JobQueue: A Computational Grid-wide Queuing System *

Dimitrios Katramatos[1], Marty Humphrey[1], Andrew Grimshaw[1], and Steve Chapin[2]

[1] Department of Computer Science, University of Virginia
Charlottesville, VA 22903, USA
{dk3x, humphrey, grimshaw}@cs.virginia.edu
http://www.cs.virginia.edu

[2] Department of Electrical Engineering and Computer Science, Syracuse University
Syracuse, NY 13244, USA
chapin@ecs.syr.edu
http://www.ecs.syr.edu

Abstract. In a Computational Grid, it is not easy to maintain grid-wide control over the number of executing jobs, as well as a global view of the status of submitted jobs, due to the heterogeneity in resource type, availability, and access policies. This paper describes the design and implementation of JobQueue, which is a Computational Grid-wide queuing system, or metaqueuing system, implemented in Legion. JobQueue is unique because of its ability to span multiple administrative domains. It can also be reconfigured dynamically along a number of dimensions and in the general case does not require special privileges to create, facilitating new flexibility for Grid users.

1 Introduction

Computational Grids can combine thousands of hosts from hundreds of administrative domains, connected by transnational and worldwide networks. A Computational Grid functions similarly to an electric power grid: it couples geographically distributed resources and offers consistent and inexpensive access to these resources irrespective of their physical location or access point [1]. In essence, a Computational Grid allows resources to be used as a single virtual and extremely powerful resource. From the perspective of the computational scientist, Computational Grids can provide the high-performance computing infrastructure necessary for dealing with modern challenging problems.

In the computational science community, sharing of resources is common. All computing systems have certain limits and exceeding these limits leads to low throughput - everybody waits while little or no work is being done. The use

* This work was partially supported by Logicon (for the DoD HPCMOD/PET program) DAHC 94-96-C-0008, NSF-NGS EIA-9974968, NSF-NPACI ASC-96-10920, and a grant from NASA-IPG.

of queuing techniques for jobs alleviates this problem. Queuing software uses certain criteria to ensure fair and efficient utilization of the computing resources it supervises. In the same sense, Computational Grids may offer tremendous computational power at much lower costs than conventional systems, but they too have capacity limits.

The ability to *throttle* the level of concurrent resource usage and otherwise schedule resources from a system-wide perspective is an important requirement of emerging Computational Grids such as NASA's Information Power Grid [7]. Without controlling the number of grid-wide simultaneously-executing jobs, there is no way to predict how long any particular job will execute because resources can become oversubscribed. Controlling resource usage from a global perspective is a particularly challenging aspect of a Computational Grid because of the existence of multiple administrative domains and the volume and complexity of heterogeneous resources.

A system-wide view of jobs includes the ability to determine if a particular job is waiting, executing, or has finished, when it was submitted, when it started/completed execution, and how long it has been running. It is also necessary and/or desirable to have the capability to kill jobs, restart them, and migrate them. Operating Systems and conventional queuing systems such as PBS [2], GRD [9], and LSF [8] offer a subset of these capabilities; however, providing such functionality for a Computational Grid requires solutions to a set of different, more complex problems.

Conventional queuing systems are used for controlling jobs within some administrative domain, e.g. a company, an organization, a research facility, etc. Within this domain the queuing system centralizes control and regulates the flow of jobs. A Computational Grid can span many administrative domains, each one with its own security restrictions, with its set of filesystems, and most importantly with or without it own queuing system. Consequently, a queuing system for a Grid must be able to deal with the individualities of all grid domains. One needs to imagine a grid-wide queuing system as being a step higher in the hierarchy: it accepts job submissions and sees to it that these jobs get executed, either by directly assigning them to resources or by handing them off to lower level resource managers. Furthermore, it monitors, directly or indirectly, execution progress. Consider a grid consisting of a large number of sites with subsets of them controlled by PBS, LSF, GRD, etc. and subsets without centralized control. A grid-wide queuing system will have to "talk" to each site's queuing manager to run and monitor a job there, however it will have to play the role of a local manager for sites without a local manager. Thus, a grid-wide queuing system needs to be able to utilize a whole range of different interfaces of "subordinate" queues and also offer regular queuing functionality. There has been a recent idea for a uniform command line interface for job submissions [10]. This idea can be easily realized when a grid-wide queuing system is present. When there is indeed an agreement on a uniform interface, grid-wide queues could be modified to "speak" the new language or more realistically be augmented with a translating module (see Fig. 1).

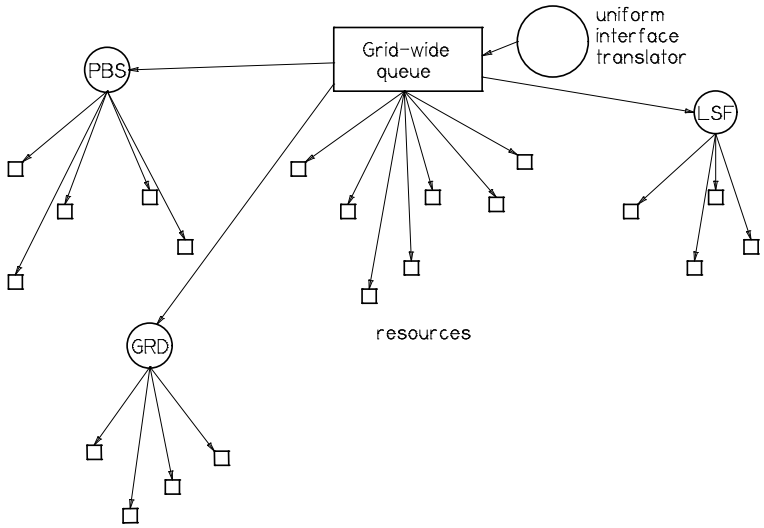

Fig. 1. Grid-wide queuing system hierarchy with uniform interface.

This paper describes the design and initial implementation of a job queuing system for the Legion object-based grid computing system [4]. The main component of the queuing system is a new object focused on job administration, the *JobQueue*. The JobQueue is a metaqueuing system that offers a job-throttling mechanism, a priority-based waiting queue, job control, and job monitoring features. In essence, the JobQueue is a high-level overseer that assures that the number of jobs simultaneously executing will never exceed the specified limits. It mediates between a user and the system's resources to accept job submissions, maintain order and fairness while at the same time provide the capability of prioritizing jobs, monitor job progress throughout execution, and provide job control and recovery capabilities.

2 Requirements

The JobQueue is not just another implementation of the typical functionality found in conventional queuing systems, although it does generally provide equivalent capabilities. There are three key requirements: the JobQueue must be able to span multiple administrative domains, it must have arbitrary scope, and cannot require special privileges for its operation.

2.1 Disjoint Administrative Domains

The JobQueue is a mechanism by which to overcome the boundaries of the domains participating in the grid system and regulate the "flow" of jobs grid-wide.

Whether or not the individual resources are themselves directly controlled by a queuing system such as PBS or LSF, the JobQueue facilitates a global control of resource usage. When a JobQueue spans multiple administrative domains, the resource providers can optionally decide an appropriate policy to implement in the JobQueue with regard to underlying resource usage. Without such a policy, the JobQueue implements a fair-share usage policy across all resources.

2.2 Arbitrary Scope

A JobQueue can be dynamically configured for particular users, particular resources, particular access control, or particular applications. Resources can be dynamically added to or removed from the resource pool of the grid system. This is to be expected in a wide-area distributed system; at any given time resource providers may limit or increase resource availability, malfunctions may happen, etc. It is desirable to have a system that can adapt to such changes. The JobQueue incorporates a feedback mechanism that essentially allows resources to ask to be given work when they're lightly loaded. Limitations to resource availability can be imposed by removing links to specific resources and thus preventing lower-level schedulers to select them. Known resource failures can also be handled the same way; if, however, the system attempts to utilize a failed resource and subsequently the job fails to execute, the JobQueue can later restart the failed job on a different resource.

2.3 No Special Privileges Requirement

Another unique feature of the JobQueue is that it can be started and set up by an ordinary user, provided that certain restrictions are in place. A possible scenario for this mode of operation is that a user wants to share a resource with some specific other users and also wants to restrict simultaneous access to a certain number of users at a time. Consider for example the case of a software package with a certain number of licenses. Conceivably, this user, as the owner of the resource, could utilize some other standard queuing system to impose these restrictions. This means he/she would have to go through the trouble of installing such a queuing system on a certain machine, on which he/she should have administrative rights. However, with the JobQueue this procedure is almost trivial, and no privileged access to a machine is required. The user can dynamically create a JobQueue instance and configure it to control the resource of interest and allow only a given number of simultaneous users. Requests to use the resource can be funneled to the private JobQueue from the "system" JobQueue, or can be sent directly to the private JobQueue. With the same ease, this instance can be destroyed when no more sharing is desired. Issues with regard to "non-system" JobQueues are further discussed in the next section.

3 Design Issues of a Metaqueuing System

The basic desired functionality of a metaqueuing system is to be able to enqueue job execution requests, initiate execution, maintain a pre-set number of grid-wide simultaneously executing jobs, and sporadically monitor progress and kill jobs. For this, the metaqueuing system can use the mechanisms a grid computing environment provides for utilizing resources spanning several different administrative domains. These mechanisms enable, for example, the execution of programs on any participating host system and the forwarding of data between participating filesystems. Additional useful functionality includes the capability to spot hung jobs and to restart jobs that have failed to execute for certain reasons. The remainder of this section discusses several issues in the design of such a queuing system.

3.1 Decentralized Control and Resource Coordination

With widely distributed resources it would be bad practice to centralize control and have one system component regulate the flow of jobs through the entire system. However, if multiple metaqueuing systems exist, it may become difficult to coordinate resource usage between the instances. There are many ways in which to distribute resource responsibility (see Fig. 2):

- each metaqueue maintains a list of resources it can schedule jobs on; resources can be added/removed dynamically to/from this list,
- each metaqueue is restricted to request lower-level services from system components that are already assigned to specific system areas and utilize resources only in those areas (Fig. 2 shows the Legion resource management infrastructure, Legion RMI, providing this functionality),
- instead of simply partitioning resources between a number of metaqueues, a hierarchy of instances can be used; each metaqueue instance can regulate resources directly or funnel requests to lower level metaqueue instances.

The first and second approaches provide control and monitoring within a partition of the grid system. The third approach is more complex but has the advantage of providing full grid-wide control and monitoring while allowing finer control of jobs, adapted to resource idiosyncrasies. In all cases however, the actual partitioning of the system's resources is a matter of policy; there can be partitioning according to physical location of resources, resource type, random, or some other policy. In an Computational Grid, it is inevitable that multiple metaqueuing systems must exist. The unresolved issues with any configuration is both how to direct user requests to the "best" metaqueue and how to ensure that the metaqueues collectively implement a well-defined policy. As the number of metaqueues increases, the actual policy implemented will be increasingly difficult to determine and/or verify.

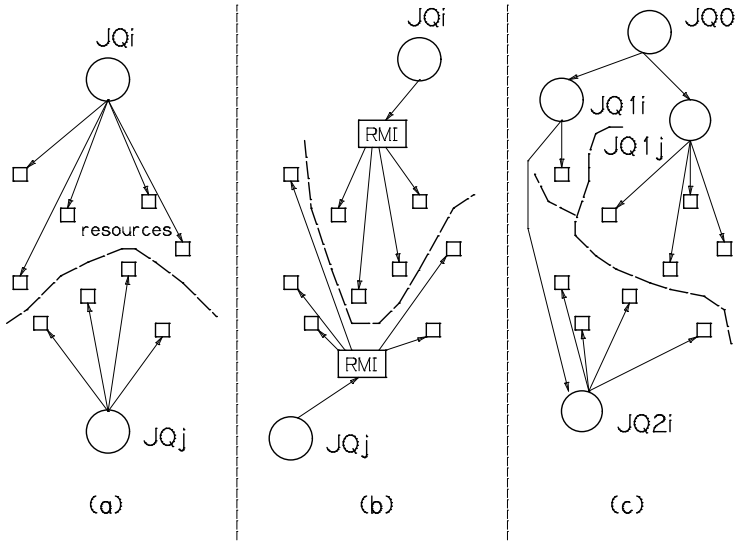

Fig. 2. Decentralized control and resource coordination: (a) direct partition, (b) indirect partition through Legion RMI, (c) hierarchy of JobQueues.

3.2 Privileged/Unprivileged Creation and Configuration

Certainly, allowing any user to create, setup, and use a metaqueue without any restrictions is not a good idea, as this would make any control over resource usage impossible. It seems prudent to make these operations privileged and available only to resource providers and administrators. A user is granted access to a metaqueuing system through the grid computing system's regular access control mechanisms. If a metaqueuing system can be created/setup by a user, then it should operate in "private" mode where several restrictions are set in place, allowing the particular user to utilize only specific resources and to only manage his/her own jobs. The particular issue is that allowing user- or group-specific queues offers new flexibility and control, but allowing too many metaqueues to be multiplexed over the same resources effectively eliminates the desired goal of control from the perspective of the resources themselves. Coordinating resource usage becomes very complex in this case as it is hard to control the number of jobs submitted by all users that run simultaneously in the overall system or even in one of its areas.

3.3 Predictability

Related to the previous issue is the predictability of the metaqueuing system. That is, after a job is submitted, how long will it be before it starts execution? In a priority-based queuing environment this time period depends on several

factors. If higher priority jobs get submitted all the time chances are that lower-priority jobs will face starvation. When all priorities are equal the scheduling is first-come first-served. Even so, the waiting period depends on the number of available running slots; if all slots are taken and thus the maximum number of jobs allowed to execute simultaneously has been reached, the next job will have to wait until one of the executing ones finishes.

It is possible to predict when a job will get started if at submission time additional data are given to the metaqueuing system. If for example the upper limit of execution time is known for the jobs already executing and for those waiting ahead of a specific job, the waiting time can be easily estimated. Another approach is by using advanced reservations. That is, a user can specify when a job is desired to run, and the Computational Grid-wide queuing system will try to make the necessary arrangements for it, if possible. Naturally, this scheme requires that the underlying infrastructure support advance reservations. In a Computational Grid, it should also be assumed that resources can be accessed via Grid means and via local means. Does predictability in a metaqueue require (temporarily) denying access via local means? It is unclear whether resource providers can justify this to their local users.

4 Implementation in Legion

Legion [4] is an object-oriented Grid Computing system developed at the University of Virginia. It is intended to connect many thousands or even millions of computers ranging from PCs to massively parallel supercomputers. Legion offers the necessary resource management infrastructure (Legion RMI) [3] for managing this vast amount of resources uniting machines of thousands of administrative domains into a single coherent system while at the same time supports site autonomy.

4.1 Job-handling in Legion

Legion supports two kinds of programs [5]:

- Legion-aware, which run as objects, and
- general, non Legion-aware.

In both cases programs need to get registered with the Legion system. The registration process creates managing objects and instructs the system which binaries to use for creating and executing program instances. However, a Legion-compatible program runs as a Legion object whereas for non-Legion programs—practically anything that executes on a regular operating system, even scripts—a different, more complex procedure needs to be followed. In essence, a special object is created to act as a proxy between the Legion system and the program binary. This is necessary as a generic binary lacks the special functionality needed to execute as a Legion object. With the use of proxy objects and a set of system tools it is possible to run almost any kind of program—without making

any changes to the program code—on any suitable resource participating in the computational grid.

4.2 New Functionality

The raw program execution mechanism of Legion covers the basic service of executing programs. However, using the mechanism independently, without a queuing system, creates the set of problems discussed in section 1, i.e.:

- there is no way of controlling the total number of jobs executing simultaneously grid-wide, as any user can start any number of running sessions at anytime,
- there is no global job handling and monitoring capability

The JobQueue provides the additional functionality needed to overcome these problems. While the procedure for registering executables with the Legion system remains exactly the same, in the new queuing system the run operation is broken down to three phases: (1) job submission, (2) job preparation, execution, and monitoring, and (3) job clean-up.

The main component of the queuing system, the JobQueue object, is responsible for the second phase. The first and third phases are done with the use of special software tools. When a job is submitted to the JobQueue, the object stores all necessary information for handling this job and returns a handle, a "ticket", which one can use for future references to the job. After submission, the JobQueue takes care of preparing, starting, and monitoring progress of this job.

Breaking down the run operation in the above mentioned phases and assigning the main portion of the operation to an administrative object makes possible to control the number of jobs that execute simultaneously in the Legion system. A submitted job will enter the second phase only when the number of total simultaneously executing jobs monitored by the JobQueue is less than the number of total jobs allowed, as specified by the Legion system administrator, i.e. when a running slot is available. If none is available, the job will have to wait - along with other submitted jobs - in a waiting queue. The order of waiting jobs in this queue is determined by priority number assigned to jobs by their owners and/or the administrator.

4.3 Internal JobQueue Operations

While a job is handled by the JobQueue, a user can communicate with the queue object and perform certain operations like inquiring about the job's status, changing its priority (only while waiting and within certain limits), killing it, or restarting it. The basic operation of the queuing system starts with job submission. All necessary information for executing a job are stored in a data structure, a job data block, and submitted to the queue object. The queue object creates a unique ticket for the job (which is returned to the user as a response to

the submission), as well as a record to store the data block, the ticket, and other job-related information. This record gets stored in a priority queue structure, and waits for its turn to be dequeued and used for executing the job. At this point two things are possible: either the job will get successfully started and its job record will get moved to the list of running jobs, or it will fail to start, in which case the job record will get moved to the list of failed jobs. The run operation is non-blocking; once the job finishes execution, the queue object receives notice that execution is over by means of a callback, in the same manner that a CPU is notified of an event with an interrupt. Once a job has finished its record gets moved to either the list of done jobs, if execution was successful, or the list of failed jobs otherwise (see Fig. 3).

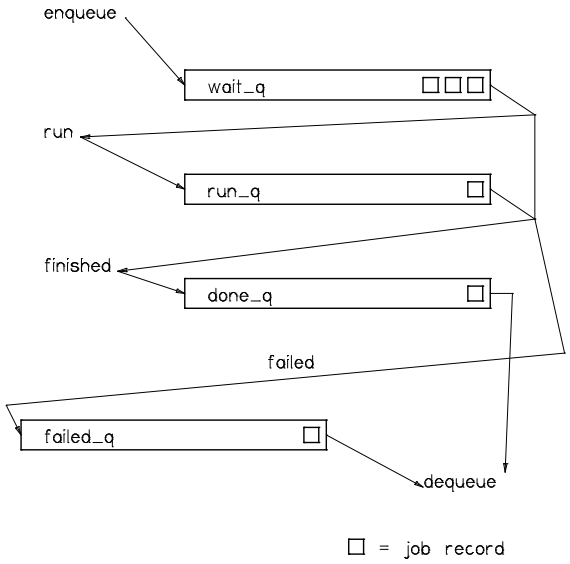

Fig. 3. Diagram of internal queue object operations.

The queue object will attempt to run a job whose record is waiting at the head of the priority queue when there is an available run slot. The initial attempt is always after a job submission. If this attempt fails, the queue object will attempt again after another job submission or in the following cases:

- when a job is done,
- when a job fails,
- when the number of allowed jobs gets increased.

Note that the job that will run in any case is always the one waiting at the head of the queue and not the one submitted, unless they're the same. The basic operation cycle ends with an inquiry about the job by the user, in which case

the queue object responds with the job's status. If the job is done or has failed, the corresponding record gets marked for deletion, and later on gets deleted, at which point the job exits the queuing system.

The JobQueue object also performs two periodical independent operations, using timer alarms. The first one involves checking for job records marked for deletion and deleting them if a certain amount of time has passed. The second one is much more complex as it is the main job monitoring operation. During this operation the queue object:

- pings running jobs to verify they're alive,
- updates job statistics,
- kills jobs that have exceeded their maximum allocated running time, and
- restarts jobs that have exceeded their maximum restart time period.

The ping operation is non-blocking in the same manner as the run operation described above. A problematic job is bound not to respond to a ping, thus the queue object should never block and wait when pinging jobs. Instead, proxy objects report status to the queue object. A running job is the child of a proxy object and is being watched over by its parent. The proxy object will notify the queue object when the child's execution terminates for whatever reason. Thus, the pinging operation is intended to watch over the proxy objects themselves, since a failed proxy object causes faulty job monitoring. The queue object attempts to kill jobs that do not respond to pings for a certain amount of time and moves their corresponding records to the list of failed jobs.

Whenever the queue object performs an operation on a job, it "pretends" to be the owner of the job by switching its own method parameters, its "credentials", to the ones of the actual job owner, then switching back again. In this way the queue object invokes methods on other system objects on behalf of the job owner. Thus, any security privileges and restrictions that the owner may have remain in effect.

4.4 Implementation

The initial implementation of the queuing system has as its main focus the throttling, handling, and monitoring of jobs. It does not yet address the control centralization and resource coordination issues. The invocation of JobQueue object methods is controlled by means of an access control list (ACL). Certain methods are accessible only by the object owner. As the administrator is typically the owner of the (all) queue object (objects) it is not normally possible for regular users to affect the queuing system operations in any way other than they are allowed to (while not directly restricted, we have not experimented with users creating their own JobQueues). Additionally, when a tool operation affects a specific job in some manner (e.g. when killing a job), the tool includes in its communication to the queue object the identity of the user who made the request. The queue object always enforces the rule that certain operations can be requested only by the user who is the actual job owner, or the administrator.

For example, it will prevent user A from killing a job belonging to user B, unless A is the administrator.

We are currently investigating the user of the JobQueue across NPACInet, which is a Legion network in daily operation across NPACI resources of CalTech, Virginia, SDSC, Michigan, Texas, etc. [6] We are working with a select number of users to evaluate the effectiveness of the API and core mechanisms. Initial experiments have examined reliability issues with good results, e.g. handling of over 10,000 job submissions (several different jobs, each with a number of instances) presented no problems. Even after system failures the JobQueue "remembered" its previous status and continued to handle running and submitted jobs.

5 Conclusions

The significant problem a metaqueuing system has to solve is the throttling of grid-wide simultaneously executing jobs. Without control over the number of jobs the response time of the system becomes totally unpredictable. The JobQueue, a metaqueuing system for the Legion grid computing system, provides this important functionality, as well as global job handling and monitoring. In its current implementation the JobQueue is intended to be an administrative system object and perform job control with a single instance. Future work will focus on offering solutions to the issues of non-centralization of control and resource coordination in the presence of multiple instances, privileged vs. unprivileged creation of metaqueuing systems, and predictability of metaqueuing systems.

References

1. Grid Computing Info Center (*http://www.gridcomputing.com*)
2. A. Bayucan, R.L. Henderson, C. Lesiak, N. Mann, T. Proett, and D. Tweten. "Portable Batch System: External Reference Specification." Technical Report, MRJ Technology Solutions, November 1999.
3. Steve J. Chapin, Dimitrios Katramatos, John Karpovich, Andrew Grimshaw. "Resource management in Legion." *Future Generation Computer Systems*, vol. 15 (1999), pages 583–594.
4. A. Grimshaw and W. Wulf. "The Legion Vision of a Worldwide Computer." *Communications of the ACM.* pages 39–45, January 1997.
5. A. Natrajan, M. Humphrey, and A. Grimshaw. "Capacity and Capability Computing using Legion." In *Proceedings of the 2001 International Conference on Computational Science*, San Francisco, CA, May 2001.
6. A. Natrajan, A. Fox, M. Humphrey, A. Grimshaw, M. Crowley, N. Wilkins-Diere. "Protein Folding on the Grid: Experiences using CHARMM under Legion on NPACI Resources." In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC)*, San Francisco, California, August 7-9, 2001.
7. Bill Johnston, Dennis Gannon, and Bill Nitzberg. "Grids as Production Computing Environments." In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing*, 1999.

8. LSF Reference Guide, Version 4.0.1, June 2000.
(*ftp://ftp.platform.com/docs/lsf/4.0/html/ref_4.0.1/index.html*)
9. Global Resource Director.
(*http://www.arl.hpc.mil/docs/grd/grd.html*)
10. Joe Werne and Michael Gourlay. "Uniform Command-line Interfaces for Job Submissions and Data Archiving." Presentation given at GGF-2, July 15-18, 2001, Vienna, Virginia USA.