The Legion JobQueue

Dimitrios Katramatos
University of Virginia
May 2000

1 Introduction

This document describes the role, design, implementation, setup, and use of a job queuing system for Legion. This system improves upon the way Legion handles jobs as it can control the number of jobs that are executing simultaneously in a Legion system. The main component of the queuing system is a job administrating object, the “JobQueue”, which offers a priority-based job waiting queue, job control, and job monitoring features.

2 Role of the JobQueue

2.1 Traditional Job-handling in Legion

Currently, job-handling in Legion is done by means of the following set of command-line tools:

- `legion_registerRunnable` for registering runnable Legion programs,
- `legion_registerProgram` for registering non-Legion programs,
- `legion_run` for running a registered program,
- `legion_run_multi` for running multiple registered programs (uses `legion_run`).

The first two commands register programs with the Legion system (create class objects and register implementations with them) and the last two start program instances. The procedure for running a runnable Legion program is standard; the program’s class is instructed to create an instance of the program which will actually execute the program code. The procedure is more complex when dealing with non-Legion programs (also scripts, etc.);
a special metaclass is needed, the BatchQueueMetaClass (BatchQueueClass-
object), used to create a special kind of class object to act as the class
object of the non-Legion program. Also, another standard class object, the
JobProxyClass, serves as the creator of JobProxyObjects which are essen-
tially program watchers. Instructing the program’s class to create an in-
stance actually creates an instance of the JobProxyClass, a job proxy, which
when instructed to execute the program forks and execs the corresponding
binary. Figure 1 depicts the above procedure.

2.2 New Functionality

Although the legion_run mechanism covers the basic service of executing
programs, it has certain disadvantages:

- one legion_run “watcher” process is necessary per running job, thus
  the resource requirement on the front-end machine increases rapidly
  when a large number of jobs is executed, which can lead to crashes,

- job progress cannot be monitored,

- jobs cannot be restarted,

- there is no way of controlling the total number of jobs executing simul-
taneously in the Legion system, as any user can start any number of
legion_run sessions, at anytime.

The JobQueue provides the additional functionality needed to overcome
these disadvantages. While the procedure for registering executables with
the Legion system remains exactly the same, in the new queuing system the
operation of legion_run is broken down to three phases:

1. job submission,

2. job preparation, execution, and monitoring, and

3. job cleanup.

The main component of the queuing system, the JobQueue (an instance of
a JobQueueObject owned by the Legion system administrator), is responsible
Figure 1: Current non-Legion program executing scheme.
for phase 2. Phases 1 and 3 are done with the use of special command-line 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. Figure 2 shows the place for the JobQueue in the job-handling procedure (compare with figure 1).

Breaking down the operation of \texttt{legion\_run} in the above mentioned phases and assigning the main portion of the operation to an administrator-owned object makes possible to control the number of jobs that execute simultaneously in the Legion system. A submitted job will enter phase 2 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.

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 use of the JobQueue yields significant resource savings. Essentially, one special object, corresponding to a single process running on a Legion host, can take care of hundreds or even thousands of jobs. With \texttt{legion\_run} there is one-to-one correspondence of job processes with watcher processes, which means that e.g. 1000 jobs require 1000 watchers. Additionally, with the traditional scheme one cannot do much about a job while executing other than killing it by killing its watcher, which will propagate the kill signal appropriately.

To avoid having one centralized component responsible for all jobs in a whole system, it is possible to have a set of JobQueue instances, each one responsible for a portion of the Legion system’s resources. Assigning system resources to a JobQueue instance and default JobQueue instances to system areas is more a matter of policy than of mechanism. The current implementation of the JobQueue object does not have support for enforcing any restrictions on resources (nor is it clear that it’s necessary). It is conceivable
Figure 2: JobQueue operation for a non-Legion program.
that minor code modifications might be necessary for this purpose.

3 Queuing System Design

The queuing system has four components:

- JobQueue object, the main component,
- \texttt{legion\_nq} command-line tool for submitting jobs,
- \texttt{legion\_manage\_job} command-line tool for performing certain operations on jobs, and
- \texttt{legion\_manage\_queue} command-line tool for performing certain operations on JobQueue objects.

The basic operation of the 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. Once the job finishes execution it “notifies” the queue object that it is done. This is possible by executing a program graph which invokes the proxy object’s run method to start executing the job, but also treats the return value of this method as an argument for subsequently invoking a special method of the queue object which takes care of done jobs. Thus, when a job finishes execution the job proxy object responds with a return code; this response fires up the second method (essentially a callback) which notifies the queue object of the event in the same manner as a CPU is notified of an event with an interrupt. The queue object does not have to block and wait for a job to finish and can go about its business. 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. The operation is depicted in figure 3.
Figure 3: Internal JobQueue operation.
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 queue 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, jobs call a second special queue object method to report their status. 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 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.

The three command-line tools mentioned above perform their functions by preparing a set of data and invoking the corresponding method of the queue object. `legion_nq` and `legion_manage_job` also perform, when necessary, the important function of copying the files a job requires to run back and forth between the Legion filesystem and a user’s local filesystem.

For security reasons, the invocation of JobQueue methods is controlled by means of an ACL. Certain methods are accessible only by the administrator. As the administrator is the owner of the (all) queue object (objects) and the corresponding class object, it is not normally possible for regular users to affect the queuing system operations in any way other than they are allowed to (for example, a user cannot start his/her own JobQueue). Additionally, when a command-line 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. Therefore it will prevent user A from killing a job belonging to user B, unless A is the administrator.

4 Queuing System Implementation

4.1 Infrastructure

As mentioned in the previous section, the queue object uses a priority queue structure to store the job records and determine the order of job startup and execution. The abstract data type for this purpose is called LPQueue
(Legion Priority Queue) and is a general-purpose, packable, priority queue structure. The operations supported by an LPQueue are the following:

- enqueue element
- append element (internal use)
- dequeue head element
- dequeue specific element (internal use)
- change priority of element
- find element (internal use)
- number of elements
- show
- pack
- unpack

The queue elements are of type LPQueueElement; this abstract data type is essentially a data container for use with LPQueue and consists of a triplet containing a data structure, a priority structure, and a search-key structure. For the case of the JobQueue, the data structure is a job record, the priority structure is an integer number wrapped in a C++ class, and the search-key structure is a job ticket.

When submitting a job, legion.nq packs all job-related information necessary for executing the job in a data structure called LegionJobDataBlock. The contents of the data block are the following:

- all arguments from command line,
- owner LOID,
- owner implicit parameter list.

The queue object creates a ticket for the submitted job and returns it as a response. The ticket data are stored in a LegionJobTicket structure. This becomes the full identity of a job and is used in all transactions related to this job. The contents of a ticket are the following:
- ticketNo: unique positive integer,
- LOID of job class,
- LOID of job object,
- time of issue,
- LOID of queue,
- LOID of owner.

The pair (ticketNo, LOID of queue) uniquely identifies a job in the Legion system. Note that the owner LOID is included in both the ticket and the data block. This is because \texttt{legion_nq} submits a data block and not a ticket. The ticket, however, includes the job owner for security reasons when communicating with the queue object in regard with a specific job. The integer number used in the ticket is generated by a counter (UVaL\_Counter). This kind of counter counts between \(-10,000,000\) and \(+10,000,000\). As only positive numbers are used as ticket numbers, a ticket number ranges between 0 and 10,000,000. When the 10,000,000 limit is exceeded, the counter resets to 0. Although the queue object doesn't perform any checks to avoid to use duplicate numbers, it is unlikely that new jobs will be assigned ticket numbers that are still in the system, as jobs exit the system sooner or later (job records get deleted). The possibility exists however, and in such a case the LPQueue structure will refuse to enqueue the new job as it will have a duplicate key. The user will then have to re-submit the job, so that it gets assigned a different ticket number.

The data structure holding all job-related information that is used by the queue object for seeing a job through is the LegionJobRecord. Its contents are the following:

- the job's ticket,
- the job's data block,
- timestamps, and
- other necessary internal bookkeeping information.
4.2 Objects and Command-line Tools

An instance of a JobQueueObject is the only object that the queuing system requires. The object’s methods are included in the following list:

- interface
  - **jobSubmit** enqueues a job,
  - **jobStatus** reports status of a single job; also marks done or failed jobs for deletion,
  - **jobPriority** changes a job’s priority while in the waiting queue,
  - **jobKill** kills a running job or dequeues a waiting one,
  - **jobRestart** restarts a running or failed job,
  - **jobsList** lists all job records in the system,
  - **jobSlotsSet** sets the number of running slots,
  - **jobSlotsIncrease** increases the number of running slots and stores the LOID of the invoker (intended to be used by host objects that are looking for work),
  - **jobsPurge** dequeues waiting jobs, kills running ones, and ultimately deletes all job records,
  - **jobDone** special use, gets invoked when a job finishes execution to perform partial cleanup and deactivate any temporary context and file objects involved,
  - **jobReport** special use, gets invoked as a response to a ping,

- internal use
  - **jobRun** non-blocking, invoked internally to start a job’s execution,
  - **jobFailed** invoked when a job fails to startup,
  - **jobsPing** alarm, non-blocking, periodically pings all running jobs,
  - **pingOne** called by jobsPing to ping an individual job,
  - **jobsDelete** alarm, periodically deletes jobs that were marked for deletion by jobStatus,
**PrivateSaveState** saves the internal state of the queue object; for fault tolerance reasons, this method gets invoked internally whenever jobsPing and jobsDelete are activated,

**PrivateRestoreState** the complement of the save routine, restores the internal state of the queue object.

The queuing system’s command-line tools perform a variety of operations in co-operation with a queue object. More specifically:

**legion.nq** is the command-line tool used to submit jobs:

- it is used the same way as legion.run, and it supports the same options, plus a few extra ones; it can be used by any user,
- copies files from a user’s local filesystem to the Legion filesystem and subsequently deactivates the corresponding temporary context and file objects to save process slots (if -d is specified); these objects will become automatically active when referenced,
- it supports the following extra options:
  - `-p < integer >` specifies the priority of a job; the accepted values are between 5 and 9, 9 is the lowest priority,
  - `-r < integer >` specifies the number of minutes after which job is restarted,
  - `-q < contextpath >` specifies which queue object to contact (default is `/etc/JobQueue`),
  - `-d` instructs the tool to deactivate file objects, see above,
  - `-help` displays a usage page.
- it will propagate tty information to the queue object so that stdout and stderr output from the program can be redirected to the window where the tool executed.

**legion.manage.job** is the command-line tool used to manage a job:

- it accepts as necessary argument the ticket number of a job,
- its default operation is to report the status of a specified job; if the job is done or has failed, the tool also copies back files from the Legion filesystem to the local filesystem, cleans up temporary
files and contexts, and marks the corresponding job record for deletion; only the job’s owner and the administrator are allowed to perform these operations on a job,

- the options supported are the following:
  - **-k** kills specified job (owner/administrator),
  - **-cp** < integer > changes job’s priority, accepted values are between 0 and 9, 0 is the highest priority; only the administrator can use this option and can specify higher priorities (0 to 4) than a regular user,
  - **-fr** forces the restart of a specified job which is running or has failed (owner/administrator),
  - **-q** < contextpath > specifies which queue object to contact (default is /etc/JobQueue),
  - **-nc** forces the tool to not perform file cleanup for the specified job,
  - **-help** displays a usage page.

`legion manage queue` is the command-line tool used to manage a queue object:

- its default operation is to list all jobs; any user is allowed to do this,
- it supports the options:
  - **-maxrs** < integer > sets the number of the maximum run slots - administrator only,
  - **-purge** kills all jobs - also administrator only,
  - **-q** < contextpath > specifies which queue object to contact (default is /etc/JobQueue),
  - **-help** displays a usage page.

### 4.3 Source files, Default Values, and Status Codes

The set of source code files for the queuing system includes the following files:

**Queue object files:**
- JobQueueObject.h
- JobQueueObject.c
- JobQueueObject.trans.h
- JobQueueObject.trans.c
- JobQueueObjectFunctionIdentifiers.h
- JobQueueObjectFunctionIdentifiers.c

**Tools** files:

- legion_nq.c
- legion_manage_job.c
- legion_manage_queue.c

**ACLs**: 

- JQC.acl (acl for /class/JobQueueClass)
- JQCI.acl (acl for an instance of /class/JobQueueClass, i.e. a JobQueue)
- legion_initial_security_policy.h

**Auxiliary** files:

- LPQueue.h
- makefile
- setupJobQueue

In order to compile the system, one has to copy all files into the Legion/src/ServiceObjects/Queuing directory (makefile will overwrite the original file), except for legion_initial_security_policy.h which has to be copied into the Legion/src/UserObjects/Security directory (will also overwrite the original file). File setupJobQueue is not needed for compilation. All files except makefile and legion_initial_security_policy.h are new to the standard system file set.

The file setupJobQueue is a script that has to be executed by the Legion system administrator. It creates the objects /class/JobQueueClass and
/etc/JobQueue and creates and sets the ACLs for these two objects. It is possible to skip the ACL creation phase since the ACLs are available in the file set; the ACLs have to be re-created if something changes in the system’s policy.

Several default values for important parameters are included in the file JobQueueObject.h. Among them the following:

**MAX_KEEP_TIME** determines the time in minutes that a job record will be kept in the system when it is marked for deletion after the job finishes,

**MAX_TIME_BETWEEN_PINGS** determines the time in minutes that a job has to respond to a ping; if it gets exceeded, the job is considered hung,

**MAX_JOB_PRIORITY**, **MIN_JOB_PRIORITY** specify the lower and upper integer limit of the priority number range (note that the lowest number corresponds to the highest priority),

**DEFAULT_JOB_PRIORITY** specifies the default priority,

**DEFAULT_MAX_RUN_SLOTS** specifies the default number of run slots.

Another two important parameters are included in the JobQueueObjects.trans.c file:

**L_JOBQUEUE_JOBSPING_INTERVAL** determines the timer alarm interval in seconds for executing the jobsPing method, and

**L_JOBQUEUE_JOBSDELETE_INTERVAL** determines the same for the jobsDelete method.

Note, however, that these alarms are not regular interrupts and get triggered only when the Legion system allows them to. Thus, the real interval can be a lot longer than the actual specified.

Finally, the following list explains the job status descriptions that the JobQueue reports when the jobStatus or the jobsList method are invoked:

**WAIT** the job is waiting its turn in the waiting queue,
STBY the job is getting prepared to run,

RUN the job is executing,

CLNUP the job has just finished executing and the JobQueue performs object clean-up,

DONE the job has successfully completed execution,

FAIL40 the job failed to start because the specified host was not found,

FAIL41 the job failed to start because the job object (an instance of the program class) could not be created,

FAIL42 the job failed because the program is not runnable,

FAIL46 the job failed because the remote run method was unsuccessful,

FAIL47 the job failed because the remote run failed,

FAIL48 the job failed because the program class is invalid (i.e. doesn’t exist, wrong name, wrong context path) and the job object could not be created,

FAIL49 the job failed because there is no implementation for the specified architecture,

CANCEL the job was dequeued (from the waiting queue) as a result of a kill operation,

KILLED the job was killed.

5 Unresolved Issues

There are at least two unresolved issues with the queuing system. The first issue is the matter of having multiple queue objects in a Legion system. What needs to be defined here, is the way of sharing the system resources among the objects so that there is not only one point of job control. Centralized control does not scale well and has fault tolerance problems. On the other hand, having many queue objects without restricting the resources that each
one can access creates grave job control problems. However, one JobQueue might be enough for moderate-size systems.

The second issue has to do with the way to treat work requests from host objects. The method associated with this is the jobSlotsIncrease method. A host object can invoke this method to tell the queue that it can take a specified number of jobs. In the current implementation, this method keeps track of the host LOIDs and the specified number of jobs, but has no effect to the operation of the queue object. The reason not implementing this further is that the queue object will have to force a job on a specific host even though the job’s owner did not ask for a specific host. Normally, this decision is left to the system scheduler. A second problem is that an implementation for a particular host may not be available; this will cause a job to fail although there are other implementations available. Finally, there is the question of which job should be placed where, which is actually a question for the scheduler and not for the queue.

6 Acknowledgements

Many thanks to John Karpovich for tirelessly helping me several times with the debugging of the code as well as for his useful ideas. I’d also like to thank professor Marty Humphrey for his help with the JobQueue security issues and finally professor Andrew Grimshaw for his guidance throughout this project.