startd. Within the connection request, the Schedd instructs the Startd contact it on a specific TCP port. The Startd establishes a new outgoing TCP connection to the Schedd, completing the CCB-assisted *connection reversing* procedure.

The default GlideinWMS deployment embeds the CCB daemon in each of the child collectors. When running at 150,000 parallel jobs, we found this dual-role for the child collectors caused sufficient network load that the system became unresponsive. We patched our GlideinWMS setup to move the CCB functionality to a separate host. As each Startd maintains separate TCP connections for job updates and CCB functionality, this halved the number of connections on the central hosts.

4.3. High Availability

The **HA** mode consists on running two parallel central managers in two physically separate nodes; a heartbeat / fencing protocol is used to keep only one *Negotiator* active at any given time.

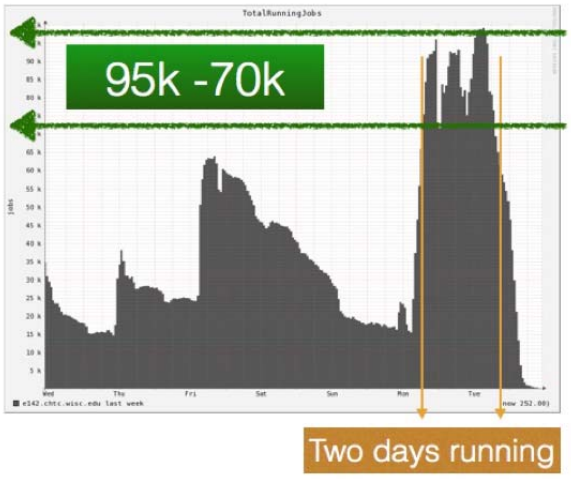


Figure 6. A single HTCondor reaching 95k parallel running jobs for two days. The tests were performed in a 128GB memory machine and having HTCondor spool and log directories mounted on ssd disks. The jobs were submitted at random intervals with average job length of 8 hours.

To perform this coordination, a **HA** *daemon* is run on both hosts. If it detects the primary one to have gone off it kicks in the secondary *Negotiator*. As mentioned in Section 2 High Availability will be a key factor in CMS operation for LHC Run II. Hence the requirement to include this recovery scenarios at higher scales in our tests. Due to last sections outcome on removing the CCB tree out of the same node as the Collectors, a new single point of failure was added. This was later fixed by adding a secondary CCB node and having all the *startd* registering to both. In this operational mode our final and recommended testbed for GlideinWMS deployments can be seen in Figure 4.

Never before the **HA** mode has never been tested before to fully work when running at the scales mentioned in this paper. The highest scale at which it has been known to run is at $O(30k)$ running jobs[16]. But we tested and probed that given this topology and the recommend versions and configuration changes mentioned in the section before, $O(200k)$ parallel running jobs can be achieved while maintaining resiliency on the system. We performed the test of bringing down a single CCB or a single Collector while running $O(120k)$ running jobs for more than 12 hours, and discovered the jobs were able to reconnect as seen in Figure 5.

4.4. Outgoing long-lived TCP streams

Although our initial topology depicted in Figure 4 consisting of dual redundant Collectors and dual redundant CCB servers yielded good results in terms of scalability and fail-over resiliency, it also doubled the number of outgoing long-lived TCP streams per execution node, potentially causing problems at large site border firewalls, NATs, or routers. Our initial topology resulting in 11 TCP streams from the worker node to the outside: 2 connecting the Startd to each Collector, 2 for Master to each Collector, 2 for Startd to each CCB server, 2 for Master to each CCB server, plus when a job is running, 2 more for Starter to each CCB server and 1 more for the Schedd to the Startd.

To reduce the number of required TCP connections, two configuration changes were added. First the *SharedPort* daemon, which previously was configured to run at just the submit nodes, was enabled on the execute nodes as well to handle CCB connections for all daemons on the execute node with just 2 streams. Second the connection from the *Master* daemon in the execution node to the collector(s) was shut down. The aftermath was that only 5 long-lived TCP connections were needed from the execution node to the outside: 2 for Startd to each Collector, 2 for SharedPort to each CCB server, and 1 more for the Schedd to the Startd.

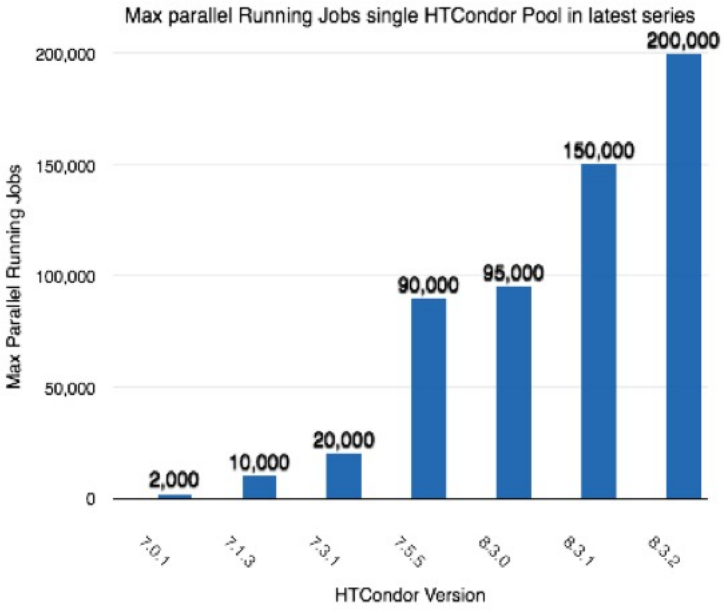


Figure 7. The maximum running parallel jobs for some HTCondor versions. This is a combination of results from [13], [8] and Figure 3. Bear in mind this tests were performed by different people with different hardware and are presented here as a vision of the expanded capabilities of the software.

4.5. Scheduler

The *Schedd* piece of HTCondor architecture that manages the submit machine is suitable for horizontally scaling. It is common to have several of them serving a single pool, since HTCondor provides for correct accountability for jobs submitted by the same user from different schedulers. However as explained in Section 2 the until now results would not entirely satisfy the needs on a 200k *Startd* pool. By leveraging access to the large testbed created as part of this work, throughout the past year we have been able to identify and remedy several bottlenecks in the implementation of the Schedd. A few examples include:

- **Batched Resource Requests** Previously the Schedd would send one resource request to the Negotiator, and then await a response before sending another request. While acceptable on a LAN, the latency of this many network turnarounds on the WAN increased negotiation times significantly; for instance, processing 1000 resources requests on a LAN took 30 seconds, but over 1153 seconds on a WAN w/ 100ms latency. By batching requests, negotiation times on the WAN dropped by $\tilde{7}0\%$.
- **Remove file locking** The Schedd used to obtain a write file lock when writing job and audit events to disk. Because Linux does not schedule lock requests but simply grants them in random order, lock starvation could cause the Schedd to become blocked for minutes at a time when 75k+ jobs were running. By leveraging the POSIX guarantee that *write()* system calls on a file opened in append mode will occur atomically, we removed several instances of file locking from the Schedd.
- **Reduce incoming TCP connections** To handle the case where either a submit or execute node disappears, each claimed execute node sends a keep-alive lease message to the Schedd every 20 minutes. On a Schedd managing 75k+ jobs, this meant 60+ TCP connections to accept, authenticate, and decrypt each second. We removed this Schedd burden by utilizing the TCP's stack's KEEP_ALIVE socket option on the long-lived socket between the Shadow and Starter process.

As a result of the above and numerous other scalability improvements added to HTCondor over the past few years, we were able to manage to get in a stable fashion a single Schedd instance to run $O(90k)$ parallel running jobs as seen in Figure 6.

5. Conclusions

The OSG Software team, in conjunction with HTCondor and GlideinWMS development teams have collaborated to push the scalability limits of a single HTCondor pool. Achieving a pool size of 200,000 execution nodes has required continuous, iterative work (as illustrated by Figure 7) by all participants. This allows HTCondor to continue to deliver functionality and resiliency to match the scientific needs of its stakeholders - CMS in particular.

Acknowledgments

This work is supported in part by the National Science Foundation through awards PHY-1148698 and ACI-1321762. Many thanks to the FNAL CMS T1, UCSD CMS T2 and UW Madison CHTC for the hardware to perform the tests as well as the numerous sites on which the test ran.

References

[1] Thain D, Tannenbaum T and Livny M 2005 Distributed computing in practice: the condor experience. vol 17 pp 323–356

[2] Sfiligoi I, Bradley D C, Holzman B, Mhashilkar P, Padhi S and Würthwein F 2009 The pilot way to grid resources using glideinwms vol 2 (Los Alamitos, CA, USA: IEEE Computer Society) pp 428–432 ISBN 978-0-7695-3507-4

[3] Raman R, Livny M and Solomon M 1998 Matchmaking: Distributed resource management for high throughput computing *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)* (Chicago, IL)

[4] Sfiligoi I, Würthwein F, Andrews W, Dost J M, MacNeill I, McCrea A, Sheripon E and Murphy C W 2011 Operating a production pilot factory serving several scientific domains vol 331 p 072031 URL `http://stacks.iop.org/1742-6596/331/i=7/a=072031`

[5] CERN 2014 *5th Evian Workshop on LHC beam operation* (Geneva: CERN) organisers: Lamont, M; Meddahi, M; Goddard, B URL `https://cds.cern.ch/record/1968515`

[6] Letts J, Balcas J, Belfort S, Bockelman B P, Wüerthwein F, KHAN F A, Majewsky K, MASCHERONI M, Mason D, Mcrcrea A, Santos M D S, Sfiligoi I, Linares L and Gutsche O 2015 *Journal of Physics: Conference Series* CHEP 2015

[7] Silberstein M, Kliot G, Sharov A, Schuster A and Livny M 2006 Short paper: Materializing highly available grids *High Performance Distributed Computing, 2006 15th IEEE International Symposium on* pp 321–323 ISSN 1082-8907

[8] Bradley D, Clair T S, Farrellee M, Guo Z, Livny M, Sfiligoi I and Tannenbaum T 2011 An update on the scalability limits of the condor batch system vol 331 p 062002 URL `http://stacks.iop.org/1742-6596/331/i=6/a=062002`

[9] Amazon Web Services I 2015 Amazon web services pricing URL `http://aws.amazon.com/es/ec2/pricing/`

[10] Sfiligoi I, Würthwein F and Theissen C 2010 Using condor glideins for distributed testing of network facing services *Computational Science and Optimization (CSO), 2010 Third International Joint Conference on* vol 2 pp 327–331

[11] Altunay M, Avery P, Blackburn K, Bockelman B, Ernst M, Fraser D, Quick R, Gardner R, Goasguen S, Levshina T, Livny M, McGee J, Olson D, Pordes R, Potekhin M, Rana A, Roy A, Sehgal C, Sfiligoi I and Würthwein F 2011 A science driven production cyberinfrastructure???the open science grid vol 9 (Springer Netherlands) pp 201–218 ISSN 1570-7873 URL `http://dx.doi.org/10.1007/s10723-010-9176-6`

[12] Kranzlmüller D, de Lucas J and Öster P 2010 The european grid initiative (egi) *Remote Instrumentation and Virtual Laboratories* ed Davoli F, Meyer N, Pugliese R and Zappatore S (Springer US) pp 61–66 ISBN 978-1-4419-5595-1 URL `http://dx.doi.org/10.1007/978-1-4419-5597-5_6`

[13] Bradley D, Sfiligoi I, Padhi S, Frey J and Tannenbaum T 2010 Scalability and interoperability within glideinwms vol 219 p 062036 URL `http://stacks.iop.org/1742-6596/219/i=6/a=062036`

[14] Butler R, Welch V, Engert D, Foster I, Tuecke S, Volmer J and Kesselman C 2000 A national-scale authentication infrastructure vol 33 pp 60–66 URL `http://dx.doi.org/10.1109/2.889094`

[15] Tannenbaum T 2010 What's new in condor? whats coming up? slides of talk given at Condor Week 2010, April 12-April 16, 2010, University of Wisconsin, Madison, WI URL `http://research.cs.wisc.edu/htcondor/CondorWeek2010/condor-presentations/tannenba_roadmap_2010.pdf`

[16] Sfiligoi I, Letts J, Belforte S, McCrea A, Larson K, Zvada M, Holzman B, Mhashilkar P, Bradley D C, Santos M D S, Fanzago F, Gutsche O, Martin T and Würthwein F 2014 Cms experience of running glideinwms in high availability mode vol 513 p 032086 URL `http://stacks.iop.org/1742-6596/513/i=3/a=032086`