Monitoring Remote Jobs in a Grid System

Anand Natrajan, Michael P. Walker
Department of Computer Science, University of Virginia
Charlottesville, VA 22904
{anand, mpw7t}@virginia.edu

Abstract. *When users submit jobs to a grid system, they desire to monitor the progress of the jobs for a number of reasons. In this paper, we present a small set of attributes about jobs that are of interest to users. Some grid systems already provide mechanisms for retrieving some of these attributes, whereas others do not. We use Legion as the implementation platform for demonstrating how these attributes can be retrieved.*

1 Introduction

As available computing power increases because of faster processors and faster networking, computational scientists are attempting to solve problems that were considered infeasible until recently. Grid systems are becoming more pervasive platforms for running distributed jobs to solve such problems. A *grid system* is an environment in which users, such as scientists, can access resources in a transparent and secure manner.

In a grid system, when a user submits a job, the system runs the job on distributed resources, and enables the user to access the results of the jobs when they complete. This model of submitting jobs and accessing their results is similar to the batch processing model prevalent in the early days of computing.

Often, users desire to monitor their jobs *while they execute*. Some of the reasons why users choose to monitor their jobs are:
- To check on the progress (or lack thereof) of the jobs[*]
- To access intermediate results, either for checkpointing or for display
- To terminate the jobs if their progress is undesirable
- To steer the computation of the jobs after they have begun
- To check on the resource consumption of the jobs
- To share the results of their jobs in a secure manner

Naturally, many more reasons, some application-specific, may exist.

In this paper, we present a small set of attributes of executing jobs that may be of interest to a user. Finding a complete set of attributes is a futile task, particularly because some of the attributes may be application-specific. Instead, our approach is to present a reasonable set of attributes that grid systems must make available to users. Individual grid systems may make additional attributes available.

In §2, we discuss how users monitor jobs in traditional (i.e., non-grid), systems. We present how existing grid systems permit users to monitor jobs and contrast and compare these approaches with traditional systems. In §3, we present a set of attributes of jobs. A grid system must enable a user submitting a job to access these attributes. In §4, we demonstrate how to access these attributes in the Legion system [GRIM98]. In §5, we summarise this discussion.

2 Related Work

We present advantages and disadvantages of job monitoring in traditional (i.e., non-grid) systems and queuing systems and discuss how grid systems can improve on it. Also, we show how existing grid systems enable monitoring jobs, and contrast them with traditional systems. We compare and contrast the tools provided by existing grid systems with those provided by traditional systems. Given the large number of grid systems in existence, we evaluate only a few representative systems.

2.1 Traditional Systems

In traditional or non-grid systems, after a user initiates jobs, she has the ability to monitor the jobs as they execute. Unix-like operating systems provide a rich set of tools that can be used to monitor jobs [RIT74]. For example, `ps` and `top` can be used to monitor the process status of jobs. Typically, these tools display the process IDs of each job, the process ID of the parent of each job, the CPU usage of each job, the memory consumption of each job

---

[*] A user's lack of confidence in the progress of her jobs may stem from her lack of confidence in the application as well as her lack of confidence in the grid system itself, particularly in its ability to sustain jobs in the "running" state. The latter attitude is not seen usually with traditional non-grid operating systems. As grid systems evolve from their current, incipient implementations, this attitude can be countered.

and its status according to the operating system. Likewise, `du` and `df` can be used to monitor the disk usage and available disk space for the jobs. Tools such as `traceroute` and `netstat` can be used to monitor the network behaviour of the jobs. In Windows-like operating systems, a number of graphical tools are available that can be utilised to monitor jobs [RIC98].

Traditional systems provide tools to perform all the tasks listed in §1 (and more). Tools such as `ps`, `du` and `traceroute` can be used to check on the progress of jobs and to monitor their resource consumption. Additional input files or can be sent and intermediate results accessed over NFS or using tools like `ftp`. Jobs can be terminated by sending them signals either from the command line or by using tools like `kill`. Finally, users can share the results of their jobs by setting appropriate permissions on files created by their jobs. Although in many cases the tools were not written for the explicit purpose of job monitoring, as users have become more familiar with the available tools, they have been able to use them for job monitoring. However, traditional systems suffer a number of disadvantages with respect to job monitoring:

- Checking the progress of a job or monitoring its resource consumption typically requires starting a shell on the machine on which the job is executing, in turn, requiring that the user have an account on that machine. As the number of machines participating in a grid increases, requiring users to procure accounts on each of them becomes unscaleable rapidly.
- Accessing intermediate files or providing additional input files requires shared file system support or `ftp`-like capability. File systems shared by widely-distributed machines can be inefficient. Tools like `ftp` and `scp` typically require users to have accounts on the machines.
- Checking aggregate progress of large sets of jobs is difficult. Users must check on the progress of each job individually. Alternatively, they must construct interfaces to monitor the progress of the entire set of jobs.
- Monitoring jobs typically requires knowing on which machines the jobs are executing. For large sets of jobs, requiring users to know or record the names of the machines on which their jobs are running is unscaleable.
- Typically, users can monitor not only their own jobs but also other jobs on the machine. For example, tools like `ps` and `top` can be used to monitor the jobs of other users, albeit in a limited manner. For some users, even this limited monitoring may be an invasion or privacy.

These disadvantages are not inherently a result of bad design. Rather, the tools are well-designed extensions to operating systems that were not meant to be grid systems.

In summary, traditional systems offer users a rich set of tools that can be utilised to monitor jobs. However, those tools come with a few disadvantages. With grid systems we have the opportunity to construct job monitoring tools that combine the best features of traditional tools with the benefits of grid systems.

2.2 Queue Systems

Traditionally, queue systems have been used to manage job scheduling on a cluster of nodes [BAY99] [FER93] [IBM93] [ZHOU92] [ZHOU93]. Typically but not necessarily, the nodes are homogeneous and are located close to one another (e.g., within a single building). Most queues operate in "batch" mode. In other words, when a user submits a job, the submission program immediately returns the user to the prompt and provides her with a ticket or job ID or token, which can be used to monitor the job at any later time. The ticket becomes invalid shortly after the job completes. Most queuing systems comply with a POSIX interface that requires three standard tools for running jobs:

- a submit tool (`qsub` in PBS, `bsub` in LSF, `llsubmit` in LoadLeveler)
- a status tool (`qstat` in PBS, `bjobs` in LSF, `llstatus` in LoadLeveler)
- a cancel tool (`qdel` in PBS, `bkill` in LSF, `llcancel` in LoadLeveler)

In addition, some queues provide other tools to check on the aggregate status of the queuing system (e.g., `bqueues` in LSF and `llq` in LoadLeveler).

In queuing systems, the progress of jobs can be checked with the status tool. Typically, the progress indicator returned is the queuing system's view of a job. For example, the queue may report that a particular job is queued, running or terminated. Some queuing systems, such as LSF, permit application-specific status information to be returned to the user. The application-specific status augments the status returned by the queuing system. If the continued execution of a particular job is deemed undesirable, the cancel tool can be used to terminate the job. Most queuing systems do not provide tools to access intermediate files or supply additional inputs. If such functionality is desired by the user, the user must employ shared file systems or other file transfer tools. In most queuing systems, the user can specify limits on the resources consumed by jobs, but cannot determine how much of those resources are actually consumed by the jobs. Queuing systems do not provide any support for checking aggregate progress of large sets of jobs. Users must check on the progress of each job individually or construct interfaces to monitor the progress

of the entire set of jobs. Monitoring jobs does not require knowing on which machines the jobs are executing. Finally, most queuing systems can be configured such that only privileged users can see the status of all jobs; ordinary users can see the status of only their own jobs.

2.3 Grid Systems

A grid system is an operating system for managing heteregeneous resources distributed over a network. Typically, in a grid system, when a user submits a job, the job runs with the permissions of an ordinary user. Grid systems typically span multiple organisations and administrative domains. Often grid systems run on machines that are controlled by queuing systems. Currently, there are no standard methods for monitoring the progress of jobs executed by grid systems. Different grid systems use different techniques for monitoring jobs.

In Globus [FOST99], a user submits a job to a specific set of machines. After the jobs have begun executing, the user may log on to any of the machines and monitor specific jobs. Naturally, this process requires the user to possess accounts on each machine. However, if the user does have accounts on the machines, the user may use any of the tools available on traditional systems to monitor the job.

In Legion [GRIM97], monitoring jobs is rudimentary (until version 1.6.5). In Legion, running jobs are objects, similar to hosts, files and directories. Legion tools that submit jobs for execution monitor the objects created on behalf of the jobs. However, the information returned to the user is sparse. If the user indicates that a particular job must run on a particular host machine, and the user happens to have an account on that machine, then the user may log on to that machine and use tools available on traditional systems to monitor the progress of the jobs. If a user submits a job but does not specify the machine on which the job must run, then the user may not be able to access any information about the job.

In Nimrod [ABR95], the system itself monitors jobs submitted by the user. The Nimrod system starts a graphical tool that the user can use to observe the progress of jobs and terminate jobs if necessary. The system does not provide any means for accessing intermediate files or providing additional input files unless the user logs on to a system directly and uses traditional tools.

None of the above systems provide any mechanism for checking on the resource consumption of a job. With the exception of Nimrod, none of the systems provide aggregate job monitoring facilities. None of these systems permit a user to monitor other users' jobs unless the user presents the correct credentials or circumvents the grid system altogether by logging on to a specific machine. In summary, grid systems provide sparse support for monitoring jobs. There is room for improvement in the amount of information a grid system can provide about jobs.

3   Job Attributes

In this section, we construct a reasonable set of attributes of running jobs and discuss the mechanisms for accessing these attributes. Users submitting jobs to a grid system should be able to access these attributes for their jobs and their jobs alone. If a grid system requires a privileged user or administrator, this user may also be permitted to access these attributes for every job in the system. Generally, there are three kinds of jobs that are submitted to a grid system:

1. Sequential jobs, i.e., single-process jobs
2. Parallel jobs, i.e., multi-process jobs
3. Parameter-space jobs, i.e., multiple instances of typically sequential jobs

For each of these jobs, the minimum set of attributes are:
- status of the jobs according to the grid system
- name of the machine on which the job is running
- working directory of the job
- list of the files in the working directory of the job
- permissions, timestamp and size of any file in the working directory of the job

Therefore, a reasonable set of operations on a running job is:
- get status of the job
- get name of the machine on which the job is running
- get the name of the working directory for the job
- get the list of files in the working directory of the job
- get the permissions, timestamp and size of a file in the working directory of the job
- get any file from the working directory of the job
- send any file to the working directory of the job

- delete any file from the working directory of the job
- terminate the job

Additional attributes and operations are possible and encouraged. For example, a particular grid system may offer the following attributes:

- application-specific status
- CPU usage on the machine on which the job is running

Accordingly, additional operations that can be provided may be:

- get application-specific status
- get CPU usage of the machine on which the job is running
- change the priority or "nice"ness of the job

Other operations that a particular grid system may provide may be:

- archive the working directory of the job after the job is complete
- send arbitrary signals to the job

In this paper we will describe an implementation only of the first set of operations. We believe that this set is reasonable and small, whereas the latter set is somewhat esoteric. As users run large numbers of jobs run on grid systems, their patterns of usage may educate grid designers about an even more reasonable set of operations.

4   Legion Implementation

Legion is an architecture for a grid system [GRIM98]. Just as an operating system provides an abstraction of a machine, Legion provides an abstraction of the grid system. This abstraction supports the current performance demands of scientific applications. A number of applications already run using Legion as the underlying infrastructure. In the future, users will demand support for new methods of collaboration. Legion supports these expected demands as well.

The Legion project is an architecture for designing and building system services that present users the illusion of a single virtual machine. This virtual machine provides secure shared objects and shared name spaces. Whereas a conventional operating system provides an abstraction of a single computer, Legion aggregates a large number of diverse computers running different operating systems into a single abstraction. As part of this abstraction, Legion provides mechanisms to couple diverse applications and diverse resources, vastly simplifying the task of writing applications in heterogeneous distributed systems.

Each system and application component in Legion is an object. Running instances (a.k.a. jobs) of programs are objects. All Legion objects respond to a set of mandatory methods (for example, ping and list-attributes). In addition, specific objects may respond to additional methods. For example, as of version 1.7, the objects corresponding to running programs have been written to respond to methods corresponding to the operations outlined in §3.

Users may start either legacy jobs or jobs that use Legion libraries and objects using a number of tools that are part of the Legion distribution:

- `legion_run` enables users to start legacy sequential jobs
- `legion_native_mpi_run` enables users to start legacy MPI jobs
- `legion_mpi_run` enables users to start Legion MPI jobs
- `legion_run_multi` enables users to start parameter-space jobs, typically of sequential legacy programs

Each tool starts a Legion "runnable" object on some machine. This runnable object then starts the user's binary on the machine, either by a fork/exec or a queue submit. After starting the binary, the runnable object periodically checks the status of the binary using whatever mechanisms are available on that machine.

After starting a job, a user may inquire about its status. In Legion, this action translates into invoking specific methods on the runnable object corresponding to that job. The Legion tools that enable status inquiries are:

- `legion_probe_run` enables users to check legacy sequential or MPI jobs
- `legion_mpi_probe` enables users to check Legion MPI jobs

After a user starts a job, he may request that a probe be returned. This probe is similar to a job ID or ticket returned by a queuing system. Subsequently, the user may use this probe and one of the tools above to inquire about the status of the job. Suppose a user started a legacy sequential job. He would have used `legion_run` in the following manner:

```
legion_run -v -probe pfile -IN file1 -IN file2 -OUT file3 myClass arg1 arg2
```

In the above command, the probe is stored in the local file *pfile*. The user has indicated that the local files *file1* and *file2* are to be supplied as input files and the file *file3* is to be retrieved as an output file at the end of the run. *myClass* is the name of the Legion object that corresponds to the user's program. `legion_run` effectively creates an instance of *myClass* on some remote machine in order to run the user's job. *arg1* and *arg2* are arguments for that job.

Once the job has been started (i.e., the Legion runnable object has been created) on the remote machine, the file *pfile* is written. The user can now inquire about the status of the job using commands like the ones below:

```
legion_probe_run -probe pfile -statjob -pwd -hostname -list
legion_probe_run -probe pfile -IN file4 -stat file4 -OUT file2
legion_probe_run -probe pfile -chdir subdir -pwd -kill
```

The first of these commands requests Legion to print, in that order, the status of the job, the current working directory of the job, the machine name on which the job is running and a list of the files in the current working directory of the job. The second command requests Legion to send in a new file *file4* as an input file, print the permissions, size and timestamp of *file4* and get *file2* as output. These new inputs and outputs do not affect the inputs and outputs requested in the `legion_run` earlier. The third command requests Legion to change the working directory of the runnable object, print the new working directory and kill the job. The options to `legion_probe_run` may be specified in any order (with the caveat that nothing is executed after a `-kill`).

This probing mechanism can be used not only to check the status of jobs, but also to steer the progress of the job in interesting ways. For example, if *myClass* was written appropriately, sending in *file4* later may change the rest of the computation. Retrieving *file2* periodically can be used to display intermediate results or save checkpoints. Since all of these operations are performed on the Legion runnable object, and since every Legion object has access control lists associated with it, a single job can be started, viewed, steered and terminated by different sets of users.

The Legion mechanisms provided above are rich enough to monitor the progress of jobs in detail. However, the mechanisms are flexible enough so that new features can be added as they are deemed essential. For example, suppose at a later date it is deemed important that a user should be able to check the CPU load on the host on which hisor her job is running. Adding that functionality to the Legion runnable object and the `legion_probe_run` tool is straightforward, and requires less than an hour's worth of programming effort.

5   Summary

The ability to monitor the progress of jobs is essential to any user. Most grid systems enable users to monitor jobs. Some of these systems, for example, traditional operating systems, provide a rich set of tools, but can be too permissive. Other systems are less permissive, but do not provide enough richness in the toolset for many users.

We have identified a set of key attributes of a job, namely, job status, machine name, working directory, available files and status of each file. In our experience, users often desire to know these attributes about their job. We implemented a job monitoring scheme in Legion. The tools used for job monitoring have been deployed and are being used widely. The mechanism for implementing these tools is rich yet flexible enough to accommodate future improvements.

6   References

ABR95   Abramson, D., et al., *Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations*, Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing, August 1995.

BAY99   Bayucan, A., Henderson, R. L., Lesiak, C., Mann, N., Proett, T., Tweten, D., *Portable Batch System: External Reference Specification*, Technical Report, MRJ Technology Solutions, November 1999.

FER93   Ferstl, F., *CODINE Technical Overview*, Genias, April 1993.

FOST99   Foster, I., Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

GRIM97   Grimshaw, A. S., Wulf, W. A., *The Legion Vision of a Worldwide Virtual Computer*, Communications of the ACM, Vol. 40, No. 1, January 1997.

GRIM98   Grimshaw, A. S., Ferrari, A. J., Lindahl, G., Holcomb, K., *Metasystems*, Communications of the ACM, Vol. 41, No. 11, November 1998.

IBM93   International Business Machines Corporation, *IBM LoadLeveler: User's Guide*, September 1993.

KING92   Kingsbury, B. A., *The Network Queueing System (NQS)*, Technical Report, Sterling Software, 1992.

RIC98   Richter, J., *Custom Performance Monitoring for your Windows NT Applications*, Microsoft Systems Journal, August 1998.

RIT74   Ritchie, D. W., Thompson, K., *The UNIX Time-sharing System*, Communications of the ACM, Vol. 17, No. 7, July 1974.

ZHOU92   Zhou, S., *LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems*, Workshop on Cluster Computing, December 1992.

ZHOU93   Zhou, S., Wang, J., Zheng, X., Delisle, P., *Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Software Practice and Experience, Vol. 23, No. 2, 1993.