Chapter 1

GRID RESOURCE MANAGEMENT IN LEGION

Anand Natrajan, Marty A. Humphrey, and Andrew S. Grimshaw
Department of Computer Science, University of Virginia

Abstract Grid resource management is not just about scheduling jobs on the fastest machines, but rather about scheduling all compute objects and all data objects on machines whose capabilities match the requirements, while preserving site autonomy, recognizing usage policies and respecting conditions for use. In this chapter, we present the Grid resource management of Legion, an object-based Grid infrastructure system. We argue that Grid resource management requires not a one-size-fits-all scheduler but an architectural framework that can accommodate different schedulers for different classes of problems.

Keywords: Legion, Grid Scheduling, Resource Management

1. INTRODUCTION

The Legion Project began in late 1993 with the recognition of the dramatic increases in wide-area network bandwidth, truly low-cost processors, and cheap disks looming on the horizon. Given the expected changes in the physical infrastructure, we asked what sorts of applications would people want, and what system software infrastructure would be needed to support those applications. As a result of this analysis, we designed and implemented the Legion Grid Computing system, which is reflective, object-based to facilitate encapsulation, extensible, and is in essence an operating system for Grids. Whereas Globus is a collection of tools from a toolkit [FK99], Legion provides standard operating system services – process creation and control, interprocess communication, persistent storage, security and resource management – on a Grid. By doing so, Legion abstracts the heterogeneity inherent in distributed resources and makes them look like part of one virtual machine. We feel strongly that having a common underlying architecture and set of necessary services built over it is critical for success in Grids, particularly as the line between computational Grids and data Grids blurs [AVD01]. In this sense, the

Legion architecture anticipates the drive to Web Services and the Open Grid Systems Architecture (OGSA) [FKNT02]. There are many papers describing Legion's core architecture and use (e.g., [GW97, GFKH99, NCWD+01, LFH+03]); in this chapter, we focus on the Legion resource management system.

2. OBJECT PLACEMENT IN A GRID

The scheduling process in Legion broadly translates to placing objects on processors. Scheduling is invoked not just for running users' jobs but also to create any object on a Grid, such as a Grid file, a Grid directory, a Grid application or even a Grid scheduler. After an object is created on a processor, it can perform its tasks, for example, respond to read/write calls if the object is a Grid file, or respond to status requests if it is a Grid job. Therefore, object placement is crucial to the design of the Legion run-time system because it can influence an object's run-time behavior greatly. An improper placement decision may impede an object from performing its tasks, for example, because it cannot start on any processor of a given architecture or because the processor is no longer available. Even if a placement decision ensures that an object starts correctly, it does not guarantee that the decision is beneficial to the user. A good placement decision is certain to vary depending on the object being placed and the user's goals as well as resource usage policies.

Determining good object placements in a large distributed, heterogeneous environment, such as a Grid, is difficult because the underlying system is complex, and because object behavior can be influenced by many different factors, such as system status (number, type, and load of components), hardware capabilities (processor, network, I/O, memory, etc.), interactions between objects and object-specific characteristics (size, location of persistent state, estimated performance, etc.). Factors such as security concerns, fault-tolerance objectives and special resource requirements may place hard restrictions on where an object can be placed. These factors are usually expressed as constraints on the placement decision. In general, finding an optimal placement is prohibitively expensive. Several research efforts, such as Utopia [ZWZD93], NOW [ACP+94], Condor [LL90, PL95], Zoom [WASB95], Prophet [Wei95] and others [Cof76, FC90, GY93, WKN+92], have focused on algorithms and systems for near-optimal solutions or optimal solutions to very restricted problem sub-types or user goals [Kar96].

In Legion, we designed a scheduling framework that can accommodate different placement strategies for different classes of applications. In addition to the expected Grid goals of support for heterogeneity and multi-organizational control, the goals included [Kar96]:

- *Support for multiple placement algorithms.* The framework must be flexible enough to be able to incorporate placement algorithms developed by others.

- *Support for user selection of placement.* Users must be permitted to choose the placement approach that best matches their goals.

- *Ease of use.* It should be easy for developers to add new placement algorithms as schedulers in the framework. Additionally, it should be easy for end-users to access the schedulers for performing their tasks. Default placement mechanisms ease the use of the framework for inexperienced users.

- *Ability to cope with uncertain, outdated or partial information.* We expect that in Grids, information may be missing or inaccurate. The scheduling framework in general and the schedulers that are part of it must continue to perform acceptably even when the available system information is less than perfect.

- *Ability to resolve conflicts.* In a system that supports shared objects and resources, conflicts may arise over their use. The framework must have a well-defined resolution behavior in the case of conflicts.

- *Scalability.* The framework should not degrade significantly (or at least degrade gracefully) when the number of processors becomes large or when the requests for placement become more frequent.

- *Low overhead.* The framework should not impose penalties on users who choose not to use it. For users who do choose to use it, the overheads involved in invoking the scheduling process should be small compared to the duration of the task performed.

- *Integration with other Legion services.* As a direct relation to the Legion philosophy of providing an integrated Grid infrastructure, the scheduling framework must cooperate and communicate with other frameworks in Legion, such as those for security and fault-tolerance. The framework must also take into account persistent storage associated with the shared object space in Legion.

In the following sub-sections, we describe the main tasks of the Legion scheduling framework. The goal of this description is not to advocate one placement policy over another. Although we did select a particular placement policy in order to validate (and populate) our framework, we did not and do not claim that placement policy to be optimal or best-suited for all objects.

2.1 Initiating Placement

Placement decisions can be initiated in two ways. In the first case, an object can request the underlying infrastructure explicitly to place (or schedule) other objects with which it must interact. For example, a user (represented as an object in Legion) may request Legion to run a particular job on a particular machine. In this case, the placement initiation is explicit in two senses: the request to place is a direct implication of issuing the run command, and the placement location is provided by the user directly. Legion does not require a user to initiate placement in the latter sense – an undirected run command transfers the burden of finding a placement location from the user to Legion.

In the second case, placement initiation may be implicit, and therefore must be automatic. In other words, an object, say a user, may access another object without realizing that the latter is currently inactive. In this case, Legion will re-activate the second object automatically, which in turn may require it to be placed on an available processor. The processor chosen in this case may or may not be the same processor on which that object was active previously. Implicit or automatic placement initiation occurs frequently in Legion; in order to conserve resources, Legion may deactivate infrequently-used objects. When a subsequent action does require such objects to be available, Legion will re-activate them. Seemingly mundane Grid operations such as logging in, checking the contents of a Grid directory and issuing a run may cause several objects to be re-activated.

2.2 Preparing for Placement

Regardless of how placement is initiated, preparing for placement involves three tasks. The first task is selecting an appropriate scheduler from the framework. To be most effective, the scheduler chosen must factor in criteria that are important to the user [BW96, Ber99, LYFA02]. Since the scheduler itself may require time and CPU cycles to make a decision, its performance and cost must be weighed against its anticipated benefits. This selection may be made automatically by Legion, or may be specified by sophisticated users who choose to indicate which scheduler, or even which processor, must be used. When placement is initiated automatically, there exists a mechanism for indicating which scheduler to use. This mechanism is captured in attributes of *class objects*, which are managers or factories for creating instances of different kinds of objects. For example, when users decide to *port* an application to Legion, they use a tool that essentially creates an application class object in the Grid. A class object may be associated with any of the schedulers available in the Grid. When a user requests that this application be run, the class object consults its attributes, determines the associated scheduler and invokes this scheduler to

perform a placement decision for an instance of the application, namely the run desired by the user.

The second task in preparing for placement is sending the selected scheduler a placement request. Each scheduler may implement a different algorithm and may require different system information for performing placement. Designing a format for the placement request is a non-trivial task; some may argue that if this problem can be solved the problem of designing a general-purpose scheduler for all classes of applications is made much easier. One approach for designing a placement request format is to design a general description language that is flexible and extensible enough to express most placement requests. The challenge with this approach is actually being able to design a scheduler that takes all possible *programs* that can be written in this language and do something useful. Another approach is to develop a standard interface for all schedulers. Unfortunately, a standard interface often implies being able to express only a small subset of functionality possible just so that the more simplistic schedulers can be accommodated. In Legion, we incorporated both approaches. The scheduling framework required conforming to a standard interface, but we also provided a language for querying the database object that collected information on all processors in a Grid so that other schedulers could be written.

The third task is to specify object-specific placement constraints to the scheduler. In Legion, specific placement constraints are specified as attributes on the associated class objects. Typically, these constraints permit specifying either processors that are suited (or unsuited) for this class object or arbitrary attributes that a processor is expected to possess as well in order to qualify as a match. When a class object receives a request to create an instance, it passes these constraints to the scheduler as part of the placement request. The default scheduler we provided with the system takes these constraints into account when making a decision; however, we do not require all schedulers that were part of the framework to take those constraints into account.

2.3 Performing Placement

Placement is performed by the selected scheduler. The scheduler is clearly the heart of the placement process; however, we recognized that others were better at writing schedulers than we were. We provided a framework wherein experts could write schedulers and plug them into our framework. Naturally, in order to validate the framework as well as provide default placement, we wrote our own scheduler. The main tasks of this scheduler are what we expected of any scheduler:

- *Determine the application requirements.* These are available as constraints passed in by the class object.

- *Determine the resources available.* These are available from a database object, called a *collection*, which can be accessed programmatically as well as by using a query language.

- *Invoke a scheduling algorithm.* Invoking the algorithm results in a schedule. For our default scheduler, we employed a random algorithm. Given that this scheduler was a default, we did not expect it to be used frequently. Moreover, given that we could not predict which objects in a Grid would end up using the default scheduler as opposed to a more appropriate scheduler, we felt that *random* was as good or as bad an algorithm as any.

- *Enforce the schedule.* A schedule is of academic interest unless it results in the object actually being created on some suitable processor. As part of the placement process, the framework must ensure that the schedule generated results in object creation, or if it does not, invoke the scheduling algorithm again, perturbing it so that it generates a different schedule. Alternatively, the framework must communicate its failure clearly back to the class object or the user so that other actions can be taken.

2.4 Gathering System Information

A key component in making any placement decision is gathering the necessary information, such as processor types, OS types and versions, attached devices, available disk space, memory and swap size, CPU load, run queue length, security and reservation policies, network bandwidth, network latency, packet drop percentages, etc. Earlier, we alluded to this step, but assumed that the information was already available when preparing for placement. However, when designing a scheduling framework, we had to design mechanisms to ensure that this kind of information was available to the scheduler. We designed a new object, called a *collection* (similar in spirit to MDS [CFFK01]), which functioned as a database for this information. We felt a collection object was necessary so that a scheduler could find reasonably-current information about available processors in one place instead of contacting every processor in a Grid. In turn, either the collection object polled every processor periodically for system information or processors themselves pushed this data into the collection. Collections are generic repositories of object attributes; collections that specifically store information about processors are associated with schedulers in order to aid placement.

2.5 Gathering Application Information

Accurate and detailed information about the behavioral characteristics of different objects can aid scheduling. In Legion, application-specific informa-

tion can be specified in a class object, as we discussed earlier. The class object can be given a set of placement constraints that essentially restricts the processors on which its instances can run. Also arbitrary attributes, typically called *desired host properties*, can be used to restrict the choice of processors; only processors that possess those properties may be selected. Setting these constraints and attributes can be done at any time in the lifetime of the Grid. An additional manner in which the choice of processors can be constrained is by controlling the platforms on which instances of the class object can run. For example, if a user provides only Solaris and Windows binaries for a particular class object, then instances of that class can never be scheduled on, say, a Linux or SGI machine. Furthermore, the user can instruct a particular run – which creates an instance of the class object – to run on any machine of a particular architecture. Thus, Legion provides users with mechanisms to control the scheduling process with application-level information.

3. MECHANICS OF RESOURCE MANAGEMENT

Legion is both an infrastructure for Grids as well a collection of integrated tools constructed on top of this infrastructure. The basic infrastructure enables secure, dataflow-based, fault-tolerant communication between objects. Communicating objects could be diverse resources, such as applications, jobs, files, directories, schedulers, managers, authentication objects (representations of users in a Grid), databases, tools, etc. The Legion scheduling framework acts as a mediator to find a match between placement requests and processors. The scheduling process in Legion is one of negotiation between resource consumers, i.e., autonomous agents acting on behalf of applications or users or objects, and resource providers, i.e., autonomous agents acting on behalf of processors or machines or resources. By providing mechanisms for specifying security and usage policies, resource providers can control who runs what and when on their processors. Likewise, by specifying constraints and choosing schedulers, users can control how their applications run.

The scheduling framework that exists between the providers and consumers attempts to satisfy the expectations of both the providers and the consumers. In the context of Grid resource management, the main contribution of the Legion project is not the algorithm used by the default scheduler, but the surrounding infrastructure that takes security, fault-tolerance, matching, etc. into account for every single object created in a Grid. The infrastructure enables creating objects, jobs included, on any appropriate processor in a Grid, whether across a room or across the globe. The location transparency gained is a deep-rooted part of the Legion philosophy of providing a single virtual machine abstraction for the disparate resources in a Grid.

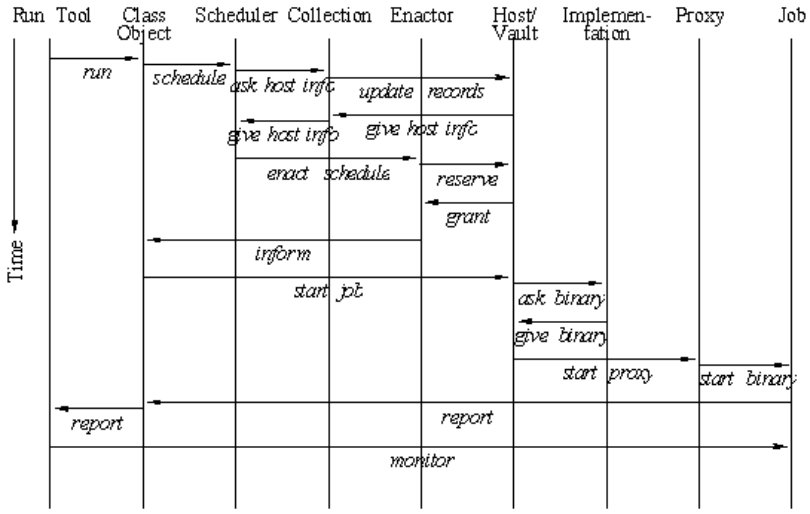

Figure 1.1. Scheduling Process in Legion

The components of the Legion resource management framework are: *class objects*, resource objects (*hosts* and *vaults*), information database objects (*collections*), *scheduler objects*, schedule implementor objects (*enactors*) and *implementation objects* [CKKG99]. Before we examine each component in detail, we will examine their interactions at a higher level (Figure 1). A typical chain of events in a Grid could involve a user initiating a tool to start an instance of an application on a machine. This chain results in a tool object contacting an application class object (to create an instance of this application); which in turn contacts a scheduler (to generate schedules for running this application on a machine); which contacts a collection (to procure information about machines), an enactor (to reserve time on the target machine) and a host object (to start the job on the machine). After the scheduler selects a host object, it contacts the application class object with enough information to start a job instance on the machine.

In the rest of this section, we describe the different objects that participate in resource management. The implementation and interaction of these objects echoes the philosophy we discussed above. However, we regard this set of objects as only one of many possible implementations of that philosophy.

3.1 Class Objects

In Legion, class objects define the type of their instances, as in other object-oriented systems, but in addition are also active entities, acting as managers for their instances. A class is the final authority in controlling the behavior of its instances, including object placement. When a Legion Grid is deployed, a variety of class objects are pre-created already. These class objects can create instances of commonly-used Grid objects such as directories, files, schedulers, collections and even other class objects. Later, other class objects may be added to the Grid. For example, a developer may add a new class object that creates instances of an entirely new kind of object, such as a network object. Alternatively, a developer may refine an existing class object, such as the file class object in order to create a new class object that can create specialized instances, such as matrix or two-dimensional files. Finally, users who port their applications to a Grid typically create, unbeknownst to them, application class objects that are managers for every single run of that application.

All class objects define a *create instance* method, which is invoked during placement initiation. This method may take parameters for an explicit placement or may be called with minimum parameters for an implicit placement. If the placement is explicit, Legion bypasses schedulers, enactors and collections and attempts to start objects on host-vault pairs directly. If the placement is implicit, the scheduling framework is invoked with as much information as available.

3.2 Scheduler and Enactor Objects

A scheduler objects maps requests to resources. As part of this process, the scheduler is given information by the class object about how many instances to create, as well as what constraints apply. Application-specific schedulers may demand and may be supplied with more information about the resource requirements of the individual objects to be created. In addition, a scheduler also requires information about the platforms or architectures on which instances of this class can run. All of this information is procured from the class object.

A scheduler obtains resource information by querying a collection, and then computes a schedule for placing the requested objects. This schedule is passed to an enactor that bears the responsibility of ensuring that the schedule is successful. Each schedule has at least one master version and a list of variant versions. Master and variant versions contain a list of mappings, with each mapping indicating that an instance of the class should be started on the indicated host/vault pair. The master version of a schedule contains the scheduler's best attempt to schedule the requested objects. A variant version differs from a master schedule slightly in terms of the resources selected, representing a poorer scheduling decision to which the enactor can resort if the master fails.

Upon receiving a schedule from a scheduler, the enactor attempts to determine whether the schedule will be successful. In order to do so, it extracts the mappings from the master version, contacts each host/vault pair involved and inquires whether the sub-request on it will be successful. A host/vault pair may choose to reject this sub-request based on its current situation – such a rejection is part and parcel of the negotiation philosophy. If the master version cannot be satisfied because of such rejections, the enactor resorts to the variant versions to schedule successfully. If no version can be made successful, the enactor reports an error and cancels the rest of the scheduling process. If a successful version can be found, the enactor procures reservations from the host/vault (if the host/vault support it) and reports back to the class object with the successful version.

3.3 Collection Objects

A collection is an object that acts as a repository for information describing the state of the resources in a Grid. Each record is stored as a set of Legion object attributes. Collections provide methods to join them and update records. Typically, host and vault objects join collections, although other objects may also join. Members of a collection may supply their attributes in either a *pull* model or a *push* model. In a pull model, the collection takes on the responsibility of polling its members periodically for updates. In a push model, the members periodically initiate updates to the collection (Legion authenticates the member to ensure it is allowed to update the data in the collection). A push model is more appropriate in a scenario in which the members of a collection may lose and regain connectivity with the rest of the Grid. A pull model is more appropriate in a scenario in which we wish to avoid the update implosion of several members updating a single collection.

Users, or their agents such as schedulers, obtain information about resources by issuing queries to a collection. A collection query is a string conforming to some grammar. Currently, a collection is a passive database of static information, queried by schedulers. Collections can be extended to support the ability for users to install code to compute new description information dynamically and integrate it with existing description information for a resource. This capability is especially important to users of the Network Weather Service [WSH99], which predicts future resource availability based on statistical analysis of past behavior.

Another use of collections is to structure resources within the Legion system. Having a few, global collections can reduce scalability. Therefore, collections may receive data from, and send data to, other collections. Making collections be members of other collections gives us the flexibility to have a

collection for each administrative domain and thus achieve hierarchical structuring of Grid resources.

3.4 Host and Vault Objects

Host and vault objects represent two basic resource types in Legion, processors and disk space respectively. Typically, these objects are started on the same machine, but they are not required to be co-located. A host object encapsulates processor capabilities (e.g., a processor and its associated memory) and is responsible for instantiating objects on the processor. Thus, the host object acts as an arbiter for the processor's capabilities. A host object can represent single-machine systems as well as a queue management system such as LoadLeveler [Cor93], NQS [Kin92], PBS [BHL+99] or LSF [Zho92]. A vault object encapsulates storage capabilities (e.g., available disk space) and is responsible for storing the persistent state of objects running on that machine. Every Legion object must have a vault to hold its *Object Persistent Representation* (OPR) . The OPR holds the persistent state of the object, and is used for migration and for shutdown/restart purposes. When requested by an enactor, a host object grants reservations for future service. The exact form of the reservation may vary by implementation of the host object, but it must be non-forgeable tokens; the host object must recognize these tokens when they are passed in with subsequent requests from the class.

There are three broad groups of host/vault functions: reservation management, object management, and information reporting. Reservation functions are used by an enactor to obtain a reservation token for each sub-request in a schedule. When asked for a reservation, a host is responsible for ensuring that its vault is accessible, that sufficient resources are available, and that its local placement policy permits instantiating the object. A host/vault pair is responsible for managing an object during its lifetime. Object management may involve de-activation and re-activation if requested as well as migration. Migrating an object involves collecting its OPR and transmitting it to some other host/vault pair. Hosts and vaults repopulate their meta-data after reassessing their local state periodically. This reassessment is done by invoking local resource management tools or calls on the underlying machine or queuing system. The resultant meta-data, also called attributes, may be pushed into or pulled by a collection object.

3.5 Implementation Objects

Implementation objects may be viewed as representations of the actual binaries required to run objects on a processor. Every object, whether it be a user's job or a Legion object, requires a binary of the appropriate architecture to run on a processor. Registering these binaries with a class object is

the porting process in Legion; the crux of Legion's support for running legacy applications as well as Legion-aware applications is registering binaries with class objects. A class object tracks the implementation objects associated with itself when initiating placement. Therefore, a class that has only Solaris and Windows implementations will never request a schedule containing Linux or SGI machines. When a class receives a viable schedule from an enactor, it communicates with the host/vault objects in order to start objects. Host/vault objects in turn receive the names of the implementation objects for that class, and contact the implementation objects to download the associated binary for running the instance.

Since implementations are objects themselves, they are created in much the same way as any other object. Implementations for hosts and vaults, however, are more basic than implementations of most other class objects. Therefore, host/vault implementations are procured by looking in well-known directories in the Legion installation. Once the host/vault pairs are running, implementations for any other object can be procured from anywhere in the Grid. A minor but interesting point about implementations is that it is perfectly possible and reasonable that the Linux implementation of a particular application class object may actually be started on a Solaris machine. The distinction to remember is that the application's Linux binary happens to be stored on a Solaris machine; therefore the implementation runs on a Solaris machine, but the application binary, when desired, will run only on a Linux machine.

3.6 Proxy Objects

Proxy objects are used to execute legacy application binaries on host and vault pairs and do not play a role in scheduling. However, they are a resource management component because they enable users to employ Legion tools to monitor the progress of a job on a remote machine even though the original job does not respond to Legion requests. Instead, the proxy responds to Legion requests about the status of the job. Since the proxy is not the job itself, it cannot give application-specific status of the job. However, the proxy can provide information such as the name of the machine on which the job runs, the current working directory of the job, the files present in that directory as well as contents of those files at any time, etc.

4. LESSONS LEARNED FROM THE LEGION RESOURCE MANAGEMENT SYSTEM

Several of the key lessons we learned about Grid resource management are captured in the design decisions we incorporated in the scheduling framework. **First, in many ways, scheduling should be treated no differently than the other parts of the Grid infrastructure.** Although not shown in Figure 1,

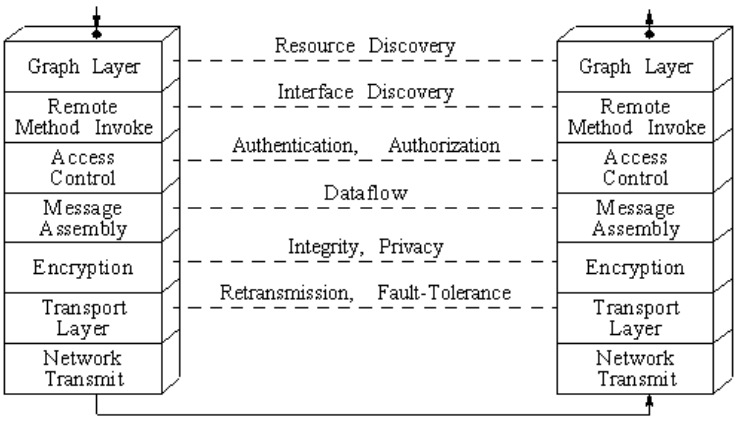

Figure 1.2. Protocol Stack in Legion

every object-to-object communication in the scheduling sequence requires the reliability, efficiency, and privacy/integrity of those object interactions not related to scheduling. We chose to implement the scheduling framework using the same policies and mechanisms available to all object interactions – every communication between any pair of objects must go through the Legion protocol stack (see Figure 2 for an example stack), which involves constructing program graphs, making method invocations, checking authorization, assembling or disassembling messages, encrypting messages, retransmitting messages, and so on. Since every communication goes through such a stack, Legion provides security and fault-tolerance as well as scheduling as part of an integrated resource management framework.

Second, scheduling in Legion is a process of negotiation. Most schedulers view CPU cycles as passive resources waiting to be utilized by the next available job. However, in a multi-organizational framework, a CPU is not necessarily available simply because it is idle. The owner of the CPU – the organization that controls the machine – may impose restrictions on its usage. Therefore, when matching a job to an available CPU, Legion initiates a ne-

gotiation protocol which respects the requirements of the job as well as the restrictions imposed by the CPU owner. In other words, we consider site autonomy an important part of the scheduling, or more correctly, the resource management process. Even if a scheduler selects a particular host for running a job, the host may reject the job based on its current policies. Depending on the implementation, the scheduler may investigate variant schedules or may inform the user of the failure to run the job.

Third, the scheduler can be replaced. Each and every component of a Legion Grid is replaceable. Thus the scheduler in the figure can be replaced by a new one that employs any algorithm of choice. Not just the scheduler, but the toolset that uses the scheduler can be changed as well. For example, we wrote a queue object that uses a similar chain of events to mimic the operation of a queuing system. Also, we wrote a parameter-space tool (similar in spirit to Nimrod [ASGH95]) that can start jobs instantaneously or send them to our queue. A Legion Grid can have multiple schedulers or even multiple instances of a particular scheduler. Applications can be configured to use a specific scheduler. Thus, the Legion Grid resource management framework explicitly allows for different schedulers for different classes of applications. Of course, users can bypass the entire scheduling mechanism, by specifying machines directly or using some non-Legion tool for constructing a schedule for their applications. Bypassing the scheduling mechanism does not mean bypassing security and fault-tolerance, because those functions are at lower levels in the stack. Naturally, if desired, lower levels can be replaced or eliminated as well with the attendant implications.

Fourth, the scheduling infrastructure can be used as a meta-scheduling infrastructure as well. The host object shown in Figure 1 could be running on the front-end of a queuing system or the master node of an MPI cluster. Thus, Legion could be used to select such a host, but subsequent scheduling on the queue or the cluster could be delegated to the queuing system or the MPI system.

When designing the Legion Grid resource management framework, we had a wider definition of resource management than most other distributed systems. We tried to construct a framework within which other parties could write schedulers for different classes of applications. We consciously did not design for only the *classic* applications – long-running, compute-intensive, parallel applications, requiring high performance. Naturally, we did provide a single reference implementation of a scheduler in order to perform resource management on a Legion Grid immediately upon installation. However, we intended this scheduler to be a default – a catch-all scheduler for users who wished to use a Legion Grid as-is. We always intended permitting other schedulers to be part of any Legion Grid.

We did make mistakes in the design of our Grid infrastructure; some of those mistakes were in the scheduling framework. Some of these mistakes are technical, whereas others are psychological. If we were to re-design Legion, here are some lessons we would keep in mind:

People are reluctant to write schedulers. We could not rely on Grid scheduling experts to write schedulers for Legion. Once we learned this lesson, we wrote two new schedulers to complement the default scheduler that already came with every Legion installation. One was a round-robin scheduler for creating instances of files, directories and other objects on a Grid. The round-robin scheduler made quick decisions based on a machine file that was part of its state, thus avoiding expensive scheduling decisions for simple object creation. The second scheduler was a performance-based scheduler for parameter-space studies. This scheduler took CPU speeds, number of CPUs and loads into account for choosing machines on which to run parameter-space jobs.

Writing schedulers deep into a framework is difficult. While we did provide a framework for writing schedulers, a mistake we made was requiring scheduler writers to know too much about Legion internals. Typically, in addition to the scheduling algorithm of interest, a scheduler writer would have to know about schedulers, enactors, hosts, classes and collections; their internal data structures; the data they packed on the wire for several method calls; and Legion program graphs. The effort required to write such a *deep scheduler* was too much. In essence, we had violated one of our own principles: ease of use. Our mistake lay in making Legion easy-to-use for end-users, but not necessarily so for developers. Once we recognized our error, we wrote a *shallow scheduler*, i.e., a scheduler that was about as complex as the default scheduler but did not require knowing too much about Legion internals. The performance-based scheduler for parameter-space studies mentioned earlier is an example of a shallow scheduler. This scheduler is a self-contained Perl script that requires knowing about the collection object (a database of attributes) and the one command to access it. Not having to know Legion details was a significant advantage in the design of this scheduler.

The lesson we learned from this experience was that a high cost of constructing new schedulers is a deterrent to development. Another lesson we learned was that a high cost of running a scheduler can hurt a Grid as well. Put differently, we learned that a quick and acceptable scheduler is much better than a slow but thorough scheduler.

High scheduler costs can undermine the benefits. In Legion, a scheduler is invoked every time an object must be placed on some machine on a Grid.

Given the Legion view of scheduling as a task for placing any object not just a compute object, creating files and directories, implementations and queue services, consoles and classes, all require an intermediate scheduling step. For long, the scheduler that would be invoked for any creation was the default scheduler. While we fully understood the need for different schedulers for different kinds of objects, an artifact of our implementation was that we created only one scheduler – the default one.

The default scheduler's algorithm was complex in two respects. One, the actual processing time took long, especially as the number of machines in a Grid grew. Moreover, the scheduler constructed variant versions for every request just in case the master version did not meet with success. Two, the process invoked methods on too many remote objects. Each method call (or outcall) was a relatively expensive operation. Therefore, even a simple schedule would take too long to generate. Accordingly, we built faster schedulers which perhaps did not find near-optimal and variant schedules, but were far quicker than the default. The round-robin scheduler made fewer outcalls and had a simple algorithm for choosing hosts, but was adequate for scheduling files and directories. Likewise, the shallow scheduler we wrote for performance-based scheduling scheduled parameter-space jobs quickly [NHG02]. It initially spent a few seconds building a schedule, but re-used the schedule for the duration of the application.

Over-complex schedulers are unnecessary. In Legion, we created a sophisticated scheduling framework, but we also implemented this framework in a complicated manner. In particular, splitting the scheduling process from the reservation process (the scheduler and enactor objects respectively), was overkill. The added flexibility this split gave us was never used, and we believe that it will not be used for a while because complex scheduling techniques, such as co-scheduling, that require reservations are useful for a small subset of applications only [SF02]. Too many objects were involved in the scheduling process, making it feel like the process had too many moving parts. The failure of any one object could derail the scheduling process, making it hard to create new objects – files, directories, implementations, jobs, etc. – on a Grid.

5. SUMMARY

In this chapter, we discussed the philosophy and mechanisms of the Legion resource management framework. In Legion, resource management is invoked not just for running jobs but also to place other Grid components, such as files, directories, databases, etc. The key element in resource management is placement, i.e., determining on which machine to start running an object. In Legion, placement is a negotiation process between the requirements of users and the policies of resource managers. This negotiation process is car-

ried out by a scheduler which also employs an algorithm to determine which resources of the available ones is most suited for starting the requested object. Every scheduler in Legion implements the negotiation process, although different schedulers may employ different algorithms.

As Grids mature, diverse resources will be included in Grids and Grid resource management will be central to the working of a Grid. We hope that our experience will serve to guide the design of resource managers. In particular, we believe that the pressing challenges that face the Grid community are the design of rich and flexible resource specification languages in order to match resources with requests, and the design of a framework that can incorporate different solutions for different aspects of Grid resource management.

Acknowledgments

We thank the members of the Legion Team at the University of Virginia for their hard work over the years. In particular, we thank John Karpovich, who developed much of the initial philosophy underlying resource management in Legion. This work was partially supported by DARPA (Navy) contract N66001-96-C-8527, DOE grant DE-FG02-96ER25290, DOE contract Sandia LD-9391, Logicon (for the DoD HPCMOD/PET program) DAHC 94-96-C-0008, DOE D459000-16-3C, DARPA (GA) SC H607305A, NSF-NGS EIA-9974968, NSF-NGS ACI-0203960, NSF-NPACI ASC-96-10920, and a grant from NASA-IPG.

References

[ACP+94] T. E. Anderson, D. E. Culler, D. A. Patterson, et al. A case for now (network of workstations). In *Principles of Distributed Computing*, August 1994.

[ASGH95] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *4 th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC)*, August 1995.

[AVD01] D. C. Arnold, S. Vadhiyar, and J. Dongarra. On the convergence of computational and data grids. *Parallel Processing Letters*, 11(2-3):187–202, September 2001.

[Ber99] F. Berman. *High Performance Schedulers*, chapter 12, pages 279–309. Morgan Kaufmann Publishers, Inc., 1999, 1999.

[BHL+99] 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.

[BW96] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *High Performance Distributed Computing Conference.*, 1996.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing*, August 2001.

[CKKG99] S. J. Chapin, D. Katramatos, J. F. Karpovich, and A. S. Grimshaw. Resource management in legion. *Future Generation Computing Systems*, 15:583–594, October 1999.

[Cof76] E. G. Jr Coffmanm. *Computer and Job/Shop Scheduling Theory*. John Wiley and Sons, 1976. New York.

[Cor93] International Business Machines Corporation. Ibm loadleveler: User's guide. Technical report, IBM, 1993.

[FC90] R. F. Freund and D. S. Conwel. Superconcurrency: A form of distributed heterogeneous supercomputing. *Supercomputing Review*, 3(10):47–50, October 1990.

[FK99] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 11, pages 259–278. Morgan Kauffmann, 1999.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum., 2002.

[GFKH99] A. S. Grimshaw, A. J. Ferrari, F. Knabe, and M. A. Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5), May 1999.

[GW97] A. S. Grimshaw and W. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[GY93] A. Ghafoor and J. Yang. A distributed heterogeneous supercomputing management system. *IEEE Computer*, 26(6):78–86, June 1993.

[Kar96] J. F. Karpovich. Support for object placement in wide area distributed systems. Technical report, CS-96-03, Univ. of Virginia, 1996.

[Kin92] B. A. Kingsbury. The network queueing system (nqs). Technical report, Sterling Software, 1992.

[LFH+03] M. J. Lewis, A. J. Ferrari, M. A. Humphrey, J. F. Karpovich, M. M. Morgan, A. Natrajan, A. Nguyen-Tuong, G. S. Wasson, and A. S. Grimshaw. Support for extensibility and site autonomy in the legion grid system object model. *Journal of Par. and Dist. Computing*, 2003.

[LL90] M. Litzkow and M. Livny. Experience with the condor distributed batch system. In *In IEEE Workshop on Experimental Distributed Sgstems*, 1990.

[LYFA02] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resourceselection framework for grid applications. In

Proceedings of the 11 th IEEE Symposium on HighPerformance Distributed Computing, 2002.

[NCWD+01] A. Natrajan, M. Crowley, N. Wilkins-Diehr, M. A. Humphrey, A. D. Fox, A. S. Grimshaw, and C. L. Brooks III. Studying protein folding on the grid:experiences using charmm on npaci resources under legion. In *10 th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC)*, August 2001.

[NHG02] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. The legion support for advanced parameter-space studies on a grid. *Future Generation Computer Syst.*, 18(8):1033–1052, October 2002. Elsevier Science.

[PL95] J. Pruyne and M. Livny. Parallel processing on dynamic resources with carmi. In *Work. on Scheduling Strategies for Par. Processing, Intl. Par. Processing Symp. (IPPS)*, April 1995.

[SF02] W. Smith and V. Foster, I.and Taylor. Scheduling with advanced reservations. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2002.

[WASB95] R. Wolski, C. Anglano, J. Schopf, and F. Berman. Developing heterogeneous applications using zoom and hence. In *Proceedings of the Work. on Heterogeneous Computing Scientific Computing*, April 1995.

[Wei95] J. Weissman. *Scheduling Parallel Computations in a Heterogeneous Environment*. PhD thesis, Univ. of Virginia, August 1995.

[WKN+92] M. C. Wang, S. D. Kim, M. A. Nichols, R. F. Freund, H. J. Seigel, and W. G. Nation. Augmenting the optimal selection theory for superconcurrency. In *Proc. Work. on Heterogeneous Processing*, pages 13–22, 1992.

[WSH99] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *The Journal of Future Generation Computing Systems*, 1999.

[Zho92] S. Zhou. Lsf: Load sharing in large-scale heterogeneous distributed systems. In *Work. on Cluster Computing*, December 1992.

[ZWZD93] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Soft. Prac. and Exp.*, 23(2), 1993.

Index