The Legion Resource Management System [⋆]

Steve J. Chapin   Dimitrios Katramatos
John Karpovich   Andrew S. Grimshaw

Department of Computer Science
School of Engineering & Applied Science
University of Virginia
Charlottesville, VA 22903–2442
{chapin,dk3x,karp,grimshaw}@virginia.edu

Abstract. *Recent technological developments, including gigabit networking technology and low-cost, high-performance microprocessors, have given rise to metacomputing environments. Metacomputing environments combine hosts from multiple administrative domains via transnational and world-wide networks. Managing the resources in such a system is a complex task, but is necessary to efficiently and economically execute user programs. The Legion resource management system is flexible both in its support for system-level resource management but also in their adaptability for user-level scheduling policies.*

1 Introduction

Legion [6] is an object-oriented metacomputing environment, intended to connect many thousands, perhaps millions, of hosts ranging from PCs to massively parallel supercomputers. Such a system will manage millions to billions of objects, Managing these resources is a complex task, but is necessary to efficiently and economically execute user programs. In this paper, we will describe the Legion scheduling model, our implementation of the model, and the use of these mechanisms to support user-level scheduling. To be successful, Legion will require much more than simply ganging computers together via gigabit channels—a sound software infrastructure must allow users to write and run applications in an easy-to-use, transparent fashion. Furthermore, the software must unite machines from thousands of administrative domains into a single coherent system. This requires extensive support for autonomy, so that we can assure administrators that they retain control over their local resources.

In a sense, then, we have two goals which can often be at odds: users want to optimize factors such as application throughput, turnaround time, or cost, while administrators want to ensure that their systems are safe and secure, and will grant resource access according to their own policies. Legion provides a

[⋆] This work was funded in part by NSF grant CDA9724552, ONR grant N00014-98-1-0454, Northrup-Grumman contract 9729373-00, and DOE contracts DEFG02-96ER25290, SANDIA #LD-9391, and D45900016-3C.

methodology allowing each group to express their desires, with the system acting as a mediator to find a resource allocation that is acceptable to both parties.

Legion achieves this vision through a flexible, modular approach to scheduling support. This modularity encourages others to write drop-in modules and to customize system behavior. We fully expect others to reimplement or augment portions of the system, reflecting their needs for specific functionality. For scheduling, as in other cases, we provide reasonable default policies and allow users and system administrators to customize behavior to meet their needs and desires. Our mechanisms have cost that scales with capability—the effort required to implement a simple policy is low, and rises slowly, scaling commensurately with the complexity of the policy being implemented. This continuum is provided through a substrate rich in functionality that simplifies the implementation of scheduling algorithms.

Before we proceed further, it is important to note a crucial property of our work: we neither desire nor profess to be in the business of devising scheduling algorithms. We are providing enabling technology so that researchers focusing on research in distributed scheduling can build better schedulers with less effort. To paraphrase a popular television commercial in the USA, "We don't make a lot of the schedulers you use. We make a lot of the schedulers you use better."[1]

Section 2 describes the Legion metacomputing system, and Section 3 outlines the resource management subsystem. We develop a Scheduler using Legion resource management in Section 4, and describe other resource management systems for metacomputing in Section 5. Finally, we give concluding remarks in Section 6.

2 Legion

The Legion design encompasses ten basic objectives: site autonomy, support for heterogeneity, extensibility, ease-of-use, parallel processing to achieve performance, fault tolerance, scalability, security, multi-language support, and global naming. These objectives are described in greater depth in Grimshaw et al. [6]. Resource Management is concerned primarily with autonomy, heterogeneity, and performance, although other issues certainly play a role.

The resulting Legion design contains a set of *core* objects, without which the system cannot function, a subset of which are shown in figure 1. These objects are critical to resource management in that they provide the basic resources to be managed, and the infrastructure to support management. Between core objects and user objects lie *service* objects—objects which improve system performance, but are not truly essential to system operation. Examples of service objects include caches for object implementations, file objects, and the resource management infrastructure.

In the remainder of this section, we will examine the core objects and their role in resource management. For a complete discussion of the Legion Core Objects, see [10]. We will defer discussion of the service objects until section 3.

[1] Apologies to BASF.

2.1 Legion Core Objects

Class objects (e.g. HostClass, LegionClass) in Legion serve two functions. As in other object-oriented systems, Classes define the types of their instances. In Legion, Classes are also active entities, and act as managers for their instances. Thus, a Class is the final authority in matters pertaining to its instances, including object placement. The Class exports the create_instance() method, which is responsible for placing an instance on a viable host.[2] create_instance takes an optional argument suggesting a placement, which is necessary to implement external Schedulers. In the absence of this argument, the Class makes a quick (and almost certainly non-optimal) placement decision.

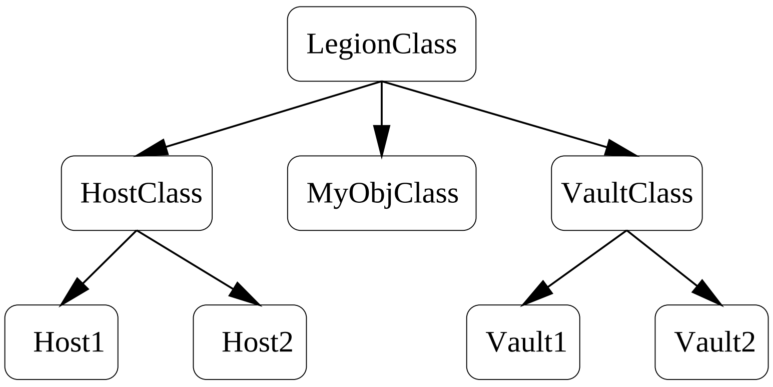

Fig. 1. The Legion Core Object Hierarchy

The two remaining core objects represent the basic resource types in Legion: Hosts and Vaults. Each has a corresponding guardian object class. Host Objects encapsulate machine capabilities (e.g., a processor and its associated memory) and are responsible for instantiating objects on the processor. In this way, the Host acts as an arbiter for the machine's capabilities. Our current Host Objects represent single-host systems (both uniprocessor and multiprocessor shared memory machines), although this is not a requirement of the model. We are currently implementing Host Objects which interact with queue management systems such as LoadLeveler and Condor.

To support scheduling, Hosts grant reservations for future service. The exact form of the reservation depends upon the Host Object implementation, but

[2] When we write "host" we refer to a generic machine; when we write "Host" we are referring to a Host Object.

they must be non-forgeable tokens; the Host Object must recognize these tokens when they are passed in with service requests. It is not necessary for any other object in the system to be able to decode the reservation token (more details on reservation types are given in section 3.1). Our current implementation of reservations encodes both the Host and the Vault which will be used for execution of the object. Vaults are the generic storage abstraction in Legion. To be executed, a Legion object must have a Vault to hold its persistent state in an Object Persistent Representation (OPR). The OPR is used for migration and for shutdown/restart purposes. All Legion objects automatically support shutdown and restart, and therefore any active object can be migrated by shutting it down, moving the passive state to a new Vault if necessary, and activating the object on another host.

Hosts also contain a mechanism for defining event triggers—this allows a Host to, e.g., initiate object migration if its load rises above a threshold. Conceptually, triggers are guarded statements which raise events if the guard evaluates to a boolean true. These events are handled by the Reflective Graph and Event (RGE) mechanisms in all Legion objects. RGE is described in detail in [13,15]; for our purposes, it is sufficient to note that this capability exists.

3 Resource Management Infrastructure (RMI)

Our philosophy of scheduling is that it is a negotiation of service between autonomous agents, one acting on the part of the application (consumer) and one on behalf of the resource or system (provider). This approach has been validated by both our own past history [4,8] and the more recent work of groups such as the AppLeS project at UCSD [1]. These negotiating agents can either be the principals themselves (objects or programs), or Schedulers and intermediaries acting on their behalfs. Scheduling in Legion is never of a dictatorial nature; requests are made of resource guardians, who have final authority over what requests are honored.

Figure 2 shows several different layering schemes that can naturally arise in metasystems. In part (a), the application does it all, negotiating directly with resources and making placement decisions. In part (b), the application still makes its own placement decision, but uses the provided Resource Management services to negotiate with system resources. Part (c) shows an application taking advantage of a combined placement and negotiation module, such as was provided in MESSIAHS [4]. The most flexible layering scheme, shown in part (d), performs each of these functions in a separate module. Without loss of generality, we will write in terms of the fourth layering scheme, with the understanding that the Scheduler may be combined with other layers, thus producing one of the simpler layering schemes. Any of these layerings is possible in Legion; the choice of which to use is up to the individual application writer.

Legion provides simple, generic default Schedulers that offer the classic "90%" solution—they do an adequate job, but can easily be outperformed by Schedulers with specialized algorithms or knowledge of the application. Application writers

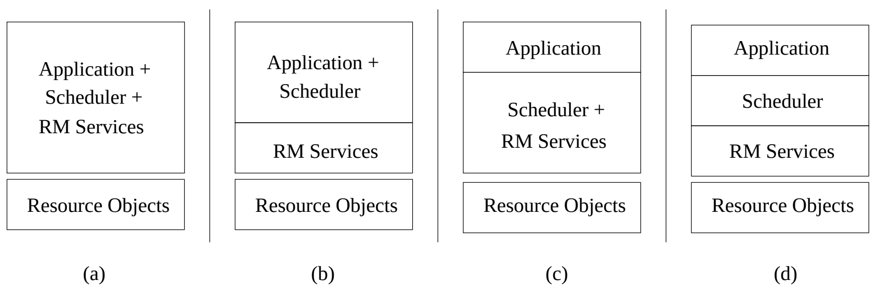

Fig. 2. Choices in Resource Management Layering

can take advantage of the resource management infrastructure, described below, to write per-application or application-type-specific user-level Schedulers. We are working with Weissman's group at UTSA [16] to develop Schedulers for broad classes of applications with similar structures (e.g. 5-point stencils).

Our resource management model, shown in figure 3, supports our scheduling philosophy by allowing user-defined Schedulers to interact with the infrastructure. The components of the model are the basic resources (Hosts and Vaults), the information database (the Collection), the schedule implementor (the Enactor), and an execution Monitor.

The Scheduler is responsible for overall application coordination (recall that we are using option (d) from the layering scheme in figure 2). It decides the mapping of objects (subtasks) to hosts, based on the current system state. The Scheduler can obtain a snapshot of the system state by querying the Collection, or it may interact directly with resources (Hosts and Vaults) to obtain the freshest state information available. Once the Scheduler computers the schedule, it passes the schedule to the Enactor, and the Enactor negotiates with the resources objects named in the schedule to instantiate the objects. Note that this may require the Enactor to negotiate with several resources from different administrative domains to perform co-allocation. After the objects are running, the execution Monitor may request a recomputation of the schedule, perhaps based on the progress of the computation and the load on the hosts in the system.

Figure 3 and the following discussion describe the logical components and steps involved in the scheduling process. Again, this description conforms to our implementation of the interfaces; others are free to substitute their own modules—for example, several components may be combined (e.g. the Scheduler or Enactor and the Monitor) for efficiency. The steps in object placement are as follows:

The Collection is populated with information describing the resources (step 1). The Scheduler queries the Collection, and based on the result and knowledge

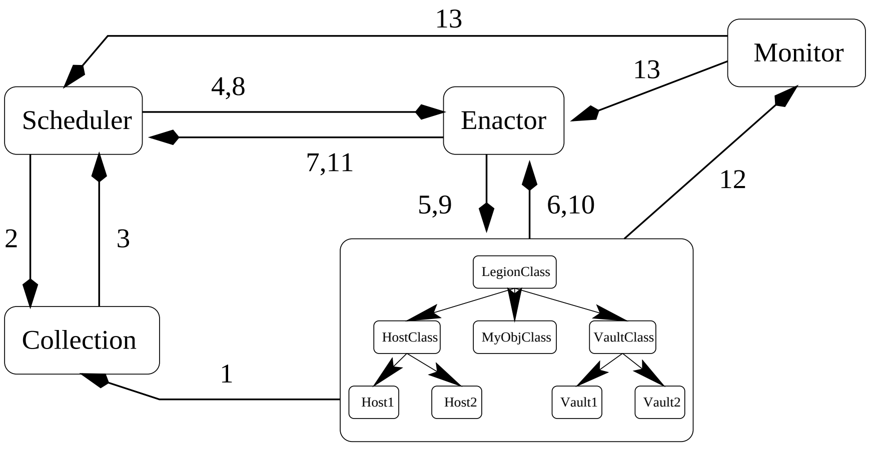

Fig. 3. Use of the Resource Management Infrastructure

of the application, computes a mapping of objects to resources. This application-specific knowledge can either be implicit (in the case of an application-specific Scheduler), or can be acquired from the application's classes (steps 2 and 3). This mapping is passed to the Enactor, which invokes methods on Hosts and Vaults to obtain reservations from the resources named in the mapping (steps 4, 5, and 6). After obtaining reservations, the Enactor consults with the Scheduler to confirm the schedule, and after receiving approval from the Scheduler, attempts to instantiate the objects through member function calls on the appropriate class objects (steps 7, 8, and 9). The class objects report success/failure codes, and the Enactor returns the result to the Scheduler (steps 10 and 11). If, during execution, a resource decides that the object needs to be migrated, it performs an outcall to a Monitor, Which notifies the Scheduler and Enactor that rescheduling should be performed (optional steps 12 and 13).

The remainder of this section examines each of the components in greater detail.

3.1 Host and Vault Objects

The resource management interface for the Host object appears in table 1. There are three broad groups of functions: reservation management, object management, and information reporting.

The reservation functions are used by the Enactor to obtain a reservation token for each subpart of a schedule. When asked for a reservation, the Host is responsible for ensuring that the vault is reachable, that sufficient resources are available, and that its local placement policy permits instantiating the object.

Reservation Management	Process Management	Information Reporting
make_reservation()	startObject()	get_compatible_vaults()
check_reservation()	killObject()	vault_OK()
cancel_reservation()	deactivateObject()	

Table 1. Host Object Resource Management Interface

Host Object support for reservations is provided irrespective of underlying system support for reservations (although the Host is free to take advantage of such facilities, if they exist). For example, the standard UNIX Host Object maintains a reservation table in the Host Object, because the UNIX OS has no notion of reservations. Similarly, most batch processing systems do not understand reservations, and so our basic Batch Queue Host maintains reservations in a fashion similar to the UNIX Host Object.[3] A Batch Queue Host for a system that does support reservations, such as the Maui Scheduler, could take advantage of the underlying facilities and pass the job of managing reservations through to the queuing system. Our real ability to coordinate large applications running across multiple queuing systems will be limited by the functionality of the underlying queuing system, and there is an unavoidable potential for conflict. We accept this, knowing that our Legion objects are built to accommodate failure at any step in the scheduling process.

Legion reservations have a start time, a duration, and an optional timeout period. One can thus reserve an hour of CPU time (duration) starting at noon tomorrow (start time). The timeout period indicates how long the recipient has to confirm the reservation if the start time indicates an instantaneous reservation. Confirmation is implicit when the reservation token is presented with the `StartObject()` call. Our reservations have two type bits: reuse and share. This allows us to build four types of reservations, as shown in table 2. A reusable reservation token can be passed in to multiple `StartObject()` calls. An unshared reservation allocates the entire resource; shared reservations allow the resource to be multiplexed. Thus, the traditional "machine is mine for the time period" reservation has $\mathsf{reuse} = 1$, $\mathsf{share} = 0$, while a typical timesharing system that expires a reservation when the job is done would have $\mathsf{reuse} = 0$, $\mathsf{share} = 1$.

The object (process) management functions allow the creation, destruction, and deactivation of objects (object reactivation is initiated by an attempt to access the object; no explicit Host Object method is necessary). The StartObject function can create one or more objects; this is important to support efficient object creation for multiprocessor systems.

In addition to the information reporting methods listed above, the Host also supports the attribute database included in all Legion objects. In their simplest form, attributes are (name, value) pairs. These information reporting methods for Host Objects allow an external agent to retrieve information describing the

[3] We have Batch Queue Host implementations for UNIX machines, LoadLeveler, and Codine.

one-shot space sharing (share = 0, reuse = 0)	reusable space sharing (share = 0, reuse = 1)
one-shot timesharing (share = 1, reuse = 0)	reusable timesharing (share = 1, reuse = 1)

Table 2. Legion Reservation Types

Host's state automatically (the host's state is a subset of the state maintained by the Host). All Legion objects include an extensible attribute database, the contents of which are determined by the type of the object. Host objects populate their attributes with information describing their current state, including architecture, operating system, load, available memory, etc.

The Host Object reassesses its local state periodically, and repopulates its attributes. If a push model[4] is being used, it will then deposit information into its known Collection(s). The flexibility of Legion object attribute databases allows the Host Object to export a rich set of information, well beyond the minimal "architecture, OS, and load average" information used by most current scheduling algorithms. For example, the Host could export information such as the amount charged per CPU cycle consumed, domains from which it refuses to accept object instantiation requests, or a description of its willingness to accept extra jobs based on the time of day. This kind of information can help Schedulers to make better choices at the outset, thus avoiding the computation of subtly nonfeasible schedules.

The current implementation of Vault Objects does not contain dynamic state to the degree that the Host Object implementation does. Vaults, therefore, only participate in the scheduling process at the start, when they verify that they are compatible with a Host. They may, in the future, be differentiated by the amount of storage available, cost per byte, security policy, etc.

3.2 The Collection

The Collection acts as a repository for information describing the state of the resources comprising the system. Each record is stored as a set of Legion object attributes. As seen in figure 4, Collections provide methods to join (with an optional installment of initial descriptive information) and update records, thus facilitating a push model for data. The security facilities of Legion authenticate the caller to be sure that it is allowed to update the data in the Collection. As noted earlier, Collections may also pull data from resources. Users, or their agents, obtain information about resources by issuing queries to a Collection.

[4] We are implementing an intermediate agent, the Data Collection Daemon, which pulls data from Hosts and pushes it into Collections.

A Collection query is a logical expression conforming to the grammar described in our earlier work [3]. This grammar allows typical operations (field matching, semantic comparisons, and boolean combinations of terms). Identifiers refer to attribute names within a particular record, and are of the form $AttributeName.

```
int JoinCollection(LOID joiner);
int JoinCollection(LOID joiner, LinkedList <Uval_ObjAttribute>);
int LeaveCollection(LegionLOID leaver);
int QueryCollection(String Query, &CollectionData result);
int UpdateCollectionEntry(LOID member, LinkedList<Uval_ObjAttribute>);
```

Fig. 4. Collection Interface

For example, to find all Hosts running with the IRIX operating system version 5.x, one could use the regular expression matching feature for strings and query as follows:[5]

```
match($host_os_name, “IRIX”) and
match(“5\..*”, $host_os_name)
```

In its current implementation, the Collection is a passive database of static information, queried by Schedulers. We plan to extend Collections to support function injection—the ability for users to install code to dynamically compute new description information and integrate it with the already existing description information for a resource. This capability is especially important to users of the Network Weather Service [17], which predicts future resource availability based on statistical analysis of past behavior.

3.3 The Scheduler and Schedules

The Scheduler computes the mapping of objects to resources. At a minimum, the Scheduler knows how many instances of each class must be started. Application-specific Schedulers may implicitly have more extensive knowledge about the resource requirements of the individual objects, and any Scheduler may query the object classes to determine such information (e.g., the available implementations, or memory or communication requirements). The Scheduler obtains resource description information by querying the Collection, and then computes a mapping of object instances to resources. This mapping is passed on to the Enactor for

[5] The match() function uses the Unix regexp() library, treating the first argument as a regular expression. Some earlier descriptions of the match() functions erroneously had the regular expression as the second argument.

implementation. It is not our intent to directly develop more than a few widely-applicable Schedulers; we leave that task to experts in the field of designing scheduling algorithms. Our job is to build mechanisms that assist them in their task.

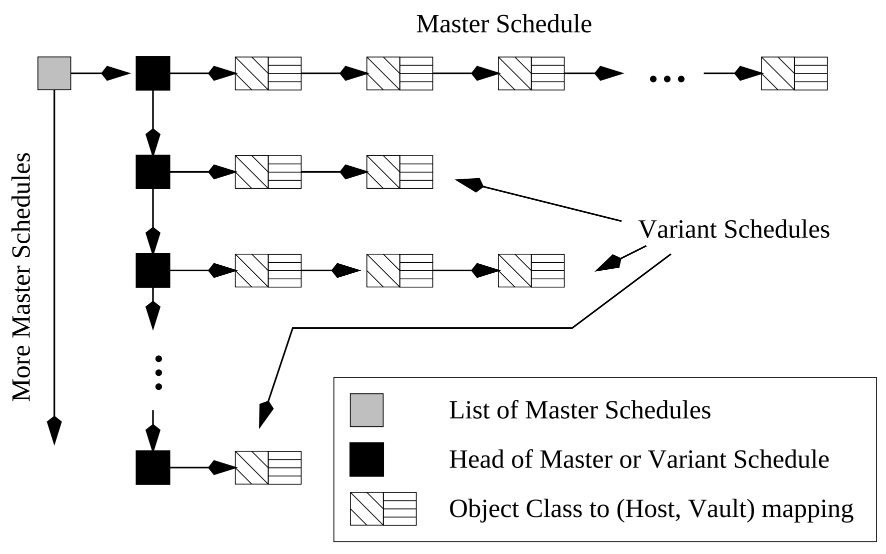

Fig. 5. The Schedule data structure

Schedules must be passed between Schedulers and Enactors. A graphical representation for a Schedule appears in figure 5. Each Schedule has at least one Master Schedule, and each Master Schedule may have a list of Variant Schedules associated with it. Both master and variant schedules contain a list of mappings, with each mapping having the type (Class LOID $\rightarrow$ (Host LOID x vault LOID)). Each mapping indicates that an instance of the class should be started on the indicated (Host, Vault) pair. In the future, this mapping process may also select from among the available implementations of an object as well. We will also support "k out of n" scheduling, where the Scheduler specifies an equivalence class of n resources and asks the Enactor to start k instances of the same object on them.

There are three important data types for interacting with the Enactor: the LegionScheduleFeedback, LegionScheduleList, and LegionScheduleRequestList. A LegionScheduleList is simply a single schedule (e.g. a Master or Variant schedule). A LegionScheduleRequestList is the entire data structure shown in figure 5. LegionScheduleFeedback is returned by the Enactor, and contains the original LegionScheduleRequestList and feedback information indicating whether the reservations were successfully made, and if so, which schedule succeeded.

3.4 The Enactor

The pertinent portion of the Enactor interface appears in figure 6. A Scheduler first passes in the entire set of schedules to the make_reservations() call, and waits for feedback. If all schedules failed, the Enactor may (but is not required to) report whether the failure was due to an inability to obtain resources, a malformed schedule, or other failure. If any schedule succeeded, the Scheduler can then use the enact_schedule() call to request that the Enactor instantiate objects on the reserved resources, or the cancel_reservations() method to release the resources.

```
&LegionScheduleFeedback make_reservations(&LegionScheduleList);
int cancel_reservations(&LegionScheduleRequestList);
&LegionScheduleRequestList enact_schedule(&LegionScheduleRequestList);
```

Fig. 6. Enactor Interface

We have mentioned master and variant schedules, but have not explained how they are used by the Enactor. Each entry in the variant schedule is a single-object mapping, and replaces one entry in the master schedule. If all mappings in the master schedule succeed, then scheduling is complete. If not, then a variant schedule is selected that contains a new entry for the failed mapping. This variant may also have different mappings for other instances, which may have succeeded in the master schedule. Implementing the variant schedule entails making new reservations for items in the variant schedule and canceling any corresponding reservations from the master schedule. Our default Schedulers and Enactor work together to structure the variant schedules so as to avoid reservation thrashing (the canceling and subsequent remaking of the same reservation). Our data structure includes a bitmap field (one bit per object mapping) for each variant schedule which allows the Enactor to efficiently select the next variant schedule to try. This keeps the "intelligence" where it belongs: under the control of the Scheduler implementer.

As mentioned earlier, Class objects implement a create_instance() method. This method has an optional argument containing an LOID and a reservation token. Use of the optional argument allows directed placement of objects, which is necessary to implement externally computed schedules. The Class object is still responsible for checking the placement for validity and conformance to local policy, but the Class does not have to go through the standard placement steps.

3.5 Application Monitoring

As noted earlier, Legion provides an event-based notification mechanism via its RGE model [13]. Using this mechanism, the Monitor can register an outcall with the Host Objects; this outcall will be performed when a trigger's guard evaluates to true. There is no explicitly-defined interface for this functionality, as it is implicit in the use of RGE facilities. In our actual implementation, we have no separate monitor objects; the Enactor or Scheduler perform the monitoring, with the outcall registered appropriately.

4 Examples of Use

We now give an example of a Scheduler that uses our resource management infrastructure. While it does not take advantage of any application-specific knowledge, it does serve to demonstrate some of the flexibility of the mechanisms. We start with a simple random policy, and demonstrate how to build a "smarter" Scheduler based on the simple random policy. This improved Scheduler provides a template for building Schedulers with more complex placement algorithms. We then discuss our plans for building more sophisticated Schedulers with application and domain-specific knowledge.

The actual source code for the default Legion Scheduler was too voluminous to include here. For the sake of brevity and to keep the focus on the facilities provided by Legion rather than the details of a simple random Scheduler, we have presented pseudocode. The source code is contained in release 1.5 of the Legion system, released in January 1999. The current release of the Legion software is available from [9], or by contacting the authors via e-mail at legion@cs.virginia.edu.

4.1 Random Scheduling

The Random Scheduling Policy, as the name implies, randomly selects from the available resources that appear to be able to run the task. There is no consideration of load, speed, memory contention, communication patterns, or other factors that might affect the completion time of the task. The goal here is simplicity, not performance.

Pseudocode for our random schedule generator in figure 7. The Generate_Random_Placement() function is called with a list of classes for which instantiation is desired. The Scheduler iterates over this list, and executes the following steps for each item. First, the Scheduler extracts the list of available implementations from the Class Object it is to instantiate. The Scheduler then queries the Collection for matching Hosts, and picks a matching Host at random. After extracting that Host's list of compatible Vaults from the description returned by the Collection, the Scheduler randomly selects a vault. This (Host, Vault) pair is added to the master schedule. This pair selection is done once for each instance desired for this class.

```
Generate_Random_Placement(ObjectClass list) {
    for each ObjectClass $\mathcal{O}$ in the list, do {
        query the class for available implementations
        query Collection for Hosts matching available implementations
        $k$ = the number of instances of this class desired
        for $i$ := 1 to $k$, do {
            pick a Host $\mathcal{H}$ at random
            extract list of compatible vaults from $\mathcal{H}$
            randomly pick a compatible vault $\mathcal{V}$
            append the target ($\mathcal{H}$, $\mathcal{V}$) to the master schedule
            }
        }
    return the master schedule
}
```

Fig. 7. Pseudocode for random placement

Note that this algorithm only builds one master schedule, and does not take advantage of the variant schedule feature, nor does it calculate multiple schedules. The Scheduler could call this function multiple times to generate additional master schedules. This is not efficient, nor will it necessarily generate a near-optimal schedule, but it is simple and easy. This is, in fact, the equivalent of the default schedule generator for Legion Classes in releases prior to 1.5.

After generating the mapping, the Scheduler must interact with the Enactor to determine if the placement was successful. Although not shown in figure 7, the simple implementation passes a single master schedule to the Enactor via the make_reservations() and enact_schedule() methods, and reports the success or failure of that call back to the object that invoked the Scheduler. No attempt is currently made to generate other placements, although a more sophisticated Scheduler would certainly do so.

4.2 Improved Random Scheduling (IRS)

There are many possible improvements on our random placement algorithm, both for efficiency of calculation and for efficacy of the generated schedule. The improvement we focus on is not in the basic algorithm; the IRS still selects a random Host and Vault pair. Rather, we will compute multiple schedules and accommodate negative feedback from the Enactor. The pseudocode for IRS is in figures 8 and 9.

The improved version generates n random mappings for each object class, and then constructs n schedules out of them. The Scheduler could just as easily build n schedules through calls to the original generator function, but IRS does fewer

```
IRS_Gen_Placement(ObjectClass list, int $n$) {
    for each ObjectClass $\mathcal{O}$ in the list, do {
        query the class for available implementations
        query Collection for Hosts matching available implementations
        $k$ = the number of instances of this object desired
        for $l$ := 1 to $n$, do {
            for $i$ := 1 to $k$, do {
                pick a Host $\mathcal{H}$ at random
                extract list of compatible vaults from $\mathcal{H}$
                randomly pick a compatible vault $\mathcal{V}$
                append the target ($\mathcal{H}$, $\mathcal{V}$) to the list for this instance
            }
        }
    }
    master sched. = first item from each object inst. list
    for $l$ := 2 to $n$, do {
        select the $l^{th}$ component of the list for each object instance
        construct a list of all that do not appear in the master list
        append to list of variant schedules
    }
    return the master schedule
}
```

Fig. 8. Pseudocode for the IRS Placement Generator

```
IRS_Wrapper(ObjectClass list) {
    for $i$ in 1 to SchedTryLimit, do {
        sched = IRS_Gen_Placement(ObjectClass List, NSched);
        for $j$ in 1 to EnactTryLimit, do {
            if (make_reservations(sched) succeeded) {
                if (enact_placement(sched) succeeded) {
                    return success;
                }
            }
        }
    }
    return failure;
}
```

Fig. 9. Pseudocode for the IRS Wrapper

lookups in the Collection. Note also that, because this is random placement, we do not consider dependencies between objects in the placement. A more sophisticated Scheduler would take this into account either when generating the individual instance mappings or when combining instance mappings into a schedule.

The Wrapper function has three global variables that limit the number of times it will try to generate schedules, the number of times it will attempt to enact each schedule, and the number of variant schedules generated per call to the generation function.[6] Again, this is a simple-minded approach to solving the problem, but serves to demonstrate how one could construct a richer Scheduler.

4.3 Specialized Policies

We are in the process of defining and implementing specialized placement policies for structured multi-object applications. Examples of these applications include MPI-based or PVM-based simulations, parameter space studies, and other modeling applications. Applications in these domains quite often exhibit predictable communication patterns, both in terms of the compute/communication cycle and in the source and destination of the communication. For example, we are working with the DoD MSRC in Stennis, Mississippi to develop a Scheduler for an MPI-based ocean simulation which uses nearest-neighbor communication within a 2-D grid.

5 Related Work

The Globus project [5] is also building metacomputing infrastructure. At a high level, their scheduling model closely resembles that of Legion, as we first presented it at the 1997 Legion Winter Workshop [2]. There is a rough correspondence between Globus Resource Brokers and Legion Schedulers; Globus Information Services and Legion Collections; Globus Co-allocators and Legion Enactors; and Globus GRAMs and Legion Host Objects. However, there are substantial differences in realization of the model, due primarily to two features of Legion not found in Globus: the object-oriented programming model and strong support for local autonomy among member sites. Legion achieves its goals with a "whole-cloth" design, while Globus presents a "sum-of-services" architecture layered over pre-existing components. Globus has the advantage of a faster path to maturity, while Legion encompasses functionality not present in Globus. An example of this is in the area of reservations and schedules. Globus has no intrinsic reservation support, nor do they offer support for schedule variation—each task in Globus is mapped to exactly one location.

[6] We realize that the value returned from the generator and passed to the Enactor should be a list of master schedules; we take liberty with the types in the pseudocode for the sake of brevity.

There are many software systems for managing a locally-distributed multicomputer, including Condor [11] and LoadLeveler [14]. These systems are typically Queue Management Systems intended for use with homogeneous resource pools. While extremely well-suited to what they do, they do not map well onto wide-area environments, where heterogeneity, multiple administrative domains, and communications irregularities dramatically complicate the job of resource management. Indeed, these types of systems are complementary to a metasystem, and we will incorporate them into Legion by developing specialized Host Objects to act as mediators between the queuing systems and Legion at large.

SmartNet [7] provides scheduling frameworks for heterogeneous resources. It is intended for use in dedicated environments, such as the suite of resources available at a supercomputer center. Unlike Legion, SmartNet is not intended for large-scale systems spanning administrative domains. Thus, SmartNet could be used within a Legion system by developing a specialized Host Object, similar to the Condor and LoadLeveler Host Objects mentioned earlier. IBM's DRMS [12] also provides scheduling frameworks, in this case targeted towards reconfigurable applications. The DRMS components serve functions similar to those of the Legion RMI, but like SmartNet, DRMS is not designed for wide-area metacomputing systems.

6 Conclusions and Future Work

This paper has described the resource management facilities in the Legion metacomputing environment, including reservations and schedule handling mechanisms. We have focused on the components of the resource management subsystem, presented their functionality, and described the interfaces of each component. Using these interfaces, we have implemented sample Schedulers, including a simple random Scheduler and a more sophisticated, but still random, Scheduler. These sample Schedulers point the way to building more complex and sophisticated Schedulers for real-world applications.

We are in the process of benchmarking the current system so that we can measure the improvement in performance as we develop more intelligent Schedulers. We are developing Network Objects to manage communications resources. The object interfaces will evolve in response to need—as we work with our research partners who are developing scheduling algorithms, we will enrich both the content and capability of the Resource Management Infrastructure and the Legion core objects.

References

1. F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of the 5th International Symposium on High-Performance Distributed Computing (HPDC-5)*, pages 100–111. IEEE, August 1996.
2. S. Chapin and J. Karpovich. Resource Management in Legion. Legion Winter Workshop. http://www.cs.virginia.edu/~legion/WinterWorkshop/slides/Resource_Management/, January, 1997.

3. S. Chapin and E. Spafford. Support for Implementing Scheduling Algorithms Using MESSIAHS. *Scientific Programming*, 3:325–340, 1994. special issue on Operating System Support for Massively Parallel Computer Architectures.
4. S. J. Chapin. Distributed Scheduling Support in the Presence of Autonomy. In *Proceedings of the 4th Heterogeneous Computing Workshop, IPPS*, pages 22–29, April 1995. Santa Barbara, CA.
5. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115-128, 1997.
6. A. S. Grimshaw, Wm. A. Wulf, and the Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.
7. D. Hensgen, L. Moore, T. Kidd, R. Freund, E. Keith, M. Kussow, J. Lima, and M. Campbell. Adding rescheduling to and integrating condor with smartnet. In *Proceedings of the 4th Heterogeneous Computing Workshop*, pages 4–11. IEEE, 1995.
8. J. Karpovich. Support for object placement in wide area heterogeneous distributed systems. Technical Report CS-96-03, Dept. of Computer Science, University of Virginia, January 1996.
9. Legion web page. http://legion.virginia.edu.
10. M. J. Lewis and A. S. Grimshaw. The core legion object model. In *Proceedings of the 5th International Symposium on High-Performance Distributed Computing (HPDC-5)*, pages 551–561. IEEE, August 1996.
11. M. Litzkow, M. Livny, and M. W. Mutka. Condor—A Hunter of Idle Workstations. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 104–111, June 1988.
12. J. E. Moreira and V. K. Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research & Development*, 41(3), 1997.
13. A. Nguyen-Tuong, S. J. Chapin, and A. S. Grimshaw. Designing generic and reusable orb extensions for a wide-area distributed system. In *High-Performance Distributed Computing, poster session*, July 1998.
14. A. Prenneis, Jr. Loadleveler: Workload management for parallel and distributed computing environments. In *Proceedings of Supercomputing Europe (SUPEUR)*, October 1996.
15. C. L. Viles, M. J. Lewis, A. J. Ferrari, A. Nguyen-Tuong, and A. S. Grimshaw. Enabling flexiblity in the legion run-time library. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 265–274, June 1997.
16. J. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1(1), 1998.
17. R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th International Symposium on High-Performance Distributed Computing (HPDC-6)*, August 1997.