FERMILAB-CONF-10-258-CD

Scalability and interoperability within glideinWMS

D Bradley[1], I Sfiligoi[2], S Padhi[3], J Frey[1] and T Tannenbaum[1]

[1]University of Wisconsin, Madison, WI, USA
[2]Fermilab, Batavia, IL, USA
[3]University of California, San Diego, La Jolla, CA, USA

E-mail: dan@hep.wisc.edu, sfiligio@fnal.gov, Sanjay.Padhi@cern.ch, jfrey@cs.wisc.edu, tannenba@cs.wisc.edu

Abstract.

Physicists have access to thousands of CPUs in grid federations such as OSG and EGEE. With the start-up of the LHC, it is essential for individuals or groups of users to wrap together available resources from multiple sites across multiple grids under a higher user-controlled layer in order to provide a homogeneous pool of available resources. One such system is glideinWMS, which is based on the Condor batch system. A general discussion of glideinWMS can be found elsewhere. Here, we focus on recent advances in extending its reach: scalability and integration of heterogeneous compute elements. We demonstrate that the new developments exceed the design goal of over 10,000 simultaneous running jobs under a single Condor schedd, using strong security protocols across global networks, and sustaining a steady-state job completion rate of a few Hz. We also show interoperability across heterogeneous computing elements achieved using client-side methods. We discuss this technique and the challenges in direct access to NorduGrid and CREAM compute elements, in addition to Globus based systems.

1. Introduction

glideinWMS [2] is a workload management system used by the CMS LHC experiment. In a nutshell, its purpose is to dynamically gather grid resources and present them to the users like a dedicated Condor batch system. This provides a number of advantages to the users over direct submission to the grid. For example, rather than sending the user's jobs into remote queues, which generally have unpredictable wait times, the user's jobs remain in the central submit queue until matchmaking finds a specific worker node that has been allocated to glideinWMS and which is therefore available to begin execution immediately. This is known as "late binding". glideinWMS is a specific example of a general technique known as "pilot job" submission, and late binding is one of the main attractions of such systems [4].

In the context of this paper, we shall assume that an instance of the glideinWMS system is being used by a group of users belonging to a Virtual Organization such as the CMS VO. In this case, just like in a normal Condor pool, glideinWMS provides the ability to set system-wide relative user priorities and to change them as needs of the VO change.

A final example of an advantage of glideinWMS over direct grid submission is improved monitoring, error handling and troubleshooting [5] of the running of the user's job, an essential ingredient for productivity in a distributed system such as the grid, which crosses networks, administrative domains, and organizational boundaries.

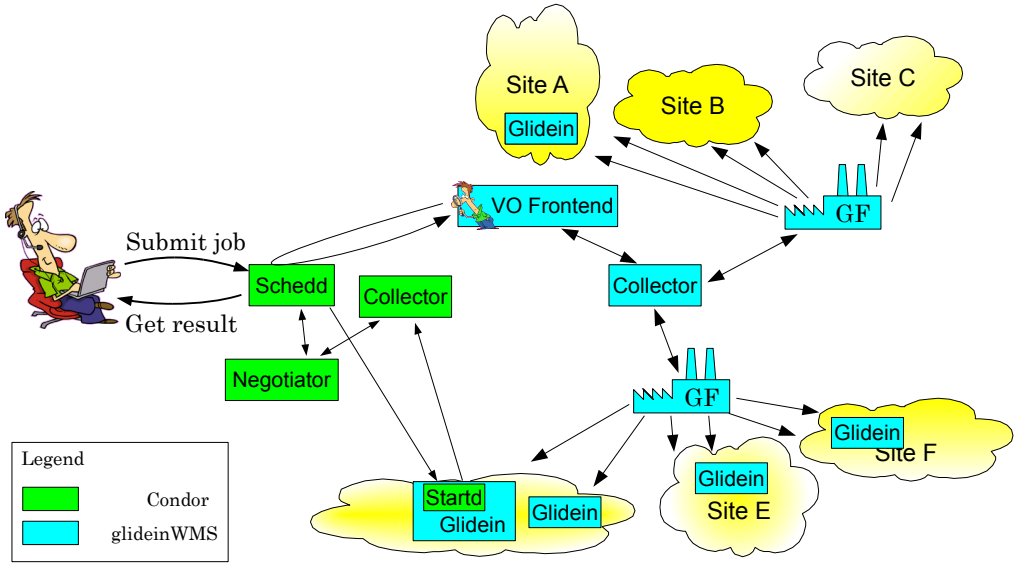

Figure 1. Overview of glideinWMS system.

The operation of glideinWMS is illustrated in Figure 1. The main idea is that when demand for more resources is sensed by the VO Frontend, Condor job execution daemons (aka glidein pilots) are submitted to the grid by the Glidein Factory. This is known as "gliding in" to the grid. When they begin running, they "phone home" and join the glideinWMS Condor pool. Then they are available to run user jobs through the normal Condor mechanisms. Once demand for resources decreases to the point where some nodes have no work to do, the Condor job execution daemons on those idle nodes exit and the resources that were allocated to glideinWMS at the grid site are released for use by others.

A brief description of the Condor components will help understand the rest of this paper. The responsibility of the Condor collector is to receive descriptions (called ClassAds) of all of the other daemons in the Condor pool. The collector also responds to queries from other Condor components wishing to find out about the existence and attributes of other parts of the pool. The Condor schedd maintains a job queue. The Condor startd runs on the worker nodes and executes jobs that it receives from the schedd. The startd is therefore the main component of a glidein worker. When we say that the glidein "phones home" we mean that the startd on a worker node sends a description of itself to the central collector. The Condor negotiator does matchmaking between jobs and startds, verifying that they are compatible and allocating resources according to the relative priorities of the users.

2. Scalability of glideinWMS

Because glideinWMS attempts to turn the Grid into one centralized Condor pool as big as the input workflow(s), scalability is an important concern. One could always scale by using multiple instances of glideinWMS, but this would detract from one of the main selling points of the glideinWMS approach, which is to unify the Grid computing resources under a single batch-system interface. Therefore a major effort has been made to understand and extend the limits of scalability in glideinWMS.

Table 1. Network latency and authentication cost in seconds for cross-Atlantic (WAN) and a local network for comparison. Only time spent blocking on the network is counted in the authentication cost, so the conversion of some client-side operations from being blocking to asynchronous helped reduce the cost in Condor 7.1.3.

	WAN	loopback
ping time	0.15	0.00006
bare GSI authentication (client)	0.43	0.15
bare GSI authentication (server)	0.74	0.15
GSI + condor sec v7.0.1 (client)	1.4	0.16
GSI + condor sec v7.0.1 (server)	1.2	0.16
GSI + condor sec v7.1.3 (client)	1.0	0.15
GSI + condor sec v7.1.3 (server)	1.2	0.16

2.1. The Cost of Authentication

Experience with glideinWMS in LHC Common Computing Readiness Challenge 08 (CCRC-08 [3]) exercise taught us that the cost of strong authentication between Condor components increases considerably with added network latency and this has a detrimental impact on scalability. Therefore, scalability tests that we had conducted on the Fermilab LAN gave us an overly optimistic expectation for scalability over the WAN, especially over large distances such as between the US and Europe. This observation prompted us to conduct extensive tests between Europe and locations in the US. (Alternatively, network latency could have been artificially introduced, but we found it convenient to simply use the Atlantic Ocean.) We set up the glideinWMS front-end scheduler (schedd) and collectors in Padova, Italy. The glidein worker daemons (startds) were submitted to test sites in the US, including Fermilab, University of Wisconsin, University of Nebraska, and University of California San Diego.

The first question we investigated was, *why does network latency make authentication so expensive in Condor?* The answer is that the authentication protocol is implemented as a series of messages that are exchanged between the two sides, and while one side is waiting for the other side to respond, the entire process blocks. This hurts scalability, because it limits the rate at which central Condor components such as the collector and the schedd can authenticate new glidein daemons that they have never talked to before. Existing daemons that they have already talked to are not much of a problem, because after authentication, a security session is cached for a configurable amount of time (e.g. one day).

Table 1 shows the authentication cost measured between the Padova and Wisconsin test sites compared to tests on a loopback network. The cost is in seconds of time in which the Condor process is devoted solely to a single authentication task. For glideinWMS purposes, we are not interested in the actual latency of authentication measured in time from start to stop but rather in the cost to throughput in the Condor daemon's event handler. Since there is only one thread of execution in current versions of Condor, any time spent on the authentication task is time that cannot be used for any other purpose, but time spent doing other things while asynchronous authentication operations complete is not counted as a cost.

The first thing to notice from the table is that there is a base CPU cost in doing GSI (0.15s) that has nothing to do with network latency. This is visible in the looback test. Our test machine in this case was a 1.5GHz Pentium 4. On faster hardware, the CPU time for GSI authentication will be less, but it is still a scalability concern.

The next thing to notice is that the cost of latency comes from both the bare GSI protocol and the Condor security protocol that is wrapped around it. With a total cost of over a second, it is easy to see why it was impossible to achieve even 1Hz in the number of new glideins entering

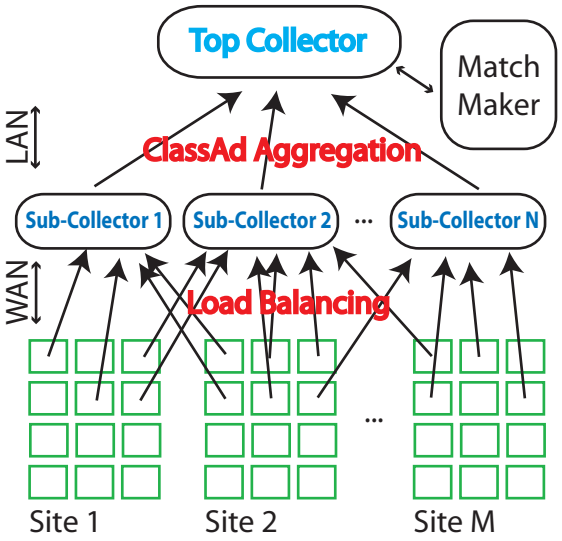

Figure 2. Multiple collectors may be used to handle the WAN authentication workload. In our largest scale tests where the collectors were also acting as authenticated Condor Connection Broker (CCB) server, 70 sub-collectors on one machine were able to handle a pool of over 20,000 glideins.

the pool and being claimed for use. If 3600 glideins started running at around the same time, the last one to be successfully authenticated to join the pool would have to wait around doing nothing for over an hour! This would result in unacceptably low efficiency when expanding the pool and when the scale of operation requires replacing glideins at anything approaching 1Hz or beyond.

Our first attempt to address this problem may be seen in the reduction of about 30% in the client-side authentication cost in Condor 7.1.3. This was achieved by converting parts of the Condor security code to using non-blocking event-driven network operations. This sort of refactoring of the code can become rather laborious in cases of deeply nested function calls, and even if we were to remove all of the blocking operations, the CPU cost of authentication would not have been addressed. We therefore looked for alternative solutions to the still high cost of authentication in 7.1.3.

The longer term solution that we chose was to introduce a limited form of threading in Condor. This work is in progress. The shorter term solution that we chose was two-pronged: use multiple processes in the collector and optimize away WAN authentication in the schedd. These steps are done (Condor 7.2) and resulted in significant gains in scalability.

2.1.1. Two-tier Collector To make it possible to use multiple collector processes while still having one global view of the pool, we made a simple extension to the existing ability to forward ClassAds from one collector to another. We configure N collectors to each forward ClassAds to one top-level collector. By making Condor forward a little more information than was previously done, it became possible to use the top-level collector for matchmaking purposes (i.e. the top-level collector is paired with a negotiator that then has access to the state of the whole pool).

The glidein daemons are then configured to advertise themselves to just one of the N sub-collectors. In this way, the authentication workload is spread across N processes rather than just one, as depicted in Figure 2. The top-level collector only has to authenticate the N sub-collectors once and then they communicate efficiently using cached security sessions. Since the authentication workload is largely just a matter of blocking on the network, glidein creation rates at our target of about 1Hz can be sustained comfortably with a single CPU. However, since the N sub-collectors run in different processes, they can take advantage of multiple cores or even multiple machines if a higher rate is needed. We have only tested at up to

1.8 new glideins/s (achieved with 70 sub-collectors on a single machine). This corresponds to about 16 authentications/s when taking into account the two extra monitoring daemons that we happened to be running in the glidein, and the fact that the collector was also serving as an authenticated CCB server (discussed in more detail in section 2.3). Although further optimization of the number of authentications per glidein is certainly possible, and increasing the use of asynchronous operations in the collector is also possible, this is not a major scalability concern: adding more sub-collectors is cheap.

The following rule-of-thumb may be used to determine the required number of sub-collectors to handle the latency in authentication in Condor 7.2 and 7.3. It simply assumes that the number of required collectors scales linearly with the rate of authentication and the network latency.

$$N_{\text{sub-collectors}} = k * RTT * glideinRate * (glideinDaemons + isCCB * (glideinDaemons + glideinStarters))$$

where k is an experimentally determined constant, RTT is the round-trip-time between the collectors and the glideins, *glideinRate* is the peak desired rate of new glideins joining the pool, *glideinDaemons* is the number of Condor daemons that publish to the collector in each glidein instance (4 in our case due to 2 extra monitoring daemons), *isCCB* is 1 if the collector is also acting as a CCB server (0 otherwise), and *glideinStarters* is the number of starter processes expected to be running in the glidein (usually 1, for one running job). The lower bound for k has not been thoroughly probed in our setup, but we have experienced good performance for values of k as low as 30. Based on our measurement of authentication cost, $k = 30$ would correspond to the collector spending only between 2% and 3% of its time authenticating glideins on average. It seems likely that smaller values of k could give reasonable performance, but whether one has 40 or 70 sub-collectors is not a very important difference, since there is only a small fixed cost per collector ($\sim$1MB RAM); the rest of the collector cost scales with the number of glideins in the pool and is split across however many sub-collectors there are.

The primary cost of deploying a two-tier collector scheme in place of a single collector is that it doubles the amount of memory required, since each ClassAd is stored twice, once in a sub-collector and once in the top collector. This double-storage could be eliminated by making the sub-collectors forward ClassAds to the top collector without storing a copy themselves. However, the cost is not very large, and the extra information is useful in debugging. In our tests, the memory required to support the two-tier collector was under 40kB per registered daemon (less than 2GB for 13000 glideins).

2.1.2. Matchmaking and Security Sessions Having an adequate solution for the collector allowed us to observe the next bottleneck, which was the schedd. In order to use a newly registered glidein, the schedd must connect to it and send the job. This connection needs to be authenticated, so here again the effect of latency created a severe limit on the rate at which the schedd could begin using new glideins. Using multiple schedds does work, in the same way that using multiple collectors works, but aggregation of job queues is not as simple a task as aggregation of collector contents. This added layer of complexity is not desirable if it can be avoided.

Fortunately, the authentication problem can be completely removed from the schedd, at least if one is willing to accept a small change to the security model. The central manager (top collector plus negotiator) authenticates and authorizes all members of the glideinWMS Condor pool. This includes the startds and the schedd. The members of the pool also authenticate the central manager, since GSI is mutual. Therefore, if the startds and schedd are willing to trust the central manager in the same way that they trust each other when communicating about a job, the act of matchmaking can be used to establish a security session between the startd and schedd without them ever directly authenticating each other beforehand. By “security session”

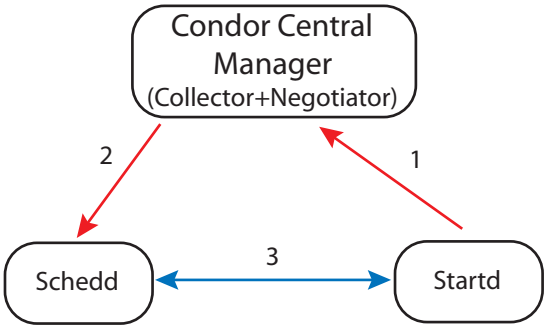

Figure 3. Integrated matchmaking and security session management optimizes secure communication between the schedd and startd, boosting scalability of the schedd, which has to communicate with many startds. The security session, which is good for the duration of the match, is established by passing a shared secret from the startd to the central manager and then to the schedd that is matched to the startd.

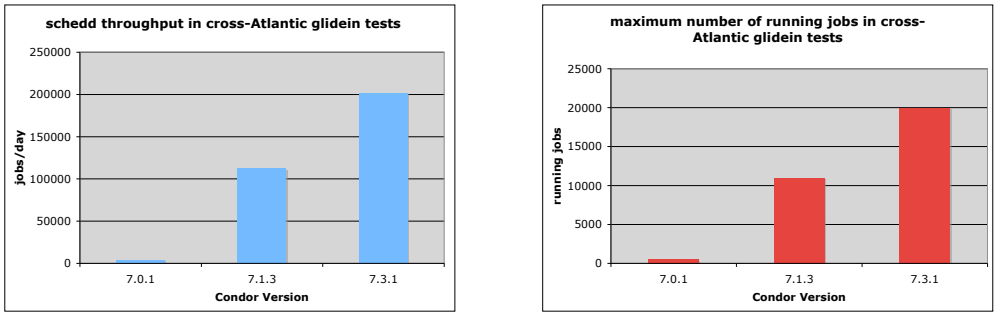

Figure 4. Number of running jobs during the cross-Atlantic glideinWMS tests. The average job runtime in these tests was 3h. Condor 7.0 suffered from authentication latency. Condor 7.1 addressed this. Condor 7.3 optimized memory usage and switched from GCB to CCB. Note that the 7.3 test also switched from a 32-bit to a 64-bit Linux kernel to handle the larger number of concurrent running jobs.

we mean a securely established shared secret that can be efficiently and securely verified by both parties in future communication and a mapping of that shared secret to an identity that is authorized to do specific operations. A simple illustration of the communication flow is shown in Figure 3.

We added support for this mode of operation to Condor in 7.1.3. It is enabled by using the SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION option. The effect was to avoid all the blocking on round-trips and CPU overhead involved in GSI authentication between the schedd and startds. The resulting improvement in throughput is shown in Figure 4.

2.2. Memory Usage

After dealing with the authentication bottleneck, the limiting factor to running more jobs under a single schedd was primarily memory usage. In Condor 7.3.0, we reduced memory usage per running job from 1.1MB to 400kB. This is memory used by the condor_shadow process that runs under the schedd. We also found that it was important to use a 64-bit Linux kernel in order to exceed 11k running jobs, because the 32-bit kernel would run out of memory for some data structures, even though there was plenty of free "HighMem" available. Figure 4 shows the observed improvement in sustained number of running jobs.

2.3. Disk Latency

In addition to memory usage, the final improvement in scalability was made possible by a reduction in the number of transactions required to prepare a running job in the schedd. The schedd keeps a database log so that the job queue can be reliably reconstructed after sudden loss of power or other similar events. We found that the expansion of $$ macros in job ClassAds was unnecessarily expensive because each individual macro was expanded in its own transaction. In Condor 7.3.1, these are now bundled together to reduce time spent waiting for data to be safely synched to the disk.

2.4. Port Usage

In our tests, after all of the other optimizations were made, the limiting factor in Condor 7.3.1 was network port usage. With more than 23k running jobs, the schedd machine began to run out of network ports. There are only 65k TCP ports per IP address. Each running job consumes between 2 and 3 during its life. The rate at which ports are opened and closed leaves a fraction of ports in the TIME_WAIT state at any given time.

Although we have not yet made any attempt to reduce port usage for running jobs, the total number of ports required to support a glidein Condor pool has decreased quite a bit in Condor 7.3 when we introduced CCB. This was more of a pleasant side effect than a design goal. CCB is the Condor Connection Broker and we are now using it in place of GCB, the Generic Connection Broker. The primary reason for introducing CCB was to make the system more robust (CCB registrations are fault tolerant) and secure (CCB registrations are fully authenticated) [6]. However, it also happens that the CCB server uses a single port whereas the GCB server uses one port per registered daemon. Therefore, to support a glidein pool of 20k glideins with 5 registered daemons per glidein as in our case, GCB requires 100k ports while CCB requires just one per CCB collector (70 in our test). CCB therefore simplifies deployment of the glideinWMS system, because more services can be squeezed onto fewer machines due to the lower port requirements. We have tested with CCB, the central manager, and the schedd all running happily on the same machine with 10k running jobs.

2.5. glideinWMS Pilot Factory Scalability

The job of the pilot glideinWMS pilot factory is to manage the submission of glideins to grid sites. In the cross-Atlantic scale tests, up to 7 gt2 gatekeepers were targeted, using up to 13 Condor-G submission instances (which will be discussed in the following section). In this configuration, glidein startup rates of up to 1.8Hz were demonstrated. However, the sustained glidein completion rate peaked at 0.5Hz, due to throughput limitations in Condor-G as of version 7.3.1. Therefore, in order to support a job completion rate of 200,000 jobs/day (2.3Hz), it was necessary to either increase the number of Condor-G instances, or site gatekeepers, or increase the glidein lifespan to be sufficiently long so that each glidein could be reused for multiple jobs. The latter is a reasonable thing to do as long as the glidein lifespan is less than the maximum allowed time for running jobs at the grid sites: 24h is a typical upper limit. With glidein lifespans of 9h, at most 16k simultaneously running glideins were sustained in our tests. With glidein lifespans of 18h, 23k simultaneously running glideins were sustained and scaling higher was limited by factors outside of the glidein factory component (port exhaustion in the Condor schedd machine managing the running user jobs).

Another important scaling factor is how many sites could be simultaneously harnessed. We did not conduct controlled tests of this, but we did learn some things during the already mentioned LHC Common Computing Readiness Challenge 08. This also gave us experience with the system in a more realistic usage environment.

All of the glideinWMS components were installed in the US, and the glideins were being sent to O(100) production Grid sites in the US and in Europe. The demand placed on the Condor

schedd and collector was much smaller than in the other tests, with an average of 1k running and 10k queued jobs, with peaks of 3k running and 50k queued. The system was being used for more than a month.

The gfactory, Condor-G and the glideinWMS dashboard were hosted on one node, the frontend shared a node with a GCB, the schedd was installed on a dedicated node, and a tree of collectors was using another dedicated node. The dashboard and the frontend never reached a scalability limit. Instead, the gfactory did.

The gfactory ran into IO transaction limits when serving more than approximately 50 Grid sites. The load comes mostly from the administrative monitoring subsystem of the gfactory. By disabling part of the monitoring, an option available since glideinWMS v1_4, the gfactory was able to handle the available CMS Tier-2 grid sites, across Europe and US, but the load on the system was still uncomfortably high. By moving the factory monitoring files to TMPFS (a memory-backed filesystem), this problem was largely overcome. These files would be lost in the event of a system crash or power loss, but since they are purely for monitoring purposes, an occasional backup is usually all that is needed. Using all the CMS Tier-2 sites, the system load factor is $\sim$2. To make the files fit into a reasonable amount of RAM, the number of plots had to be restricted. This monitoring scalability problem has been addressed in glideinWMS version 2, after the time of our testing.

3. Interoperability of glideinWMS

Another important aspect of glideinWMS is its ability to interoperate with different types of grids. The basic idea behind a glidein Condor pool is that somehow the glidein pilot is executed on worker nodes. The mechanism by which that happens is not relevant to the functioning of the glidein Condor pool, as long as the pilot's startd daemon is able to phone home and join the pool. Therefore, there is at least the potential for great flexibility in what types of grids or compute clouds glideinWMS is able to integrate.

In practice, the mechanism used by glideinWMS to distribute pilots is Condor's "grid universe", also known as Condor-G. Condor-G jobs are managed in a job queue by the schedd, just like jobs intended to run in a Condor pool. However, instead of being sent to a startd to run, the jobs are sent to other grid sites. The grid universe supports a growing number of grid protocols, including Globus gt2 and gt4, as well as Condor-to-Condor, EC2, and NorduGrid. Support for different grid types is accomplished through protocol-specific modules called GAHPs (Grid Ascii Helper Programs) plus a small amount of logic in the Condor gridmanager specific to the use of each GAHP. From the glideinWMS system's point of view, however, all that is necessary is to configure the glidein factory to set the appropriate grid type in the job description for pilot jobs destined for a given site.

In the CMS CCRC-08 exercise, glideinWMS successfully integrated over 4000 CPUs from more than 40 sites across EGEE, NorduGrid, and OSG. This was the first time that the NorduGrid ARC interface was used in CMS. The other sites were accessed via the gt2 protocol.

Recent developments to Condor-G have focussed on adding support for the CREAM CE [1]. Figure 5 shows our test results in trial runs across several sites accessed via CREAM. In submission of glideins to CREAM sites, we observed a 25% failure rate, mostly due to proxy renewal/delegation. We reported this problem to the CREAM developers. Fortunately, the user jobs are not affected, because the glidein fails before "calling home." User jobs are only affected by glideins that succeed in running, pass initial sanity checks, and join the Condor pool.

4. Related Work

A detailed comparison of glideinWMS to the current state of other workload management systems is beyond the scope of this paper. Some prior work on this subject made quantitative and qualitative comparisons of Condor-G, ReSS, gliteWMS, and glideinWMS [7]. A more recent

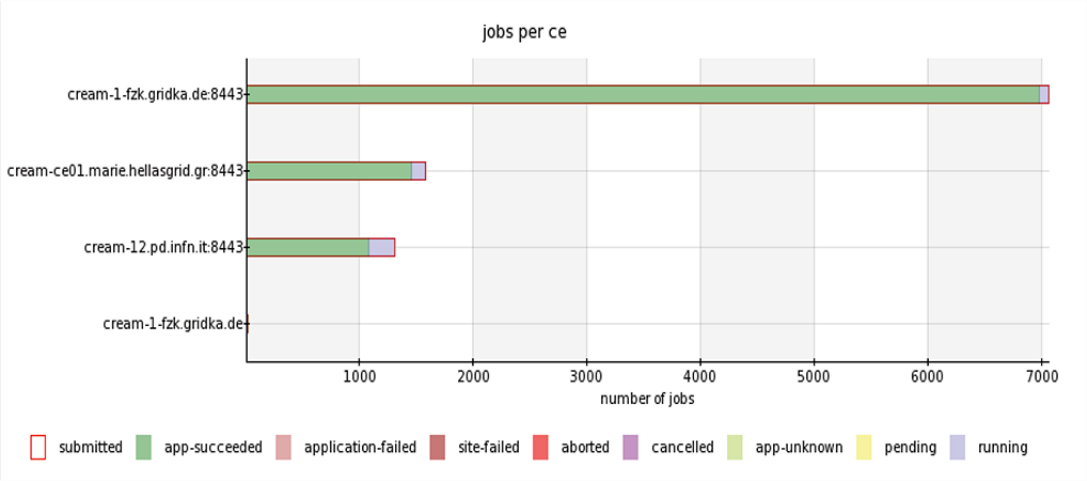

Figure 5. CMS tests of glideinWMS incorporating resources from CREAM sites.

stress test of gliteWMS demonstrated a sustained throughput of 90k jobs/day [8]. PanDA, a pilot-based workload management system used by the ATLAS experiment, routinely processes 70k jobs/day and has achieved 35k simultaneously running jobs [9]. Both gliteWMS and PanDA can use Condor-G as the underlying mechanism for submission to grid sites.

5. Conclusion

Significant progress has been made in making glideinWMS attractive for use by CMS and others who wish to harness grid resources through a powerful and uniform interface. The system scales to over 20k running jobs with a throughput of over 200k jobs/day while securely traversing global networks. In addition, it has held true to the promise of flexibility by incorporating new types of grids.

Acknowledgments

This work was supported by US DOE contract No. DE-AC02-07CH11359 and the U.S. National Science Foundation grants PHY-0427113 (RACE) and PHY-0533280 (DISUN).

References

[1] Aiftimiei C, et al., 2009 Design and implementation of the gLite CREAM job management service, INFN Technical Note INFN/TC_09/3
[2] Sfiligoi I, 2008 glideinWMS-A generic pilot-based Workload Management System, *Journal of Physics: Conference Series* **119** 062044
[3] Belforte S, Fanfani A, Fisk I, Flix J, Hemandez J, Klem J, Letts J, Magini N, Miccio V, Padhi S, Saiz P, Sciaba A, Würthwein F, 2008 The commissioning of CMS computing centres in the worldwide LHC computing Grid, N29-5, *Nuclear Science Symposium Conference Record*, NSS '08. IEEE
[4] Sfiligoi I, Bradley D, Holzman B, Mhashilkar P, Padhi S, Würthwein F 2009 The pilot way to Grid resources using glideinWMS *Preprint* CSIE.
[5] Sfiligoi I, Bradley D, Livny M 2009 Pseudo-interactive monitoring in distributed computing *Preprint* JPCS.
[6] Bradley D, 2009 The Condor Connection Broker *Condor Week 2009.*
[7] Holzman B, Sfiligoi I, 2007 A Quantitative Comparison Test of Workload Management Systems, CHEP 2007.
[8] Cecchi M, et al., 2009 The gLite Workload Management System *Advances in Grid and Pervasive Computing*, **5529** 256-268.
[9] P. Nilsson, 2008, The PanDA System in the ATLAS Experiment, ACAT '08, Italy, 2008.