Final Report
Grant No. DE-FG02-96ER25290
March 1, 1996 – September 30, 1999

LEGION CORE OBJECT MODEL

Submitted to:
Department of Energy
Acquisition and Assistance Group
Chicago Operations Office
9800 South Cass Avenue
Argonne, IL 60439

Attention:
Fred Sienko, Contracting Officer

Submitted by:
Andrew S. Grimshaw
Professor

SEAS Report No. UVA/527601/CS01/101
September 2000

PROCESSED FROM BEST AVAILABLE COPY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF
ENGINEERING & APPLIED SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903
UNIVERSITY OF VIRGINIA  
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 600. There are 160 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Applied Mechanics, Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 17,000), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Final Report for DOE

This report summarizes the Legion Research Project's work, as funded by DoE grant DE-FG02-96ER25290. Our proposal, #33086, submitted December 7, 1995, laid out a series of goals, all of which we have accomplished within our proposed timeline.

Our proposed goal was to build the Legion system's core components, language support, federated file system, miscellaneous utilities, adopt existing applications to run in Legion, and document the system. In the proposal we wrote that "We will judge Legion similarly, by the reviewed publications, by community acceptance, and by our ability to achieve our design objectives." So how have we done? We have met all of our objectives. We have built a robust, well-documented, implementation that is used around the world in academia, government, and industry. The Legion project is one of two well-known grid projects in the world. We have published a number of papers and given dozens of talks around the world. And, the technology has been successfully transferred to industry where it is now being supported at commercial sites.

Let's look at each of these in more detail.

Robust, well-documented implementation

As of August 2000, Legion can be downloaded from Legion web site. We have packages for eight platforms, as source code or executable binary files. The source code contains approximately 360,000 lines of heavily templated C++. The current platforms are:

<table>
<thead>
<tr>
<th>Platform</th>
<th>OS</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>Linux</td>
<td>Linux is our development platform.</td>
</tr>
<tr>
<td></td>
<td>FreeBSD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windows NT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Windows 2000</td>
<td></td>
</tr>
<tr>
<td>Sparc</td>
<td>Solaris</td>
<td></td>
</tr>
<tr>
<td>SGI</td>
<td>IRIX</td>
<td>We support three separate builds for this platform: n32, o32, and n64.</td>
</tr>
<tr>
<td>HP</td>
<td>HPUX</td>
<td></td>
</tr>
<tr>
<td>Alpha</td>
<td>Linux and DEC Unix</td>
<td></td>
</tr>
<tr>
<td>IBM RS6000</td>
<td>AIX</td>
<td></td>
</tr>
<tr>
<td>Cray T3E*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Complete documentation is also available, including hundreds of pages of manuals for users, system administrators, and developers in both hard and soft copy. On-line documentation includes the traditional command-line Unix "man" pages, as well as Web and HTML support. Training materials such as tutorials and sample programs have also been developed. User support includes a Legion-help e-mail list read by the Legion staff, as well as workshops and on-line tutorials.

* Previous releases included the Cray C90/J90 and T90. Due to lack of demand, however, we are not currently supporting these platforms.
In terms of "robustness", the mean time to failure of our main network, vanet, is over two months. The system survives frequent physical host crashes, file server crashes, and many other common failure modes, all of this while serving a community of dozens of users, including several very active users scattered around the country.

Legion Networks
Our premier Legion network is vanet. Vanet is an NPACI-based network run out of the University of Virginia. The network contains hosts around the world, from Tokyo to Amsterdam. In the United States, vanet consists of hosts at
- SDSC (a 64-way Sun, a 128-way SP, and a new addition, the SDSC teraflops machine)
- Caltech (128 way V-Class)
- Georgia Tech (96 processor cluster)
- UC Berkeley (NOW)
- NCSA (SGI Origins)
- Indiana University (64-way SGI Origin)
- University of Michigan (96-way IBM SP-2), and
- University of Virginia (370 processor cluster).

The graphic below from our status monitor shows the US component.

In addition to vanet, there is a DoD HPCMOD network consisting of supercomputers at three of the four DOD MSRCs, a NASA IPG network, and an intranet at Boeing.
**Demonstration applications**

We have a set of demonstration applications that we update as we add new applications. Current projects involve parameter-space studies (e.g., CNS and Gnomad), CHARMM, ADF, Amber, Feature, and VASP.

We have been adding on new projects on a regular basis. For example, at UVa we recently started working with Greg Lewin, an engineering graduate student studying insect flight, and Mike Sierk, a Biophysics grad student using X-ray crystallography and NMR spectroscopy to study the HIV-1 Rev protein structure. We have ongoing projects with projects involving molecular dynamics, catalyst particles, and protein databases.

An example of a large-scale MPI application uses the Navy’s BT-MED. To illustrate our capabilities we recently demonstrated a parallel, fault-tolerant, wide-area execution of BT-MED. BT-MED is a 2D barotropic ocean code written in Fortran. It has 2D data decomposition and uses MPI. We executed the code at four sites (SDSC, NCSA Indiana University, and UVa) and on three platforms (IBM, SGI, Intel). One eighth of the tasks were placed at SDSC on the IBM SP-2, one eighth of the tasks at Indiana, one quarter at NCSA on an Origin 2000, and half of the tasks on an Intel Linux cluster at Virginia. This illustrates cross-architecture, cross-site, computing (note also the *endian* differences).

On another run of the same application we killed a task approximately 60 iterations into the computation. The MPI-Legion fault-tolerance libraries detected the failure, rolled the computation back to last checkpoint, rescheduled and restarted the tasks, and restarted the application from the checkpoint. The computation then proceeded normally to the end.

Finally, we demonstrated the use of our parallel 2D file objects. Each worker periodically writes out variables of interest for visualization to a 2D-file object. The file object was configured with a block decomposition, with 12 sub-files used. The BT-MED workers were in Virginia; the 2D file was located at NCSA.
A partial list of applications and areas includes:

Digital Context Creation
- RenderGrid with BMRT

Aerospace
- Design Explorer (Boeing)
- Overflow (Boeing - NASA)
- Flapper (UVa - Lewin)

Biochemistry & molecular science
- complib (UVa - Pearson/Grimshaw)
- FASTA, Smith-Waterman
- gnomad (Stanford - Altman/Williams)
- feature (Stanford - Altman)
- CHARMM (Scripps - Brooks)

Astronomy
- Hydro code (Hawley/Holcomb)

Materials Science
- DSMC - Direct Simulation Monte Carlo (UVa - Wadley/Beekwilder)
- Large scale molecular dynamic (NAVO-LSU)

Information Retrieval
- PIE (French & Viles)

Climate, Weather, Ocean
- BT-MED (NAVO - Piechek)
- MM5 (NCAR)
- NLOM-COAMPS - (NAVO - Bettencourt)

Neuroscience
- Biological scale simulations of a mammalian neural network (UVa - Levy)

Publications:


In addition the Legion has been the topic of a number of invited presentations around the world. These include:

"Object-Based Grids," Grid Forum, July 2000, Seattle, WA.


“Computational Grids,” PPoPP, May 1999, Atlanta, GA.


“Legion--A View From 50,000 Feet,” Dec 1998, Georgia Tech, GA.


“Legion--A View From 50,000 Feet,” Sept 1998, Purdue University, IN.

“Legion--An Applications Perspective,” NPACI Summer Institute, Aug 1998, La Jolla, CA.

“Legion--A View From 50,000 Feet,” Symposium on Global Distributed Computing, Waseda University, June 1998, Tokyo, Japan.


“Metacomputing Support at NAVO and Other MSRCs,” Army Research Lab, May 1998, Aberdeen, MD.


“Legion--A View From 50,000 Feet,” Feb 1998, Ohio State University, Columbus, OH.
“Metasystems: What are they and what use are they?” *Signal and Image Processing Forum*, Feb 1998, Aberdeen, MA.

“Legion--A View From 50,000 Feet,” IBM Research, Nov 1997, Yorktown Heights, NY.

“Legion--A View From 50,000 Feet,” University of Texas San Antonio, Oct 1997, San Antonio, TX.

“Legion--A View From 50,000 Feet,” Rice University, Oct 1997, Houston, TX.

"Legion--A View From 100,000 Feet," *Workshop on Seamless Computing*, ECMWF, Sept 1997, Reading, UK.

"Legion--A View From 50,000 Feet," NASA AMES, Aug 1997, Mountain View, CA.

"Legion--A View From 50,000 Feet," HP Research Labs, Aug 1997, Palo Alto, CA.

"Metasystems: The WAVE of the Future (Wide Area Virtual Environment),” *NPACI Summer Institute*, Aug 1997, La Jolla, CA.


“Meta-applications in Legion,” Keynote address at *MAPINT 1997*, June 1997, Dayton, OH.

Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems

Andrew S. Grimshaw, Michael J. Lewis, Adam J. Ferrari, John F. Karpovich
{grimshaw | mlewis | ferrari | jfk3w}@cs.virginia.edu
Department of Computer Science, University of Virginia

Keywords: Distributed computing, wide-area, distributed objects, metasystems, middleware, site autonomy.

Abstract

The Legion system defines a software architecture designed to support metacomputing, the use of large collections of heterogeneous computing resources distributed across local- and wide-area networks as a single, seamless virtual machine. Metasystems software must be extensible because no single system can meet all of the diverse, often conflicting, requirements of the entire present and future user community, nor can a system constructed today take best advantage of unanticipated future hardware advances. Metasystems software must also support complete site autonomy, as resource owners will not turn control of their resources (hosts, databases, devices, etc.) over to a dictatorial system. Legion is a metasystem designed to meet the challenges of managing and exploiting wide-area systems. The Legion virtual machine provides secure shared object and shared name spaces, application adjustable fault-tolerance, improved response time, and greater throughput. Legion tackles problems not solved by existing workstation-based parallel processing tools, such as fault-tolerance, wide-area parallel processing, interoperability, heterogeneity, security, efficient scheduling, and comprehensive resource management. This paper describes the Legion run-time architecture, focusing in particular on the critical issues of extensibility and site autonomy.

1. The Legion project is partially supported by NFS CDA-9724552, DARPA contract #N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C, and DARPA (GA) SC H607305A
1. Introduction

Recent technical advances and increasingly widespread deployment of local- and wide-area high-speed networks provide new opportunities for applications developers. Just to name a few, improved network capabilities enable increasingly sophisticated collaboration tools, improved data and resource sharing, both within and across organizations, and larger scale parallel and distributed applications that may run on geographically dispersed machines. The challenge facing the computer science community is to provide software abstractions that can combine the diverse resources available into a single usable entity. We call this metasystems software—software above the physical resources and below end-user applications. Without metasystems software, the task of constructing applications that exploit the full potential of these new networks will be difficult or impossible.

We believe that, above all else, metasystem software must be extensible and must support site autonomy. It must be extensible because no single system implementation can meet all of the diverse, often conflicting, requirements of the entire present and future user community, nor can a system constructed today best take advantage of unanticipated future hardware advances. Metasystem software therefore must provide users, applications developers, and resource owners with the ability to reshape the software infrastructure as needed in a consistent, orderly manner. Site autonomy must be supported because resource owners will not turn their resources (hosts, databases, devices) over to a metasystem without being able to enforce their own policies. Metasystems must instead allow resource owners to fully control their resources (e.g., to determine who can use their resources, how and when they can be used, how much it will cost, etc.).

Legion, developed at the University of Virginia, is a metasystem designed to meet the challenges of managing and exploiting wide-area systems. The hardware base for Legion
consists of workstations, vector supercomputers, and parallel supercomputers connected by a variety of networks. The system is designed support the illusion of a single virtual machine, providing users and applications developers with secure shared object and shared name spaces. Legion tackles a wide range of distributed systems problems, including application-adjustable fault-tolerance, wide-area parallel processing, interoperability, heterogeneity, security, efficient scheduling, and comprehensive resource management. No other existing parallel processing tool supports such a broad range of services.

Legion includes a run-time system, several high level programming languages with corresponding Legion-aware compilers, and Legion-targeted versions of popular packages like PVM and MPI. Thus, Legion allows users to write programs in several different high-level languages, while the run-time system transparently creates, schedules, and utilizes distributed objects to execute the programs. To support the diverse policies and priorities of users, developers, and resource providers, Legion defines the mechanisms for system-level services such as object creation, naming, and migration but does not mandate the implementation of these services or policies for their use.

The primary purpose of this paper is to describe the Legion interobject run-time architecture. Unfortunately we cannot cover all of the important issues in one paper. Many of the most important issues not discussed here are covered in other publications, including the security model [35], scheduling [22], fault-tolerance [30], and the intra-object run-time library architecture [34]. Section 2 discusses our high-level objectives, design constraints, and philosophy and explains the motivation for the Legion object model and architecture. Section 3 defines key Legion concepts used throughout the remainder of the paper. Section 4 introduces the core Legion system elements at a high level and demonstrates how they work together
through a simple narrative example. Section 5 presents the interface and functionality of the core system objects in detail, how they combine to implement basic services, and some examples of how programmers may augment or replace different parts of our implementation. We conclude with related work (Section 6) and a summary (Section 7).

2. Legion Objectives, Constraints and Philosophy

2.1 Objectives

Realizing our vision of a wide-area metasystem is not a trivial matter. We have distilled ten design objectives that are essential to the success of the project. Each is discussed briefly below.

- **Site autonomy**: Legion will be composed of resources owned by many organizations, which properly insist on retaining control over their resources. For each resource the owner must be able to limit or deny use by particular users, specify when it can be used, etc. An important aspect of autonomy is implementation selection. Sites must be able to choose or re-write the implementation of each Legion component to best suit their needs. For example, a site may trust the security mechanisms of one particular implementation over those of another.

- **Extensible core**: Legion must be flexible enough to suit the wide variety of current user demands and capable of evolving to meet unanticipated future needs. Therefore, we feel that mechanism and policy must be realized via replaceable and extensible components, including and especially those of our core system components. This model facilitates development of improved implementations that provide value-added services or site-specific policies, while enabling Legion to adapt over time to a changing hardware and user environment.

- **Scalable architecture**: Because Legion's goal is to construct metasystems with millions of hosts and objects, it must have a scalable software architecture. That is, the system must be fully distributed with no centralized structures or servers.
- **Easy-to-use, seamless computational environment**: Legion must mask the complexity of its hardware environment and simplify the communication and synchronization involved in parallel processing. Machine boundaries, for example, should be invisible to users.

- **High-performance via parallelism**: We believe that metasystems must support high-performance parallel applications with large degrees of parallelism. Therefore, Legion must keep overhead low and support a wide variety of parallel processing models, including arbitrary combinations of task and data parallelism.

- **Single, persistent object space**: The lack of a single name space for accessing data and resources is one of the most significant obstacles to wide-area distributed and parallel processing. The current multitude of disjoint name spaces greatly impedes developing applications that span sites. All Legion objects must be able to transparently access (subject to security constraints) any other Legion object without regard to location or replication.

- **Security for users and resource owners**: Attempting to patch security on as an afterthought (as some systems are attempting today) is a fundamentally flawed approach. We believe that security must be built firmly into the foundation of a metacomputing system. We also believe that no single security policy is perfect for all users. Therefore, we must provide mechanisms that allow users and resource owners to select policies that fit their security and performance needs and meet their local administrative requirements.

- **Management/exploitation of resource heterogeneity**: Legion must support interoperability between heterogeneous hardware and software platforms. It should also exploit heterogeneity when possible by matching applications to the best suited resources (e.g., vector codes).

- **Multiple language support and interoperability**: Legion must be able to support the integration and interoperability of application components written in a variety of source
languages. We feel that interoperability also dictates that we support legacy codes and work with emerging standards such as CORBA [31] and DCE [26], wherever possible.

- **Fault-tolerance:** In a large-scale metasystem, resource failures (hosts, communication links, disks, etc.) will be commonplace. Therefore, the Legion system itself must deal with failures, through system object fault-tolerance and dynamic system reconfiguration, as well as provide mechanisms to support a wide range of user application fault-tolerance needs.

Though we focus primarily on technical issues in this paper, we recognize that there are also important political, sociological, and economic challenges in developing a metasystem, such as developing a scheme to encourage the participation of resource-rich sites while discouraging free-riding by others.

### 2.2 Constraints

The objectives listed above are framed by several practical constraints that restrict our design.

- **Cannot change host operating systems.** Organizations will not permit their machines to be used if their operating systems must be replaced. Our experience with Mentat [15] indicates, though, that building a metasystem on top of host operating systems is a viable approach.

- **Cannot change network interface.** Just as we must accommodate existing operating systems, we assume that we cannot change the network resources or the protocols in use.

- **Cannot require Legion to run in privileged mode.** To protect their objects and resources, Legion users and sites will require Legion software to run with the lowest possible privileges.

### 2.3 Philosophy

Our overall objective is to design a metasystem that will be suitable to as many users and for as many purposes as possible. One thing is clear: a rigid system design—one in which policies are limited, trade-off decisions are pre-selected, or all semantics are pre-determined and hard-
coded—will not achieve this goal. Indeed, if we were to dictate a single system-wide solution to almost any of the ten technical objectives, we would preclude large classes of potential users and uses. Therefore, we designed Legion to allow users and programmers the greatest flexibility in their applications’ semantics, resisting the temptation to dictate solutions to many system functions. Users are able, whenever possible, to select both the kind and the level of functionality, and to make their own trade-offs between function and cost.

This philosophy is manifested in the system architecture. The Legion object model specifies the functionality but not the implementation of the system’s core objects; the core system therefore consists of extensible, replaceable components. Legion provides default implementations of the core objects, although users are not obligated to use them. Instead, we encourage users to select or construct object implementations that meet their specific needs.

3. Legion Object Model and Key Legion Concepts
Legion is an object-oriented system comprising independent, logically address space disjoint objects, which communicate with one another via method invocation. The fact that Legion is object-oriented does not preclude the use of non-object-oriented languages or non-object-oriented implementations of objects. In fact, Legion supports objects written in traditional procedural languages such as C and Fortran in addition to object-oriented languages such as C++, Java, and the Mentat Programming Language (MPL) [15].

Method calls are non-blocking and may be accepted in any order by the called object. Each method has a signature that describes its parameters and return values (if any). The complete set of signatures for an object describes that object’s interface, which is determined by its class. Legion class interfaces are described in an Interface Description Language (IDL), two

---

1. MPL [15] is a parallel dialect of C++ in which classes may be denoted as Mentat classes, whose instances are address-space disjoint, and whose member functions may be executed in parallel (see also Section 3.3).
of which are currently supported—the CORBA IDL and MPL.

Each Legion object belongs to a class and each class is itself a Legion object. All objects export a common set of object-mandatory member functions (such as `deactivate()` and `getInterface()`) that are necessary to implement the core Legion services. Class objects export an additional set of class-mandatory member functions that enable them to manage their instances (such as `createInstance()` and `deleteInstance()`).

Much of the Legion object model's power comes from the role of Legion classes; much of what is usually considered system-level responsibility is delegated to user-level class objects. For instance, Legion classes are responsible for creating and locating their instances and for selecting appropriate security and object placement policies. The core Legion objects provide mechanisms that allow user-level classes to implement their chosen policies and algorithms. The philosophy of encapsulating system-level policy in extensible, replaceable class objects, supported by the set of primitive operations exported by the Legion core objects (described in detail in Section 5), effectively eliminates the danger of imposing inappropriate policy decisions and opens up a much wider range of possibilities for the application developer.

### 3.1 Naming and Binding

Legion objects are identified through a three-level naming hierarchy, depicted in Figure 1. At the highest level, objects are identified by user-defined text strings called context names. These user-level context names are mapped by a directory service called context space to unique location-independent system-level names called Legion object identifiers (LOIDs). For direct object-to-object communication, LOIDs must be bound to low-level object addresses (OA) that are meaningful within the context of the transport protocol used for communication. The process by which LOIDs are mapped to object addresses is called the Legion binding process (see Figure 1).
FIGURE 1. The three-level Legion naming hierarchy. Context names are convenient user-defined textual identifiers. These map to Legion object identifiers (LOIDs): system-wide unique, location-transparent

LOIDs: Every Legion object is assigned a unique and immutable LOID upon creation. The LOID identifies an object to various services (e.g., method invocation). The basic LOID data structure consists of a sequence of variable length binary string fields. Four of these fields are reserved by the system. The first three play a key role in the LOID-to-object address binding mechanism: Field 0 is the domain identifier, used in the dynamic connection of separate Legion systems; Field 1 is the class identifier, a bit string uniquely identifying the object’s class within its domain; Field 2 is an instance number that distinguishes the object from other instances of its class. LOIDs with an instance number field of length zero are defined to refer to class objects. Field 3 is reserved for security purposes. Specifically, this field contains a public key for encrypted communication with the named object. The format of the LOID is left unspecified beyond these four reserved fields. New LOID types can be constructed to contain additional security information, location hints, and other information in the additional available fields.

Object Addresses: Legion uses standard network protocols and communication facilities of host operating systems to support interobject communication. To perform such communication Legion converts location-independent LOIDs into location-dependent communication system-level OAs through the Legion binding process. An OA consists of a list of object address elements and an address semantic field, which describes how to use the list. An OA element contains two parts, a 32-bit address type field indicating the type of address contained in the OA and the address itself, whose size and format depend on the type. The address semantic field is intended to express various forms of multicast and replicated communication. Our current
implementation defines one OA type, consisting of a single OA element containing a 32-bit IP address, 16-bit port number, and 32-bit unique id (to distinguish between multiple sessions that reuse a single IP/port pair). This OA is used by our UDP-based data delivery layer [16].

**Bindings:** Associations between LOIDs and OAs are called *bindings*, and are implemented as three-tuples. A binding consists of a LOID, an OA, and a field that specifies the time at which the binding becomes invalid (including never). Bindings are first-class entities that can be passed around the system and cached within objects.

**Object States:** In a typical Legion system the number of objects may be orders of magnitude larger than the number of processors. Since it is unreasonable to require an active process for every object in the system, Legion objects are persistent and alternate between two states, *active* or *inert*. When active, an object runs as a process (controlled by a Legion *host object*, Section 5.2) and can be communicated with via its OA. When inert, an object exists only in persistent storage (controlled by a Legion *vault object*, Section 5.3), is described by an *object persistence representation* (OPR), and is located using an *object persistence address* (OPA). Throughout their lifetime, objects can be moved between active and inert states.

Each Legion object is associated with its own OPR in which persistent state is stored. Each Legion object implements an internal `saveState()` method that is called to store the object's state into its OPR (prior to becoming inert). Similarly, each object defines an internal `restoreState()` method, which is called immediately after reactivation to recover state from the OPR. The OPA of an inert object is analogous to the OA of an active object and is used to gain direct access to an OPR. Typically, an OPA is a file name or a set of file names, meaningful only to the controlling Legion vault object and to the associated object.

---

2. Our current implementation maps each active object to its own process. However, the Legion model does require one process per object, so future implementations may multiplex objects to processes.
3.2 Attributes

Legion attributes provide a general mechanism to allow objects to describe themselves to the rest of the system. An attribute is an $n$-tuple containing a tag and a list of values; the tag is a character string, and the values contain data that varies by tag. Attributes are stored as part of the state of the object they describe, and can be dynamically retrieved or modified via object-mandatory functions. In general, programmers can define an arbitrary set of attribute tags for their objects, although certain types of objects are expected to support certain standard sets of attributes. For example, host objects are expected to maintain attributes describing the architecture, configuration, and state of the machine(s) they represent.

3.3 Legion Programming

At its lowest level, Legion defines a message format for interobject communication and a set of services and protocols for managing, naming, and manipulating objects. Because Legion is essentially a specification, many implementation strategies are possible for its basic services. We have employed our own implementation strategy for Legion, creating the Legion Run-Time Library (LRTL) [34] and a set of core object implementations. Therefore, one way to develop Legion programs is to use the LRTL routines directly.

However, programming with a fairly low-level library like the LRTL can be tedious and error-prone. A much better application development model is to use a suitable higher-level language or library interface that is layered on the LRTL services. To support this notion, we have developed LRTL-targeting versions of several existing programming environments including the Mentat Programming Language (MPL) [15], PVM [13], MPI [19, 27], and CORBA [31]. We have also developed a specialized programming interface to support Fortran programmers called Legion Basic Fortran Support (BFS) [10].
4. An Illustrative Example

So far, we have described the Legion object model and some key features, such as naming and persistence, but we have not yet discussed the design of the fundamental object creation and binding processes. In Legion, these services are supported by a cooperating set of core objects. However, before delving into a detailed examination of the interfaces and designs of each core system object, we feel it is useful to present a simple example that illustrates at a high level the roles and interrelationships of these objects. In this section we describe how Legion implements a simplified RPC-style interaction between two Legion objects, Caller and Callee, and introduce the basic functionality of the Legion core objects. Section 5 describes these objects in much more detail and discusses alternative policies and implementations that are possible under the architecture and object model.

Suppose that a Legion object, Caller, wishes to invoke member function \texttt{func()} on another Legion object, Callee. In order to communicate with Callee, Caller must confirm that Callee exists and is active, and must resolve Callee’s OA. All of this is accomplished within the framework of the Legion binding process, described in the following sections.

4.1 Legion Binding Mechanism

In order to carry out the message passing associated with the desired method invocation, Caller must bind Callee’s LOID to Callee’s current OA. This process is called the Legion binding mechanism, and is depicted in Figure 2.

If Caller and Callee have communicated prior to the current method invocation, Caller may already have a binding for Callee stored in its local binding cache (maintained within Caller’s address space) (Figure 2a). Binding caches allow objects to take advantage of the temporal locality often observed across method invocations. An object’s binding cache is
automatically maintained by the binding process. If Caller has a cached binding for Callee, the binding process is finished (we discuss the problem of detecting stale bindings and obtaining current OAs in Section 4.4). If a cache miss occurs, Caller contacts its binding agent—a core object that implements a shared binding cache for its clients (Figure 2b). If the binding agent does not have the requested binding, it can consult an alternate external source. It can forward the request to another binding agent (e.g., binding agents can be organized hierarchically to form a multi-level cache structure). As a final option, it can consult Callee’s class, CalleeClass, since class objects are responsible for knowing the current binding of all of their instances (Figure 2d). Determining an object’s class is called the class-of mechanism (Section 4.2).

Once the binding agent obtains CalleeClass’s LOID, it can request Callee’s binding from CalleeClass. However, the binding agent must first execute the binding mechanism to determine CalleeClass’s OA. This request might in turn require executing the class-of mechanism to find CalleeClass’s class, CalleeMetaclass. There can be an arbitrarily long chain of metaclasses, in which case the binding process is repeated recursively. Since the class-of hierarchy is rooted at LegionClass, the mechanism is guaranteed to terminate.
4.2 Class-Of Mechanism

If the binding mechanism needs to consult an object’s class, it must determine that class’s LOID. The class-of mechanism maps an object’s LOID to its class’s LOID. As with bindings, objects and binding agents maintain class-of caches containing the results of recent class-of operations. In the event of a class-of cache miss the class-of mechanism is performed through a binding agent. As with bindings, binding agents provide a shared caching mechanism for class-of results. If the class-of result is not cached locally or in the binding agent, the class-of caller (in our running example, Callee’s binding agent) must consult the comprehensive and logically-global Legion class map. The class map is maintained by LegionClass, which is located at a well-known OA. In practice, LegionClass is distributed over multiple processes, providing a distributed, replicated class map. It is worth noting that the class map is a “write once, read many” database; the Legion object model does not allow the class of an object to change. Therefore, replicating the class map does not incur the overhead of maintaining cache coherence.

4.3 Stale Bindings

Bindings can become stale as the objects to which they refer deactivate or migrate. For example, if Caller has a binding for Callee, Caller may find that this binding is stale (e.g., by repeated failed attempts to communicate), in which case Caller invokes the re-binding mechanism. The re-binding mechanism mirrors the regular binding mechanism, but it uses the stale OA to ensure that the same binding is not returned. Caller begins by checking its binding cache for Callee’s LOID: if the only binding in the cache is the one containing the stale OA, that binding is removed from the cache, and the binding agent is consulted. The stale OA is passed as a parameter to the binding agent, indicating that Caller was unable to use that binding. As in the binding process, CalleeClass may be consulted as the final authority for locating its instances.
4.4 Object Activation

So far we have based our discussion of the binding process on the fundamental assumption that classes could always return a valid OA for their instances. However, inert objects are located at an OPA, not an OA. For example, if Callee were inert when Caller invoked `func()`, all bindings cached in Caller and in any binding agents in the system would be stale. The binding process would result in a call to CalleeClass to obtain a new binding for Callee. When asked for Callee’s binding, CalleeClass employs the object activation mechanism to activate Callee (Figure 3) before returning Callee’s new OA.

The object activation mechanism comprises several steps. Before activating Callee, CalleeClass selects a host on which to execute Callee. All class objects have complete freedom in selecting hosts for its instances, each class potentially using a different selection strategy. A conservative class may place all of its instances on its own host, while another class may use an external scheduling agent that employs a more elaborate and flexible placement policy. A scheduling agent may implement any specialized placement algorithm appropriate for the class, such as one appropriate for a 2D finite difference algorithm used in an ocean model, or one designed to meet a particular organization’s security requirements. Whatever the policy,
scheduling agents typically interact with other information providers (objects that dynamically gather information about the state of hosts and the rest of the system). For more details see Karpovich [22] about the Legion scheduling model and Berman [4] about application specific scheduling agents.

Once a target host is selected, the class must ensure that the instance can access its OPR from that host. A Legion object requires direct access to its OPR, which resides on a physical storage device managed by a vault object. The target host selected for an instance must have access to the storage device containing the instance's OPR. If the storage devices managed by a vault are accessible from a given host, we call this host and vault *compatible*. During activation, if a class selects a host that is not compatible with the current vault containing the instance's OPR, the class must migrate the OPR to a compatible vault. Once a compatible host/vault pair is chosen, the class invokes the host object's `startObject()` method. The `startObject()` call may fail for many reasons, e.g. because the host object refuses the activation request for policy, performance or security reasons, or because the underlying machine is temporarily down. If the `startObject()` invocation fails, the class object must make another placement selection.

The `startObject()` method requires the instance's LOID and OPA, as well as the appropriate *implementation object* LOID for the instance as parameters (implementation objects are Legion objects that store executable code for other objects—see Section 5.4). To service the activation request the host object must first obtain a local copy of the executable stored in the specified implementation object. A simple host object might download the executable for every `startObject()` invocation, but doing so wastes both communication and storage resources. Thus, groups of host objects typically share an *implementation cache*, a Legion object that
downloads and locally caches object executables. To use an implementation cache, the host object sends the desired implementation object’s LOID to the cache object. The cache object returns the name of a local file containing the executable. In servicing such requests, the cache may download the executable, if necessary, or return a cached local copy.

Once the implementation is locally available, the host object executes it according to the implementation type and the host characteristics. If the implementation is native code and the host is a Unix variety of host, then the host executes a `fork()/exec();` if the implementation is Java bytecode, the host executes it within a Java Virtual Machine; if the host represents a workstation farm managed by a queueing system such as Condor [25] or LoadLeveler [21], the host starts the object through the batch system’s interface. During activation, the host passes the object its LOID and OPA. The object then sends its OA to its class object, which marks the instance as active, records its OA, and can once again return fresh bindings for the instance.

The binding and activation processes can be time consuming. However, in practice, aggressive caching of bindings and executables avoids much of the cost and the benefits of this design are many: flexibility, transparent binary migration, one-step system-wide replacement for object executables, object-local policy autonomy, licensing and proxies, user-definable scheduling policies, user-definable persistent storage, etc. The following sections describe how users can realize these and other features by using and customizing core object implementations.

5. Core Object Types

5.1 Classes and Metaclasses

Every Legion object is defined and managed by its class object. Class objects are *managers* and *policy makers* and have system-like responsibility for creating new instances, activating and deactivating them, and providing bindings for clients. Legion encourages users to define and
build their own class objects. These two features—class object management of their instances and the ability for applications programmers to construct new classes—provide flexibility in determining how an application behaves and further support the Legion philosophy of enabling flexibility in the kind and level of functionality.

Legion classes must implement all of the class-mandatory interface (Figure 4). This interface includes `createInstance()`, which creates a new instance of the class and returns the new LOID; `createMultipleInstances()`, which creates several instances of the class at once; and `activateInstance()`, which migrates an existing instance to a new location or re-starts an instance that has become inert. Each of these three functions actually has several versions, allowing the caller to tailor the creation and placement processes. For example, the caller can indicate the host object on which the instance(s) should be placed, or specify the characteristics of acceptable hosts (processor speeds, architectures, etc.). The `addImplementation()` and `removeImplementation()` functions configure which implementations the class object will use for its instances (these functions are typically used by Legion-targeted compilers). The `getBinding()` functions support binding as described in Section 4.1. There are several other functions (not shown in Figure 4) that allow clients to

```c
class ClassObject {
    LOID createInstance(<placement info>);
    LOIDArray createMultipleInstances(int n, <placement info>);
    int activateInstance(LOID instance, <placement info>);
    int deleteInstance(LOID instance);
    int deactivateInstance(LOID instance);
    int addImplementation(LOID implementation_object);
    int removeImplementation(LOID implementation_object);
    Binding getBinding(LOID instance);
    Binding getBinding(Binding stale_binding);
};
```

FIGURE 4. A subset of the Legion class-mandatory interface

3. Note that an object can disallow any function invocation request, typically based on the identity of the caller. This is especially relevant to the system-level functions implemented in core objects[35].

Page 17
retrieve information about the location and characteristics of the class’s instances, such as the instances’ interface, their current host, their current state (active or inert), etc.

Class objects are in an ideal position to exploit the special characteristics of their instances. This does not mean that all programmers must build a class object for each Legion class that they build. On the contrary, we expect that a vast majority of programmers will be served adequately by existing metaclasses. Metaclass objects are class objects whose instances are themselves class objects. Just as a normal class object maintains implementation objects for its instances, so too does a metaclass object. A metaclass object’s implementation objects are built to export the class-mandatory interface and to exhibit a particular class functionality behind that interface. To use one, a programmer simply calls createInstance() on the appropriate metaclass object, and configures the resulting class object with implementation objects for the application in question. The new class object then supports the creation, migration, activation, and location of these application objects in the manner defined by its metaclass object.

For example, consider an application that requires a user to have a valid software license in order to create a new object, e.g., a video on demand application in which a new video server object is created for each request. To support this application, the developer could create a new metaclass object for its video server classes, the implementation of which would add a licence check to the object creation method.

5.2 Host Objects

Legion host objects abstract processing resources in Legion. They may represent a single processor, a multiprocessor, a Sparc, a Cray T90, or even an aggregate of multiple hosts. A host object is a machine’s representative to Legion: it is responsible for executing objects on the machine, protecting objects from each other, reaping objects, and reporting object exceptions. A
host object is also ultimately responsible for deciding which objects can run on the machine it represents. Thus, host objects are important points of security policy encapsulation.

Aside from implementing the host-mandatory interface (Figure 5), host object implementations can be built to adapt to different execution environments and suit different site policies and underlying resource management interfaces. For example, the host object implementation for an interactive workstation uses different process creation mechanisms than implementations for parallel computers managed by batch queuing systems.

```java
class Host {
    ObjectAddress startObject(LOID object, LOID impl, OPRAddress opa);
    void deactivateObject(LOID object);
    ObjectAddress getObjectAddress(LOID object);
};
```

FIGURE 5. Basic Legion host object interface.

Whereas host objects a uniform interface to different resource environments, they also (and more importantly) provide a means for resource providers to enforce security and resource management policies within a Legion system. For example, a host object implementation can be customized to allow only a restricted set of users access to a resource. Alternatively, host objects can restrict access based on code characteristics (e.g. accepting only object implementations that contain proof-carrying code [29] demonstrating certain security properties, or rejecting implementations containing certain "restricted" system calls).

We now consider our current default host object, and two possible alternative implementations. Our default design is very simple—it implements a non-restrictive access policy and uses the Unix process management interface (i.e. fork(), exec(), kill()) for starting and stopping objects. Although simple to implement, this design has many flaws. It places a high cost on object activation, executing one process per object and creating new processes on demand for each activated object. This implementation is severely limited in terms
of security—it executes all objects under a single Unix user id, allowing objects from different Legion users to interfere with or examine one another's state. Below we briefly present some ideas to address these limitations transparently using alternative host object implementations.

An implementation to address the performance problems might use threads instead of processes. This design improves the performance of object activation, and also reduces the cost of method invocation between objects on the same host by allowing shared address space communication. To support this style of host object, alternate forms of object implementations need to be made available, particularly object implementations in the form of dynamically loadable object files (as opposed to normal executable files). This allows the host to map the needed code for objects into its address space prior to object activation (i.e., thread creation). This need for alternate forms of object implementations fits nicely into our established model for managing multiple object implementations per class as needed to support heterogeneity.

The above host object design appeals to users with high-performance requirements, but it shares and exacerbates our basic host object's security limitations. An alternate host object implementation that supports better security properties can be based on the use of multiple Unix user-ids to run different users' objects. This design (which has been implemented and is provided as a standard part of the Legion software distribution) provides better interobject isolation, and the possibility of better attribution of resource usage to users.

We have implemented a spectrum of host object choices that trade-off risk, system security, performance, and application security. An important aspect of Legion site autonomy is the freedom of each site to select the existing host object implementation that best suits their needs, extend one of the existing implementations to suit local requirements, or to implement a new host object starting from the abstract interface. In selecting and configuring host objects, a site can control the use of their resources by Legion objects.
5.3 Vault Objects

Vault objects are responsible for managing other Legion objects' persistent representations (OPRs). Much in the same way that hosts manage active objects' direct access to processors, vaults manage inert objects on persistent storage. A vault has direct access to a storage device (or devices) on which the OPRs it manages are stored. A vault's managed storage may include a portion of a Unix file system, a set of databases, or a hierarchical storage management system.

The vault supports the creation of OPRs for new objects, controls access to existing OPRs, and supports the migration of OPRs from one storage device to another.

As previously noted, class objects manage the assignment of vaults to instances: when an object is created, its vault is chosen by the object's class. The selected vault creates a new, empty, OPR for the object, and supplies the object with its OPA. Similarly, when an object migrates, its class selects a new target vault for its OPR. These vault activities are supported by the basic Legion vault abstract interface (Figure 6). The \texttt{createOPR()} method constructs a new empty OPR, associates this OPR with the given LOID, and returns the address of the new OPR to be used by the newly created object. The \texttt{getOPRAddress()} method is used to

```c
class Vault {
    OPRAddress createOPR(LOID object);
    OPRAddress getOPRAddress(LOID object);
    LinearOPR getOPR(LOID object);
    void giveOPR(LOID object, LinearOPR OPR);
    void deleteOPR(LOID object);
    void markActive(LOID object);
    void markInactive(LOID object);
};
```

FIGURE 6. The Legion vault object interface.

determine the location of the OPR associated with any of its managed objects. The \texttt{giveOPR()} and \texttt{getOPR()} methods transfer a linearized (i.e., transmissible) OPR to and from vaults, respectively, facilitating object migration. The \texttt{deleteOPR()} method is used to terminate a
vault's management of an OPR. Finally, \texttt{markActive()} and \texttt{markInactive()} notify the vault when an object is active or inactive. This knowledge allows the vault to store the OPRs of inactive objects in compressed or encrypted forms.

5.4 Implementation Objects

Implementation objects encapsulate Legion object executables. The executable itself is treated much like a Unix file (i.e. as an array of bytes) so the implementation object interface naturally is similar to a Unix file interface: \texttt{read()}, \texttt{write()}, and \texttt{sizeof()} (Figure 7). Implementation objects are also write-once, read-many objects—no updates are permitted after the executable is initially stored. Therefore, there is no danger of replicated executables becoming inconsistent.

```c
class ImplementationObject {
    ByteArray read(size_t startByte, size_t szToRead);
    size_t write(size_t startByte, ByteArray data);
    size_t sizeof();
};
```

\textbf{FIGURE 7.} The Legion implementation object interface

Implementation objects typically contain executable code for a single platform, but may in general contain any information necessary to instantiate an object on a particular host. For example, implementations might contain Java byte code, Perl scripts, or high-level source code that requires compilation by a host. Like all other Legion objects, implementation objects describe themselves by maintaining a set of attributes (Section 3.2). In their attributes implementation objects specify their execution requirements and characteristics which may then be exploited during the scheduling process. For example, an implementation object may record the type of executable it contains, its minimum target machine requirements, performance characteristics of the code, etc.

Class objects maintain a complete list of (possibly very different) acceptable implementation objects appropriate for their instances. When the class (or its scheduling agent)
selects a host and implementation for object activation, it selects them based on the attributes of the host, the instance to be activated, and the implementation object.

Implementation objects allow classes a large degree of flexibility in customizing the behavior of individual instances. For example, a class might maintain implementations with different time/space trade-offs and select between them depending on the currently available resources. To provide users with the ability to select their cost/performance trade-offs, a class might maintain both a slower, low-cost implementation and faster, higher-cost implementation. This is similar to abstract and concrete types in Emerald [3].

5.5 Implementation Caches

Implementation caches avoid storage and communication costs by storing implementations for later reuse. If multiple host objects share access to some common storage device they may share a single cache to further reduce copying and storage costs. The interface to the implementation cache object is depicted in Figure 8—a single method is provided to return the path of a local file containing a given implementation object’s data. Host objects, rather than downloading implementations themselves, invoke `getImplementation()` on their local implementation cache object. The cache object either finds it already has a cached copy of the implementation or it downloads and caches a new copy. In either case, the cache object returns the executable’s path to the host. In terms of performance, using a cached binary results in object activation being only slightly more expensive than running a program from a local file system.

Our implementation model makes the invalidation of cached binaries a trivial problem.
Since class objects specify the LOID of the implementation to use on each activation request, a class need only change its list of binaries to replace the old implementation LOID with the new one. The new version will be specified with future activation requests, and the old implementation will simply no longer be used and will time-out and be discarded from caches.

5.6 Binding Agents

Section 4 introduced the binding and class-of mechanisms and the role of binding agents in helping clients map LOIDs to OAs and objects to their classes. Figure 9 shows the interface for binding agents. The `getBinding(LOID)` function returns a binding for a specified LOID, and `getClassBinding(LOID)` returns a binding for the class of a given LOID; both are intended to be invoked directly by a client object that is in search of a binding. The `getBinding(Binding)` and `getClassBinding(Binding)` methods support the rebinding mechanism, allowing a client to pass a stale binding and request a new binding. The `addBinding(Binding)` and `removeBinding(LOID)` functions allow a binding agent to act as a database of bindings under the control of external objects. A class can use `removeBinding(LOID)` to remove an instance's binding when that instance becomes inert or gets deleted, and can call `addBinding(Binding)` upon creation, activation, or migration of an instance.

```java
class BindingAgent {
    Binding getBinding(LOID object);
    Binding getClassBinding(LOID object);
    int addBinding(Binding new_binding);
    int removeBinding(LOID object);
};
```

Figure 9. The Legion binding agent interface

Binding agents are not, strictly speaking, necessary for the correct execution of the binding process; clients can directly contact class objects and LegionClass's class map to obtain
bindings for objects and classes. However, in order to make the binding mechanism scalable to a very large number of objects, binding agents are necessary to distribute the binding load and avoid hot-spots. To improve scalability, binding agents can be configured to cooperate with one another to serve their clients. For instance, they can be organized hierarchically, like DNS name servers, or can emulate a software combining tree [36], thereby sharing the responsibility for providing bindings away from classes and LegionClass.

5.7 Context Objects and Context Spaces

As described in Section 3.1, Legion objects are identified by LOIDs. A LOID contains a set of fields including those that identify the class of the object, a class-unique instance number, and a public key. Given this set of fields, LOIDs can grow quite large. Whereas LOIDs are typically transmitted and manipulated in binary form, a “dotted-hex” textual representation for use by human users is also supported (Figure 10). As the figure clearly demonstrates, LOIDs are by no means convenient for human users. To address the basic need for a convenient object naming mechanism and to provide a tool for organizing information we define the interface to a user-level naming service called context spaces.

```
<table>
<thead>
<tr>
<th>LOID Type</th>
<th>Legion Domain</th>
<th>Class ID</th>
<th>Instance Number</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.01.66000000.21000000.000001fc0cf5465691d88fbd0417ed590ce2a7ff4db9fd92cb95471c3eaf53e1b9b805226292bf88a6d7d50fbb676acef0fe53433410ab064713c0fcaeff3161cd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

FIGURE 10. Example of a LOID

Context spaces are directed graphs of context objects that name and organize information. A context object provides an interface for managing a list of mappings between user-defined string names and LOIDs (Figure 11). Operations are provided to insert, remove and find user name-to-LOID mappings contained within the context object, including a method (multilookup()) to return a list of mappings that match a specified regular expression.
class Context {
    int insert(String name, LOID loid);
    int remove(String name);
    LOID lookup(String name);
    List<String,LOID> multilookup(String regexp);
};

FIGURE 11. The Legion context object interface

In isolation, a context object may be used to provide a simple, convenient user-level naming service for a user’s objects. However, the names inserted into a context can map to other context objects’ LOIDs, providing a natural mechanism for constructing a directory service. Connected graphs of context objects are a basic mechanism for organizing information in Legion, and are referred to as context spaces. Every Legion object contains the LOID of a current working context and a root context, and library routines are provided for traversing context space to map context paths to LOIDs.

On the surface, context space appears to provide a basic directory service. However, much of the importance of context space in Legion is derived from the fact that any kind of object can be named in context space—contexts are not limited to listing names of other contexts and files. Therefore, context space provides a convenient way of organizing information about any of the objects that are available in a Legion system.

6. Related Work

Legion is one of a number of projects developing software to support metacomputing. This section discusses some of the current major metacomputing projects such as Globus [12] and Globe [33]. However, it is worth noting that these projects, Legion, and other metacomputing projects such as MOL [32], Ice-T [14], and Harness [8], are all outgrowths of the significant existing work in first-generation network parallel computing systems, such as PVM

4. Note that there is no notion of a global “root” context for the system. The root is a user-definable starting point for resolving fully qualified context paths.
[13] and MPI [19], and in modern transparent distributed computing systems, such as the Berkeley NOW project [1] and DCE [26].

6.1 Globus

The Globus project [12], at Argonne National Laboratory and the University of Southern California, and Legion share a common base of target environments, technical objectives, and target end users, as well as a number of similar design features. For example, similar to Legion's use of context space, Globus organizes information about resources and other entities of interest within the system in a Metacomputing Directory Service (MDS) [11]. Both systems abstract access to processing resources: Legion via the host object interface; Globus through the Globus Resource Allocation Manager (GRAM) interface [7]. Both systems also support a range of programming interfaces, including popular packages such as MPI.

Despite these similarities, the systems differ significantly in their basic architectural techniques and design principles. Whereas Legion builds higher-level system functionality on top of a single unified object model, the Globus implementation is based on the composition of working components into a composite metacomputing toolkit. For example, MDS is based on an existing directory service implementation, the Lightweight Directory Access Protocol (LDAP).

The Globus approach of adding value to existing high-performance computing services, enabling them to interoperate and work well in a wide-area distributed environment has a number of advantages. For example, this approach takes great advantage of code reuse, and builds on user knowledge of familiar tools and work environments. However, this sum-of-services approach has a number of drawbacks: as the amount of services grows in such a system, the lack of a common programming interface and model becomes a significant burden on end users. By providing a common object programming model for all services, Legion enhances the
ability of users and tool builders to employ the many services that are needed to effectively use a metacomputing environment: schedulers, I/O services, application components, and so on. Furthermore, by defining a common object model for all applications and services, Legion allows a more direct combination of services. For example, traditionally system-level agents such as schedulers can be migrated in Legion, just as normal application processes are—both are normal Legion objects exporting the standard object-mandatory interface. We believe the long-term advantages of basing a metacomputing system on a cohesive, comprehensive and extensible design outweigh the short-term advantages of reusing existing parallel and distributed computing services.

6.2 Globe

The Globe [33] project, which is being developed at Vrije Universiteit, also shares many common goals and attributes with Legion. Both are middleware metasystems that run on top of existing host operating systems and networks, both support implementation flexibility, both have a single uniform object model and architecture, both use class objects to abstract implementation details, and so on.

However, Globe's object model is different; a Globe object is passive and is assumed to be physically distributed over potentially many resources in the system. A Legion object is active, and although we don't preclude the possibility of it being physically distributed over multiple physical resources, we expect that it will usually reside within a single address space. These conflicting views of objects lead to different mechanisms for interobject communication; Globe loads part of the object (called a local object) into the address space of the caller whereas Legion sends a message of a specified format from the caller to the callee.

Another important difference is Legion's core object types. Our core objects are designed
to have interfaces that provide useful abstractions that enable a wide variety of implementations. As of the writing of this paper, we are not aware of similar efforts in Globe. We believe that the design and development of the core object types define the architecture of a system, and ultimately determine its utility and success.

6.3 CORBA

The Common Object Request Broker Architecture (CORBA) standard developed by the Object Management Group (OMG) [31] shares a number of elements with the Legion architecture, although it is not intended for metacomputing. As in Legion, CORBA systems support the notion of describing the interfaces to active, distributed objects using an IDL, and then linking the IDL to implementation code that might be written in any of a number of supported languages. Compiled object implementations rely on the services of an Object Request Broker (ORB), analogous to the Legion run-time system, for performing remote method invocations.

Despite these similarities, the different goals of the two systems result in different features. Whereas Legion is intended for executing high-performance, typically parallel applications, CORBA is more commonly used for business applications, such as providing remote database access from clients. This difference in intended usage manifests itself at all levels in the two systems—from basic object model up to the high-level services provided. For example, where Legion provides macro-dataflow method execution model suitable for parallel programs, CORBA provides a simpler remote-procedure call based method execution model suited to client-server style applications.

7. Summary

Metasystems are on the horizon. They are enabled by the tremendous increase in the available
network bandwidth. Constructing metasystem software to meet the needs of a diverse user and resource owner community will not be easy; metasystem must software be extensible to meet unanticipated needs and it must provide complete site autonomy.

Legion meets these requirements by using replaceable system components that encapsulate both policy and mechanism, and by enabling classes and metaclasses with system-level functionality. The result is a system that a user can shape to meet a particular application’s needs, controlling how the system is implemented with respect to that application, while at the same time ensuring that the resulting application can interact with other Legion applications via a standard set of basic protocols. At the same time, resource owners can protect their resources and can ensure that they are used in an appropriate manner.

In June, 1996, after a year of design work, we began code development for Legion, and in December of 1997 we released Virginia-Legion 1.0, a complete implementation including the class and metaclass structure, host objects, vault objects, binding agents, authentication, encryption, access control, context spaces, support for several languages, and many different tools and utilities. Legion is available on a variety of platforms, ranging from workstations (e.g., Sun, SGI, IBM, DEC) and PCs (Linux over Alpha or Intel) to supercomputers such as the IBM SP2, Cray T90, and SGI Origin 2000. More information about Legion, including the freely available implementation is available at http://legion.virginia.edu.

References


Heterogeneous Process State Capture and Recovery through Process Introspection

Adam J. Ferrari
Steve J. Chapin
Andrew S. Grimshaw

Department of Computer Science
University of Virginia, Charlottesville, VA 22903, USA
{ferrari|chapin|grimshaw}@cs.virginia.edu

Abstract
The ability to capture the state of a process and later recover that state in the form of an equivalent running process is the basis for a number of important features in parallel and distributed systems. Adaptive load sharing and fault tolerance are well-known examples. Traditional state capture mechanisms have employed an external agent (such as the operating system kernel) to examine and capture process state. However, the increasing prevalence of heterogeneous cluster and "metacomputing" systems as high-performance computing platforms has prompted investigation of process-internal state capture mechanisms. Perhaps the greatest advantage of the process-internal approach is the ability to support cross-platform state capture and recovery, an important feature in heterogeneous environments. Among the perceived disadvantages of existing process-internal mechanisms are poor performance in multiple respects, and difficulty of use in terms of programmer effort. In this paper we describe a new process-internal state capture and recovery mechanism: Process Introspection. Experiences with this system indicate that the perceived disadvantages associated with process-internal mechanisms can be largely overcome, indicating that this approach to state capture is the appropriate one for cluster and metacomputing environments.

1. Introduction

The ability to capture the state of a process and later recover that state in the form of an equivalent running process is the basis for a number of important features in parallel and distributed systems. For example, process migration policies supporting adaptive load sharing and/or fault tolerance rely on a state capture facility. Process state capture and recovery is the basis of a large class of backward error recovery schemes documented in the fault tolerance literature[6]. Optimistic systems such as Time Warp[12] rely on the ability to "roll back" a local computation to provide semantic guarantees (such as the causal ordering of message delivery), and thus also require a process state capture mechanism. Distributed object systems can use process state capture and recovery to implement long-lived persistent objects efficiently, as is done in the Legion system.
Traditional state capture mechanisms have employed an external agent (such as the OS kernel) to examine and capture process state [19]. However, the increasing prevalence of heterogeneous cluster and "metacomputing" systems [10] as high-performance computing platforms has prompted investigation of process-internal state capture mechanisms. Among the greatest advantages of process-internal approaches are portability and the ability to support cross-platform state capture and recovery. A process can interpret the semantics of its own data regions, and thus produce a state description that can be used to reconstruct an equivalent process on a different platform. This flexibility has been found to be a valuable feature in heterogeneous cluster and metacomputing environments such as Legion, Cumulvs [14], and Dome [2].

Despite its increasing adoption in such environments, process-internal state capture and recovery has thus far been considered lacking in at least two respects: performance and usability. Performance has been considered problematic for process-internal mechanisms by a number of measures. For example, process-internal approaches require a state-capture request to be sent to the process. The delay inherent in servicing this request appears as decreased responsiveness of the mechanism in contrast to external approaches can initiate state capture at any time. Furthermore, the added overhead of maintaining meta-information (such as type) for the process's data regions can add run-time overhead, lowering raw computational performance. Lack of usability of process-internal mechanisms is due mainly to added programmer effort and/or limitations on the types of services the process can use. For example, in some cases, such as Legion, the code needed to describe and recover the process's state must be provided by the user.

In this paper we describe a new process-internal state capture and recovery mechanism: Process Introspection. This system is based on a combination of library and compiler support to maximize the ease of use of process-internal state capture and recovery. For platform-independent modules, the compiler completely automates state capture and recovery. For modules where automatic transformation is not possible, a flexible library providing the needed primitive operations for cross-platform state capture and recovery makes adding state capture functionality straight-forward. In Section 2 we describe the design and interface of our system, and in Section 3 we present key features of the implementation. We argue that the design of the Process Introspection system overcomes the usability flaws in existing process-internal state capture mechanisms. In Section 4 we describe the results of performance measurements of our system.
These results indicate that a cross-platform process-internal state capture mechanism can offer good performance. Our results lead us to conclude that the process-internal approach to state capture is the appropriate one for cluster and metacomputing environments. In Section 5 we describe related work, and in Section 6 we discuss our conclusions.

2. Design

The design of a process-internal state capture mechanism is naturally based upon the modification of user programs to render them both self-describing and self-recovering. Beyond the basic goal of providing a platform-independent state capture and recovery service, important goals for such a mechanism are to provide good performance and ease of use. In this section, we describe the key features of our design as a basis for discussing how it meets these goals.

2.1 Model

In our model, a running process is defined to be in one of three states: normal execution, state-capture, or state-recovery. The state of the process is changed by the program itself, either in response to requests from outside sources or as the result of an internal trigger such as periodic checkpoint scheduling. We require that the program periodically execute poll points: points in the code at which the process determines if it is in the state-capture mode, in which case a state description should be produced if one has been requested (analogous to Bus Stops in Heterogeneous Emerald [23]). Certain parts of the process state are easily captured—for example, any global variable or heap allocated data structures are globally addressable and are thus easy to manipulate. The key difficulty in creating the state description is the capture of the subroutine invocation stack state.

In the Process Introspection approach, the process utilizes the native "subroutine return" mechanism to capture stack state. When a poll point is encountered during state-capture mode, the current active subroutine captures its own state, including its local variables and the logical location of the poll point at which the current call frame was saved (e.g., this can simply be an integer that uniquely identifies the poll point within the subroutine), and returns to the caller. After the return, the caller saves its own state in the same way, and this frame-by-frame stack capture repeats until the base subroutine has been reached, at
which point the stack state capture is complete. For this stack-saving mechanism to work, the program
must execute a poll point after returning from each subroutine call. At this point, the program might be in
normal execution mode, in which case it proceeds with normal processing, or it might be in state-capture
mode, in which case the stack save process continues. We name these required poll points following sub-
routine returns mandatory poll points. In fact, more frequent checks for state capture initiation may be
desirable, in which case additional optional poll points can be placed anywhere in a program.

A side-effect of capturing state in this manner (but not of simply polling) is the destruction of all data
on the call stack, implying that the program must perform some of the work of a restart to recover the stack
if it is to continue execution after capturing state. We note an important optimization to the above mecha-
nism: in the process of capturing the state of each frame, the frame state should also be saved in memory.
This permits the implementation of a quick stack recovery after state capture. Because the state capture
mechanism is non-destructive to other state (i.e. global variables and dynamically-allocated memory
blocks), this optimization permits the process to proceed without unreasonable delay after state capture.

A further optimization to the described code modification scheme is also possible. Although we ini-
tially stated that mandatory poll points must be placed immediately following every subroutine call state-
ment, we can in practice loosen this restriction. Given knowledge that all possible call chains resulting
from a subroutine call would contain no poll points, the mandatory poll point following the call site can
safely be omitted. For example, consider a call to a simple function that calls no other functions and con-
tains no poll points. Upon return from this function, we know that a capture of the stack could not yet have
been initiated. Even if state capture had been requested while the function was executing, we can safely
continue normal execution after the call returns before beginning to service the request.

To effect restarts, the process employs the native "subroutine call" mechanism. On a restart, the pro-
gram is started and is immediately placed in state-recovery mode. The base subroutine of the program
always executes a prologue that checks for state-recovery mode, then conditionally recovers the data for its
local stack frame and jumps to the location in the subroutine for the call to the next stack frame, i.e., the
poll point at which the state of the current stack frame was captured. Each stack frame in turn is recovered
by its respective subroutine, which must implement its own stack-recovery prologue. Before jumping just
past the poll point that initiated the state capture, the final frame resets the process state to \textit{normal-execution} mode to complete the state recovery and resume normal execution. Of course, at some point during the recovery process, the global variables and heap allocated data must also be restored.

With these additions, the program can restore an intermediate state as produced by its own state capture mechanism. In particular, since the state capture and recovery mechanisms are specified at a platform-independent level of representation, different implementations of the program (i.e. versions compiled for different architectures) can read and write one another's captured state, assuming the associated data is stored in a universally recognizable format, masking issues such as data representation (cf. Sun XDR [21]).

2.2 System Usage

This model of process-internal state capture and recovery appears at first glance to require significant programmer effort. In practice, many of the described code transformations can be automated. The Process Introspection system does this through the use of a source code compiler and run-time support library. For computational modules that are specified in a platform-independent form (i.e., that are written in a high-level language, are type-safe, and do not rely on the underlying features of a particular hardware platform for correctness), the described code transformations can be completely automated.

We expect this completely automatic usage of the system to be the typical mode employed by application programmers such as domain scientists. However, some program modules are not amenable to automatic transformation. For example, modules that deal with state that is external to the process (e.g., file system interfaces, communication interfaces) can not be automatically transformed by our system to incorporate state capture and recovery. In some cases, it seems that this is in fact an inherent limitation of our approach. Consider a message passing interface module. A compiler attempting to incorporate state capture and recovery functionality into such a module would have no way to know how to encode the capture of state such as messages in transit, nor would it be able to determine the state capture coordination semantics required by the application. In such cases, our model requires that the module be augmented by hand to incorporate state capture and recovery functionality—this typically involves the creation of a state-capture-enabled wrapper module for the interface in question. We envision the creation of state-capture-enabled library modules as an infrequent activity undertaken by cluster and metacomputing software system
designers. These wrapper libraries can then be reused by application programmers whose own modules will be transformed automatically. To support the interoperation of state-capture-enabled library modules and automatically transformed application modules, and to ease the hand-coding of state capture mechanisms for library modules, our system supports a library interface. This library provides basic services such as cross-platform data-format transformation routines, routines for constructing descriptive meta-information about the data regions of the process (such as a data type description table), and an event model for allowing separately developed state-capture-enabled modules to interoperate.

3. Implementation

3.1 Overview

We have constructed a prototype implementation of the Process Introspection system consisting of a library module as described in Section 2.2, the Process Introspection Library (PIL), and a source code translator called APRIL (Automatic application of the Process Introspection Library) which can automatically apply the Process Introspection transformations to platform-independent modules written in ANSI C. The implementation has been tested on a variety of workstation and PC platforms, including Sun workstations running Solaris or SunOS 4.x, SGI workstations running IRIX 5.x and 6.x, IBM RS/6000 workstations running AIX, DEC Alpha workstations running Digital Unix or Linux, and PC compatibles running Linux or Microsoft Windows 95/NT.

3.2 The Process Introspection Library

The Process Introspection Library (PIL) is the most basic component of the system. In the case of hand-coded modules, the PIL provides the API for implementing a module's state capture and recovery capability. In the case of compiler-transformed modules, the PIL provides needed run-time support. The primary job of the Process Introspection Library is to provide an easy-to-use mechanism for describing, saving, and restoring data regions in as automatic a fashion as possible. In addition, the library provides an event-based mechanism for coordinating the activities of modules during state capture and recovery. The key elements of the PIL are:

The Type Table—To capture or restore a memory block, the PIL must have a description of the basic data
types stored in that memory block. The PIL provides an interface to a table which maps type identifiers to
logical type descriptions. These type identifiers can then be used to tag data regions, indicating the types of
the data found in the region. The type table is not unlike a type description table that might be found in a
compiler, except that it is available and dynamically configurable at run-time. The interface provides pre-
defined type identifiers for the basic types supported by ANSI C, and provides an interface for composing
vector and record descriptions based on existing types.

Data Format Conversion Module — The PIL provides an interface for reading and writing typed data
from and to a state description in an architecture-independent format, respectively. This interface is respons-
able for masking differences in byte ordering and floating point representation. When storing captured
state, the library automatically includes a description of the data formats used. Later, during state recovery,
the data format can be converted to the restarting processor's representation, a protocol known as receiver-
makes-right [26]. Given this approach, the library must contain routines to translate the set of basic data
types from every available format to every other available format. This $O(n^2)$ requirement (where $n$ is the
number of different data formats) may initially appear unnecessarily costly; why not instead use an $O(n)$
solution such as XDR? In fact, the receiver-makes-right protocol makes sense only in light of the very
small number of data formats actually used by current computer systems and because the cost of format
conversions is avoided for the frequent case in which captured state is recovered on a computer with simi-
lar data formats to the one on which it was created.

Pointer Analysis Module — Memory addresses (i.e. pointers) contained within memory blocks are inher-
ently platform-dependent. Thus, they must be stored using a logical format in place of the physical address.
Similarly, at state recovery time, logical pointer descriptions must be translated to determine the actual
local memory address values that should be restored. Our mechanism for this assigns a unique identifica-
tion number to every memory block of interest in the program, and a logical pointer description comprises
a <memory block identification tag, offset> tuple. Based on this idea, the Pointer Analysis Module pro-
vides a convenient interface for generating logical descriptions of memory locations and for mapping these
logical descriptions into actual memory addresses. The pointer description implementation is based on
simple case analysis; a pointer can be one of exactly five types: a reference into a heap allocated memory
region, a reference into a global memory block, a reference into a local (stack) memory block, a pointer to some code entry point, or value which has special meaning in the program (such as NULL in C). The PIL associates id numbers with memory blocks of each class (except the last), providing the basis for logical pointer description.

The primary challenge posed by the use of logical pointer descriptions is the translation of these descriptions during state recovery—offsets into a memory region may need to be transformed due to differences in data size and alignment between the capture and recovery platforms. The Pointer Analysis Module uses a pointer translation algorithm that transforms offsets based on the type descriptions available for all memory regions (supported by the Type Table), and knowledge of the data formats and alignment issues of the source and target platforms. Full details of this algorithm are presented in [7].

**Global Variable Table**—The PIL provides an automated mechanism for saving and restoring the values of global variables at state capture and restart time, respectively. The mechanism requires that the memory addresses, type table indices, and sizes of all globally-addressable memory blocks be registered with the PIL in a Global Variable Table. Besides providing automatic capture and recovery of globals, the Global Variable Table is used to perform pointer analysis for addresses pointing within global memory blocks.

**Heap Allocation Module**—The PIL provides a mechanism for allocating memory blocks from the heap that will be automatically captured and restored. The Heap Allocation module exports heap wrapper routines that perform typed memory block allocation and deallocation. Similar to the case with globals, these wrapper routines maintain a table of the addresses, type table indices, and sizes of all active dynamically-allocated memory blocks.

**Code Location Table**—To fully resolve the meaning of all pointers, the PIL must maintain a table that maps logical code entry points to actual memory code locations. All subroutine entry points (and other addressable code locations) in a program that may be referred to by a pointer must be assigned a logical identification number via the Code Location Table interface.

**Active Local Variable Table**—Because pointers can refer to local variables, the addresses, type table indices, and sizes of some local variable memory blocks should be registered with the PIL. Note, only those locals whose addresses are ever examined (and thus whose addresses might consequently be found in some
memory block) need to be registered with the Active Local Variable Table. Local variables whose addresses are never examined should not be registered, preserving the possibility of register assignment.

**Event Module**—The Event Module provides the primary mechanism for modules to customize their state capture and recovery behavior. The Event Module allows a program module to register function callbacks that will be invoked by the system automatically at state capture and recovery time. To understand the importance of this module, consider the case of a file system interface module. Besides the normal activities of saving and restoring the data in memory blocks (as is done by every module, and which is typically automated using the PIL), the file module must perform extra actions. During state recovery, for example, it must use the local file interface to re-open the files that were in use when state was captured. It might also be responsible for maintaining the file version differences associated with different captured states. These extra activities can be coded in the form of event handlers which would be executed in response to state capture and restart events.

### 3.3 The APrIL Source Code Translator

The programming interface provided by the PIL automates some of the elements of the Process Introspection model, but is still relatively low-level. Although issues such as data representation are handled, using only the PIL the programmer would be left to manually perform code modifications such as pollpoint placement and prologue generation. Fortunately, a source code translator can automate this process for platform-independent programs. Using the Sage++ toolkit [3], we have implemented this functionality in the APrIL compiler. APrIL takes as input ANSI C code, and produces as output new ANSI C code transformed to utilizing the PIL as a run-time interface. The resulting C code can then be compiled using any ANSI C compiler. In this section, we examine the fundamental APrIL transformations.

### 3.3.1 Poll Points

APrIL inserts poll points throughout the code it transforms. At each poll point, code is inserted to check if the process is in state capture mode—this simply involves examining the value of the global variable PIL_State (we currently set the value of this variable using an interrupt handler that processes external state capture requests in the form of signals; in principle it could be set by other triggers such as periodic checkpoint scheduling, or a state capture request message). Immediately following the poll point,
code is inserted which will be executed when state capture is in progress. This code records the location in the current subroutine at which state is captured and jumps to a function epilogue that saves the actual parameters and local variables in the frame.

As described in the model, APrIL generates two kinds of poll points: optional and mandatory function call site poll points. Optional poll points can be inserted in the transformed code between any two statements in the universal representation. These poll points are designated by a single labeled code location (i.e. a C label statement). An example of an optional poll point is depicted in Figure 1.

In our model, mandatory poll points are inserted by APrIL after every function call statement in the code 1—these mandatory poll points are required to implement the stack save mechanism based on the native function return mechanism. When a function returns in APrIL transformed code, the return may be due to the normal completion of the function, or it may be a return being performed in the context of capturing the stack. Mandatory poll points must catch and implement this latter case. Mandatory poll points require two labeled code locations: one before the call site (to handle the case that state capture was initiated in a higher call frame), and one after the call site (in the event that state capture is initiated immediately following a normal function return). An example of a mandatory poll point is given in Figure 2. Note that if this poll point continues a stack capture that was initiated in a higher call frame, the code location is recorded as 2 to ensure that on

1. In fact, function calls in C occur not as specific statements, but instead within expressions. Expressions in turn can appear within other expressions, in more complex statements, etc. (e.g. a function call might be a parameter to another function call, which might be part of the conditional for an "if" statement). To perform the transformations as described in the model, APrIL utilizes a pre-processing step in which it extracts functions from complex expressions and statements, and reduces them to simple C expression statements containing a single function call.
recovery this frame will jump to label _PIL_PollPt_2, which will result in a call to the next function
needing to restore state. On the other hand, if this poll point initiates state capture, it records the code loca-
tion as 3 to ensure that on recovery the frame jumps beyond the function call—since this poll point initi-
ated the stack capture, the call to function must have completed normally (without capturing state), and
thus we must not re-call the completed function on recovery.

The placement of poll points in the code is a critical performance issue for APrIL. If poll points are
placed so that they occur frequently, the introduced overhead may be large. On the other hand, if poll
points are placed too infrequently, a state capture request sent to the process may suffer a long delay before
being serviced. Clearly, a balanced approach based on the user’s tolerance of introduced overhead and
state-capture-request wait time is required. If the user expects to perform state capture operations infre-
quently (e.g. once every minute), but demands little introduced overhead, then very sparse, conservative
poll-point placement is called for. Alternatively, if state-capture-request wait times must be very low (for
example, if state capture will be used for code migration to effect load sharing), more frequent, aggressive
placement is appropriate. However, the problem of statically examining code and determining the intro-
duced overhead and resulting state-capture-request wait time based on a given poll-point-placement strat-
egy is difficult, if not impossible. The current APrIL solution is to provide a set of heuristic placement
strategies with varying degrees of placement aggressiveness.

Our currently supported placement strategies are based on the observation that the primary mecha-
nisms for induction in procedural programming are iteration (i.e. loops) and subroutine invocation (e.g.
recursion). Although subroutine invocation already causes periodic polling (due to mandatory poll point
placement), it seemed likely that the addition of optional poll points into loops could provide more com-
plete periodic polling coverage over the lifetime of a program and thus lead to lower on-average state-cap-
ture-request wait times. Care must be taken, however, as the naive policy of placing a poll point inside each
loop body could lead to poor performance—careless placement of poll points can prevent the application
of many back-end optimizations, including those performed on loops. To allow further control over the
placement of poll points, the APrIL heuristic policies classify loops based on the number of statements in
the loop body and the nesting level of the loop. Compile time switches allow the user to restrict placement
to loops with given characteristics. Examples of possible policy selections include:

- Mandatory poll points only
- Mandatory poll points plus optional poll points placed as the last statement in each nesting loop (i.e. each loop that contains at least one other loop).
- Same as the previous, but poll points are added only to outermost loops with greater than $k$ statements.

Although many more policies are available, a better interface to the compiler would allow the programmer to specify the desired performance characteristics of the transformed code, which would guide the automatic selection of a policy. The degree to which this ideal can be approximated is the subject of future work. The performance characteristics of three currently available APRIL placement options are examined in Section 4.

3.4 Function Prologues

Function prologues are added to every function definition transformed by APRIL. If the addresses of any local variables or parameters (i.e., any objects stored on the stack) are examined in the function body, APRIL generates calls to the PIL to register those variables in the local variable table. APRIL then generates a check to determine if the process is in state recovery mode (recall that stack restoration in our model is implemented using the normal function call mechanism). APRIL generates code to be executed in case of a restart, which will restore the values of all local variables and actual parameters (using the PIL interface), determine the code location in the function at which the state for this frame was captured, and jump to the appropriate poll point label in the function. Figure 3 illustrates

```c
void example(double *A) {
    int i;
    double X[100];
}
```

(a) The original function heading.

```c
void example(double *A) {
    int i;
    double X[100];
PIL_RegisterStackPointer(X,PIL_Double,100);
    if(PIL_State&PIL_RecoverNow) {
        int PIL_code_loc;
        A = PIL_RestoreStackPointer();
        i = PIL_RestoreStackSize();
        PIL_RestoreStackDoubles(X,100);
        PIL_code_loc = PIL_FopCodeLocation();
        switch(PIL_code_loc) {
            case 1: PIL_DoneRestart();
                goto _PIL_PollPt_1;
            case 2: goto _PIL_PollPt_2;
            case 3: PIL_DoneRestart();
                goto _PIL_PollPt_3;
        }
    }
}
```

(b) The transformed function heading.

Figure 3: A function prologue transformation
an APrIL function prologue transformation. The function heading given in Figure 3(a) is transformed to include the prologue depicted in Figure 3(b).

This function has an array X whose address is used at some point in the function, and thus a call to register the address, size, and type this array is generated. The prologue checks the value of the PIL_State variable to determine if this function call was made in the process of restoring a call stack. If it was, the actual parameters and locals are restored using PIL routines. The point in the function at which the state was captured is then jumped to using a goto based on a code location marker read from the captured state. For some code locations (those corresponding to optional poll points and the second label associated with mandatory poll points), the generated code first calls PIL.DoneRestart() to complete the restart process and unset the PIL_State variable.

3.4.1 Function Epilogues

Poll points inserted by APrIL generate code to jump to a function epilogue during state capture to save all of the local variables and actual parameters for the function. APrIL generates an epilogue for each function it transforms that contains any poll points (if the function never polls for state capture requests, it will never need to save its state) placed beyond the last return statement; the epilogue is accessible only by goto, and is not executed during the normal progression of the program. The function epilogue for the example function from Figure 3 is depicted in Figure 4.

This design for saving the local state associated with a function call has the inherent implication that all local variables must be visible from the outermost scope of the function. To ensure this, APrIL moves the declaration of locals declared in inner scopes to the head of the function, renaming where appropriate to avoid name clashes.

3.4.2 Module Initialization

The three types of transformations discussed thus far are primarily aimed at implementing the state capture and recovery of function call stacks. APrIL also generates a routine to register any types defined by the translated module with the Type Table and register any globals defined by the translated module in the
Global Variable Tables. The generation of this function is a straightforward process based on any types and global variables found in the module.

3.4.3 Heap Allocation Transformations

One of the more difficult transformations that APrIL performs is the translation of all heap allocation requests into calls to the typed allocation routines provided as part of the PIL. Since heap allocation is not part of the C language syntax but is instead handled by library routines, APrIL is required to perform a heuristic to determine when heap allocation is taking place, and the type and size of the allocated memory. The currently implemented heuristic finds all calls to the standard C library heap allocation routines (e.g. malloc(), calloc(), realloc(), etc.), uses the parameters to the call determine the allocation size, and attempts to determine the allocation type first based on the type that the return value is cast to (if it is available), and (failing that), on the type of the variable to which the return value is assigned. While this heuristic is adequate in many cases, it can fail if the memory allocation method used does not match our expected patterns. Future work will include investigating better heuristics for finding and wrapping heap allocations.

4. Performance

To examine the performance characteristics of our prototype implementation, we applied the system to a set of numerical applications. This set of test programs included:

- **mm (Matrix Multiply)**—Computes the product of two dense, square matrices of 256x256 double precision floating point numbers using the standard O(n^3) algorithm.

- **gs (Gauss-Seidel)**—Solves the sparse linear system of \(10^4\) equations resulting from the discretization of a two dimensional Poisson equation with Dirichlet boundary conditions. The algorithm used is a standard Gauss-Seidel five point stencil iteration applied to a 80x80 grid of solution elements until the change in the two-norm of the solution is less than \(10^{-2}\).

- **qs (Quicksort)**—Applies a standard quicksort algorithm to an array of \(2^{21}\) integers.

- **ge (Gaussian Elimination)**—Performs Gaussian elimination with partial pivoting on a dense 512x512 matrix, followed by a back-substitution phase to obtain the solution vector.

- **cg (Conjugate-Gradient)**—Applies a basic conjugate-gradient iteration (no preconditioning) to the same linear system solved by the Gauss-Seidel test, using the same convergence criterion as that exam-
ple, with the solution discretized onto a 200x200 grid.

Tests were run on the heterogeneous set of test platforms listed in Table 1.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Processor/Configuration</th>
<th>RAM</th>
<th>OS</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>200 Mhz Pentium Pro, dual processor</td>
<td>64 MB</td>
<td>Linux 2.0</td>
<td>GNU gcc 2.7.2</td>
</tr>
<tr>
<td>alpha</td>
<td>500 Mhz DEC Alpha</td>
<td>128 MB</td>
<td>Linux 2.0</td>
<td>GNU gcc 2.7.2</td>
</tr>
<tr>
<td>rs/6000</td>
<td>PowerPC 601 based IBM RS/6000</td>
<td>128 MB</td>
<td>AIX 4.2</td>
<td>xlc 3.1</td>
</tr>
<tr>
<td>mips</td>
<td>100 Mhz MIPS R4000</td>
<td>64 MB</td>
<td>IRIX 6.2</td>
<td>SGI cc</td>
</tr>
<tr>
<td>sparc</td>
<td>50 Mhz, 4 processor SparcStation-20/514</td>
<td>512 MB</td>
<td>SunOS 5.5.1</td>
<td>SPARCCompiler C 3.0</td>
</tr>
</tbody>
</table>

Table 1: Test platforms.

Our first set of measurements was performed to examine the run-time overhead introduced by our code transformations. The transformations applied by APrIL add overhead to programs not only because they result in the execution of extra instructions, but also because they affect the ability of compilers to apply certain optimizations. The degree to which the APrIL code transformations affect performance is primarily a function of two factors: the policy for placing poll points in the code, and the characteristics of the code itself. To examine the effects of these factors, we applied three of the available set of heuristic poll-point-placement policies supported by APrIL to each of our test applications. The selected transformation heuristics were:

- **Mandatory**—no optional poll points placed, only those required for correct capture and recovery of the subroutine invocation stack.

- **Conservative**—in addition to mandatory poll points, this policy places an optional poll point after the last statement in the body each nesting loop (i.e., a loop containing at least one other loop in its body).

- **Aggressive**—in addition to those placed by the conservative policy, this policy places a poll point in each loop with more than one statement in its body. For example, whereas the conservative policy would not place a poll point in an innermost loop, this policy may perform such a placement.

We timed each of the test programs, first compiled without APrIL transformations, and then transformed using each of the above policies ("-O" optimization was performed in all cases). For the transformed versions, we measured the time to completion without capturing or recovering during execution (measured run times included time to load the process). In Figure 5 we display the relative performance of non-transformed and transformed test programs on the x86 platform. Additional results for the other test platforms
are presented in [7]—because all platforms exhibited roughly the same relative performance for the different transformed program versions, we present the timings for only one architecture here.

![Execution time of optimized example programs (std) compared to execution time of optimized transformed programs (mand, cons, aggr) on x86 platform. Times in seconds. Percent increase in run time is indicated for each transformed program.](image)

Figure 5: Execution time of optimized example programs (std) compared to execution time of optimized transformed programs (mand, cons, aggr) on x86 platform. Times in seconds. Percent increase in run time is indicated for each transformed program.

The first trend that we observe in these results is that the overhead of our transformations under the mandatory and conservative placement policies is generally low—well below 10% in all cases except for quicksort under the conservative policy. As expected, more aggressive placement can easily lead to high overhead. However, we note that the impact of the poll-point-placement policy is dependent on the application. Given our loop-structure based placement heuristics, the loop structure of an application will largely determine the performance implications of a given policy. Finally, we note that in all cases, at least one of the examined policies was able to achieve low net overhead.

Low introduced overhead, while important, is only half of the story. Recall, the frequency at which poll points are encountered not only affects overhead, but also determines the average amount of time that state capture requests would have to wait before being serviced. In our model, a state capture request is sent to the process, resulting in a global variable being set to indicate that state capture has been requested. It is only later, when the process reaches a poll point, that the state capture actually begins. This naturally leads
to the question, if a process is sent a state capture request, how long will it be before a poll point is reached and state capture begins? To investigate the state-capture-request wait time resulting from our system, we modified the APrIL compiler to instrument transformed programs at each poll point to keep a running count of the number of poll points encountered. Assuming an approximately equal distribution of poll points encountered over time, the average interval between poll points is simply the execution time divided by the poll point count. Based on measurements of the poll point counts, we present the computed average poll point interval for each program on the x86 test platform in Table 2.

<table>
<thead>
<tr>
<th>Policy</th>
<th>mm</th>
<th>gs</th>
<th>qs</th>
<th>ge</th>
<th>cg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory</td>
<td>3.7 sec.</td>
<td>0.42966</td>
<td>0.00026</td>
<td>1.2 sec.</td>
<td>5.16433</td>
</tr>
<tr>
<td>Conservative</td>
<td>0.05606</td>
<td>0.01057</td>
<td>0.00017</td>
<td>0.06234</td>
<td>0.20863</td>
</tr>
<tr>
<td>Aggressive</td>
<td>0.02769</td>
<td>0.00015</td>
<td>0.00016</td>
<td>0.06175</td>
<td>0.00022</td>
</tr>
</tbody>
</table>

Table 2: Average poll point interval (on x86 platform).
Times in milliseconds unless otherwise noted.

These results put the performance overheads presented in Table 5 in perspective. First, we note the correspondence between high poll point counts and high introduced overhead. This confirms our intuition that introduced overhead will be a function of the frequency at which poll points are encountered. Next, we note that optional poll point placement is important. For the matrix multiply and Gaussian elimination examples, the mandatory-only policy resulted in very few poll points, and thus very high average poll point intervals. This result is an artifact of the structure of these applications—each uses few function calls, and coarse, long-running loops—attributes that are not uncommon in high performance applications. Thus, we conclude that mandatory placement alone is insufficient for some applications.

Perhaps the most important result these measurements provide is that for each application, at least one of the examined policies resulted in both low introduced overhead (i.e. below 10%) and a small average poll point interval—generally below 0.1 millisecond. Since captured state will generally be written to stable storage (e.g. checkpoint/restart applications) or over a network (e.g. migration applications), poll point intervals of this duration are orders of magnitude less than the time required to perform state capture. Thus, for all applications, an acceptable level of overhead and very low on-average poll point interval were both possible with at least one policy. The implication of this fact is that Process Introspection can be applied
both automatically, and with good performance.

Our final set of experiments was performed to examine the efficiency of the state capture and recovery mechanisms. For these tests, we instrumented the PIL to note the time when either state capture or recovery was initiated, and to subsequently record the completion time. To obtain repeatable results, we also instrumented the PIL to automatically force a state capture request after a set number of poll points encountered during execution. We ran each of our transformed test applications (compiled with the conservative poll point placement policy) until 50000 poll points were encountered. At that point, a checkpoint was written to disk, and the process was terminated. We then used the checkpoint files produced on each platform to time a restart from disk. In Figure 6 we present the results for the x86 platform.

![Figure 6: State capture and recovery costs on x86 platform. Times in milliseconds; recovery time listed for checkpoints produced on each of the five test platforms. The captured state size for each test program is listed under the results for that program.](image)

We found that state capture and recovery costs on each platform were generally a function of state size and the I/O performance possible on the test platform. For example, in Table 6 we note that the time to capture the state of the Gaussian elimination program is roughly twice that required to capture the state of the conjugate gradient program, and there is approximately a factor of 2 difference in the programs' state sizes. An important result we notice is that the cost of restarting from a checkpoint produced on a platform with an incompatible data representation is greater than that of a compatible restart. For example, the times to perform state recovery from the incompatible rs/6000 format presented in Table 6 are generally about 40%
greater than the times required to restart using the native x86 format. The rs/6000 uses big-endian byte ordering, and thus extra time is required during state recovery to perform byte swapping. This result is not only important for making scheduling decisions (e.g. this might affect selection of a target host for migration), but also confirms our intuition that receiver-makes-right data conversion can improve performance.

5. Related Work

The idea of capturing the state of a running process on one kind of computer system and then later restarting an equivalent process on a different type of computer system has been the subject of a number of previous papers. Perhaps the most general coverage of this topic is presented by von Bank, Shub, and Sebesta in [25]. In this paper, the authors identify the general idea that a procedural computation can be modeled as progression through a sequence of compatible well-defined states: points in execution at which the state of a process can be used to fully describe the equivalent state of any other implementation of the process. In our model, these compatible well defined states are present in the form of process states when poll points are encountered. Related implementation work done by this group integrated a limited form of heterogeneous process migration into the Vsystem[5]. As is typical in existing approaches, this implementation relied on the operating system to examine and translate the state of the process.

A novel approach to the heterogeneous state capture/restore problem was proposed by Theimer and Hayes [24]. In their proposed solution, the state of a process is examined and captured using compiler-generated symbol mapping information. Instead of being captured in a data-only format that must be used in conjunction with a separate executable (a feature common in the other systems presented in this section, as well as our own), the process state is instead captured in the form of an intermediate code program. This program is constructed to re-initialize the full equivalent state of the captured process and proceed from its logical point of state capture. The actual process migration then consists of compiling this program on the destination machine. Such a mechanism would have the desirable property of requiring very little external support at the restart host (beyond the ability to recompile the intermediate code program). Our approach extends this desirable feature of autonomy to include state capture as well as state restore.

A more recent and fully implemented approach to the heterogeneous state capture problem was presented by Steensgaard and Jul in [23]. In this paper, the authors describe an extension of the thread- and
object-mobility capability of the heterogeneous Emerald distributed system to allow native code migration among heterogeneous hosts (previous implementations supported native code mobility for homogeneous hosts). In their implementation, native code threads can migrate at well-defined points during execution, called Bus Stops, at which time control is transferred to the Emerald run-time system, and a complete description of the running code is constructed by the system using compiler-generated mapping information (the same principle as used for symbolic debugging). This approach has the attractive property that modification to the generated code is not required; the compiler is simply responsible for generating the extra mapping information required by the run-time system. This approach differs from ours in exactly this respect—while we require modification of programs to support state capture and recovery, we do not require support from any external agent for this functionality. This affords us the desirable attribute of generality—our tool can be integrated into existing distributed systems without requiring modification to those systems or to our basic process state capture mechanism, and Process Introspection does not require extensive run-time system support. Our current implementation requires only that the system interface be accessible from C code, and that it be possible to construct a wrapper interface for system services that maintain external state for processes.

A similar approach to that of heterogeneous Emerald called Tui [20] has been proposed by Smith and Hutchinson. This approach also involves the use of compiler-generated state mapping information in the form of the symbol table typically used by symbolic debuggers. The Tui implementation has the additional desirable feature of supporting programs written in C. Again, this approach differs from Process Introspection in being external-agent-based—special programs are required to capture and restore the process state.

Recent work in the area of mobile agents has resulted in a number of state-capture and recovery mechanisms to support migration in mobile agent languages. For example, the Sumatra[1] language supports the capture and recovery of Java threads in a heterogeneous environment. State capture and recovery in Sumatra is achieved using through modifications to the Java Virtual Machine bytecode interpreter. A more flexible approach is supported by the Ara system[17]. As opposed to Sumatra which mandates used of the Java language, Ara supports mobile agents in an extensible set of interpreted languages, currently including interpreted C and Tcl. To support state capture of a running agent, the interpreters used in the system
must be able to capture their own full state (i.e. including the state of a program being interpreted). A primary drawback of these and many other mobile agent systems is the use of interpreted execution for agents. In our intended application domain, this model fails to meet the performance requirements of most users. A notable system that overcomes this limitation is Extended Facile[13], an agent programming system based on the Facile functional programming language. In Extended Facile, agents are first-class functions which may be transferred to remote nodes for execution. The code for agent functions in Extended Facile can be transferred in a higher-level, platform independent representation or as native-code executable instructions (or a mixture of the two). Extended Facile utilizes the continuation-based compilation model of the language to support state capture and recovery at function boundaries.

6. Conclusions

We have presented Process Introspection, a process-internal heterogeneous process state capture and recovery mechanism based on automatic code modification. Experiences with this system have produced encouraging results. First, we found that relatively simple poll-point-placement policies can achieve acceptable levels of incurred overhead while at the same time providing good performance in terms of average checkpoint-request wait time. This result is important—process internal state capture and recovery made possible by periodic polling can be utilized effectively, efficiently. Furthermore, the design of our system demonstrates that process-internal state capture and recovery need not place undue burden on the programmer—the typical usage mode for our system is fully automatic, requiring only an additional compiler translation of the user’s application program.

We believe our mechanism is general and widely applicable in a variety of different distributed system environments. For example, we are currently working on adapting the system for use in the Legion [13] metacomputing system, and are investigating integration into a PVM [8] or MPI [10] system. This adaptability is explicitly supported by our PIL API which provides a medium for APrIL-transformed modules and hand-coded system-interface wrapper modules to interoperate. Furthermore, we have designed extensions for our system to handle additional programming constructs such as threads, and languages such as Fortran and C++. These designs (presented in [7]) are the subject of ongoing development and evaluation.
References


Grid-Based File Access: The Legion I/O Model

Brian S. White       Andrew S. Grimshaw       Anh Nguyen-Tuong
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{bsw9d,grimshaw,an7s}@cs.virginia.edu

Abstract

The unprecedented scale, heterogeneity, and varied usage patterns of grids pose significant technical challenges to any underlying file system that will support them. While grids present a host of new concerns for file access, we focus on two issues: performance and usability. We discuss the Legion I/O model and interface to address the latter area. We compare Legion and Globus I/O against a baseline to validate the efficiency of existent grid-based file access solutions.

1. Introduction

The advent of high-speed networks coupled with a desire to harness more processing power and access immense data stores has lead to the possibility and necessity of federation such resources into computational grids [6, 7, 5]. The unprecedented scale, heterogeneity, and varied usage patterns of such grids pose significant technical challenges to any underlying file system that will support them. Many well-known distributed file system problems, such as security, scalability, performance, and usability, are exacerbated by a grid environment.

Based on our experiences with grids and their users, we highlight two aspects of grid-based file systems: performance and usability. Grid file systems face the daunting task of providing low response times and high throughput in a wide-area environment. Usability is an acute concern because the computational resources of grids are attractive for legacy scientific applications. It may be impossible or exceedingly difficult to re-tool these applications for a grid environment.

Some users will want to be sheltered from the grid environment. Ideally, their method of system interaction (both at the command line and API) need not change from whichever system they are accustomed. On the opposite end of the computing spectrum are sophisticated users and application programmers who will prefer a rich interface to take full advantage of the grid’s potential. A grid environment should accommodate both types of users.

This paper examines the I/O infrastructure and performance of Legion, comparing it with Globus. Legion [7] is an object-based grid operating system charged with reconciling a collection of heterogeneous resources, dispersed across a wide-area, with a single virtual system image. Legion provides resource management, scheduling, and other system-level tasks, as does any operating system; however, it does so on a much wider scale. Built from the ground up, Legion addresses such issues as scalability, programming ease, fault tolerance, security, and site autonomy.

Legion provides a remote access capability which attempts to address the standards delineated above. We show that Legion achieves 55-65% of ftp’s write bandwidth and 70-85% of ftp’s read bandwidth for mass transfers. For transfer sizes less than 1 MB, Legion performance suffers owing to its protocol overhead.

Related work is summarized in Section 2. In Section 3, we discuss the Legion I/O model and define terminology. To address the issue of usability, we present the Legion I/O interfaces in Section 4. Next, we present server implementations in Section 5, to motivate a discussion of their performance in Section 6. Finally, we conclude with Section 7.

2. Related work

While space does not permit a detailed scrutiny of the rich literature in distributed file systems research, we do note the excellent survey by Levy and Silberschatz [11].

I/O performance in the wide area is highly sensitive to transfer size. Initial implementations of NFS [13, 14] artificially restricted transfer sizes based on the virtual memory architecture. APS [9] transfers and copies entire files, leading to unnecessary traffic when a dataset is partitioned between multiple distributed workers. Further, the cache
consistency mechanisms of NFS (regular calls to retrieve attributes via GETATTR) and AFS (callbacks) limit throughput.

Globus [5] GASS provides access to remote files through x-gass, ftp, or HTTP protocols [2]. The HTTP GASS implementation is measured alongside Legion in Section 6. We note that GASS provides a secure transport over HTTPS, which is not treated here.

GASS stages remote files to a locally-accessible file system on first open via the globus_gass_open function. GASS utilizes whole-file caching so that subsequent operations may be satisfied locally through standard system calls. If a file is opened for writing, it is copied back to its remote store when all active file descriptors are closed via globus_gass_close. Under the x-gass protocol, GASS also supports a streaming append operation.

From a user’s perspective, the specialized calls are cumbersome. However, files may be manually pre-staged to avoid any necessary changes to legacy codes. The location-dependent URL file name scheme used by GASS is an additional burden.

3. The Legion I/O model

Legion objects represent resources and are active entities (e.g. each may be a process running in a separate address space). For example, HostObjects represent computational resources running in a Legion system and VaultObjects are responsible for maintaining state associated with Legion objects. BasicFileObjects correspond to files in a conventional file system and ContextObjects are analogous to a distributed, rooted directory tree [8].

Legion provides its users with human-readable context names. The name-space is hierarchical and rooted, but disjoint from the Unix file system. Context names are translated to location-independent identifiers called LOIDs (Legion Object Identifiers), via ContextObjects. In order to communicate with an object, Legion must bind LOIDs to OAs (object addresses), which describe the actual communication endpoint [8].

Objects export interfaces and are characterized and classified according to that interface. For example, any object implementing the BasicFileObject interface is treated as a BasicFileObject. This ability to override methods allows for specialization and extensibility.

4. User interface

In an effort to ease the transition to a grid environment, Legion presents its users with a set of familiar and intuitive interfaces (e.g. command line utilities such as legion ls and C library counterparts such as BasicFiles.creat()). In addition, more powerful interfaces, such as a two-dimensional parallel file interface are available.

We recognize that any required change in legacy code is undesirable and, in cases where source code is unavailable, infeasible. While a kernel-based Legion file system would be accessible by standard system calls, we avoid operating system modifications to achieve portability. We can achieve a similar effect by interposing a Legion-aware NFS daemon.

4.1. Command line utilities

The Legion command line utilities [16] allow a user to navigate context space and manipulate its structure and the objects it contains in a manner reminiscent of traversing a rooted directory tree. This is accomplished through commands which share a similar look and feel to their Unix-like equivalents. For example, legion ls lists a context. legion_cat displays the contents of a BasicFileObject. legion_cp copies files within context space or between context space and a native file system. A bevy of other command line tools mirror the remaining traditional Unix utilities.

Two more commands aid in bridging the gap between context space and traditional file systems. legion_import_tree recursively copies a local directory tree, creating a Legion object for each subdirectory or file. legion_export_dir is a light-weight utility that makes a Unix directory visible in context space, without creating stand-alone objects for each contained file and subdirectory. Updates effected via Legion mechanisms are immediately reflected to the underlying file system. Support for this utility is described in Section 5.2.

4.2. Remote I/O interface

The Legion file interface library provides user programs with access to BasicFileObjects [15]; bindings exist for C, C++, and Fortran. To match a user’s needs and level of familiarity with Legion programming, we also provide C-like (Section 4.2.1) and low impact (Section 4.2.2) I/O interfaces.

4.2.1. Basic I/O interface

The basic I/O interface includes functions inspired by the C I/O system calls and buffered I/O library. This interface is a wrapper around the BasicFileObject implementation accessible from C or C++. By providing communication stubs, the interface relieves the programmer of the burden of accessing the BasicFileObject directly via the Legion communication primitives. Function names are prefixed by BasicFiles, but otherwise follow the naming and argument conventions of their C counterparts.
Owing to the overhead of the unoptimized Legion protocol stack, fine-grained file accesses are relatively expensive. To reduce the frequency of remote procedure calls, applications can use the buffered interface. Further, Fortran bindings for buffered I/O provide interoperability when reading and writing integers, reals, and doubles.

4.2.2. Low impact buffered interface

The Legion low impact interface is aimed at minimizing changes to legacy codes wishing to access context space. An application writer uses lio_legion_to_tempfile to transfer the contents of BasicFileObjects into the local file system. The application may then utilize standard system calls to access the file. Finally, any modified files are copied back to context space via lio_tempfile_to_legion, in a technique similar to GASS staging [2].

Note that it is possible to avoid making any changes to legacy codes by importing the Legion BasicFileObject before the application executes, and copying the data back on exit. This technique is used frequently by our legacy applications.

4.3. Legion-aware lnsd

To remove the copy-in,copy-out requirements of the previous mechanism, we submit a less obtrusive approach. In this section, we describe the interposition of an NFS daemon between the kernel client and Legion, which allows unmodified applications to access context space.

4.3.1. Implementation

Our modified, Legion-aware NFS daemon, lnsd, receives NFS requests from the kernel and translates these into the appropriate Legion method invocations. Upon receiving the results, it packages them in a form digestible by the NFS client. The file system is mounted like any NFS file system.

To service a user’s request, the daemon must have the same rights as that user. Legion stores a user’s credentials in the /tmp file system, accessible only to that user (and root). The daemon runs as a privileged process in order to read a user’s credentials from disk and package them with messages sent on that user’s behalf. The daemon is able to usurp the user’s rights because Legion utilizes bearer credentials [4] or proxies [12]. Fully aware of the security implications of this approach, we are examining delegation, which restricts credentials to specific users and/or operations.

To prevent the performance degradation inherent in small data transfers, lnsd attempts to communicate with Legion objects using a larger granule. It employs read-ahead to avoid costly demand fetches. lnsd reads ahead only when the current request is within the read ahead window. When a user’s request cannot be satisfied by lnsd’s cache, the daemon prefetches synchronously by appending the read ahead request to the demand request. Otherwise, prefetching is done asynchronously.

To avoid small write transfers, lnsd utilizes asynchronous write-behind. Data are flushed from the cache after a configurable delay. Whenever data must be written back to the corresponding Legion object, lnsd attempts to coalesce contiguous blocks.

The execution of duplicate requests, caused by client retransmission, can hurt performance and lead to incorrectness (for non-idempotent requests). To combat these problems, lnsd maintains a request cache [10, 3].

4.3.2. Security

lnsd is derived from an NFS Version 2 [14] user-space server. This early protocol focused on the fundamentals of remote access. Under the most popular and basic authentication mechanism, security in NFS is predicated on mutual trust between kernels and/or root-privileged processes. In such an environment of trust, sophisticated authentication protocols are unnecessary. A client simply supplies the server with a UID on each transaction; the server blindly and blithely obliges. To prevent the most obvious security abuses, an exports file limits the domain of trust to certain hosts. Further, the server can be configured to accept only connections from reserved ports.

Unlike NFS, Legion has targeted security as a major concern from its inception. In providing the flexibility afforded by NFS, it is imperative that we do not introduce any new insecurities. Unfortunately, we are restricted in our ability to maintain this level of security by the fixed interface between the NFS (kernel) client and lnsd. Nevertheless, we are confident that a number of simplifications and assumptions allow us to meet our goal. These are:

1. Legion users on a Legion host trust privileged processes on that host.
2. lnsd only accepts connections made from a reserved port.
3. The NFS client and lnsd are collocated on the same host.

Because a user’s credentials reside on disk, Legion is susceptible to rogue root users. This fact leads us to make the first assertion. However, this assumption is not unique to the Legion NFS implementation and so doesn’t open any new portal for intrusion.

The second item requires that a request be made by a kernel or some other privileged process. (Note that we could
easily spoof the source port, but doing so would require being root or modifying a kernel on some host. This allows us to rule out any attacks by non-privileged users.

A client needs a valid file handle to transact with the lfnisd. (The root file handle is provided during the mount operation.) Due to the third point, lfnisd will only send file handles to addresses on the local host. Therefore, forging an IP address does not aid an intruder as the lfnisd will not expose file handles on the wire. This ensures that they can not be sniffed by a malicious user.

4.4. Parallel I/O interface

Legion supports a parallel file interface [15]. This interface allows user-specified striping of data across BasicFileObjects. Such an organization allows multiple clients to access the data without contending with one another at a central server object. Secondly, individual client performance benefits because multiple BasicFileObjects may be accessed concurrently to deliver the desired data.

5. Server implementations

Having highlighted Legion’s I/O interfaces, we now describe the implementations that store and manage user data: BasicFileObjects and ProxyMultiObjects. These objects export the BasicFileObject interface and are accessible through the above high-level libraries and interfaces.

5.1. BasicFileObject

A BasicFileObject is a file server, which serves exactly one file. The BasicFileObject polls for incoming requests in a server loop (see Figure 1).

The contents of the BasicFileObject are housed within a LegionBuffer, a random-access array. LegionBuffers provide interoperability between different architectures by encoding metadata on the representation of their contents and performing automatic conversions. Thus, a user need not be concerned with word sizes, byte order, or float-point representations.

BasicFileObjects utilize persistent LegionBuffers which store their data in a VaultObject. This data is not meant to be directly manipulated by the user and has no correspondence to any file rooted in a user’s directory tree. Users may copy data from a Unix file to a BasicFileObject in context space. However, the contents of the file and BasicFileObject are independent and may diverge.

5.2. ProxyMultiObject

A ProxyMultiObject encapsulates a context subtree, and therefore serves both ContextObject and BasicFileObject requests. Like the BasicFileObject, it utilizes a single server loop within a single address space. However, the server demultiplexes messages to contained representations of ContextObjects and BasicFileObjects (see Figure 2).
In all experiments, data are transferred from (written to) an NFS mount on the Origin array. That is, the files involved in the ftp and GASS experiments, and the persistent state of the LegionBuffers in the Legion evaluations, reside on an NFS mount. By invalidating the relevant Irix file system buffer cache entries, we ensure that all writes are flushed on file close and that file data does not persist in the cache across trials. Likewise, we prevent client-side caching to ensure remote access to the server objects. Performance measurements do not capture client-side disk accesses.

By default, the Legion communication layer utilizes a start-and-stop protocol built atop UDP. To ensure a fair basis of comparison, we present results using UDP and TCP. In both sets of experiments, security is disabled (as is also true of the GASS runs). Were security enabled, message digesting and certificate passing would contribute additional overhead.

Experiments are characterized from the perspective of the Linux host at the University of Virginia. Therefore, during read experiments, the Linux host reads data from NCSA and simply discards the transferred data. On writes, the Linux host transfers data from memory to NCSA, where they are flushed to disk.

6.1. Protocol overhead

Our first test seeks to expose the fixed cost of connection setup and tear-down of each mechanism (Table 1).

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Latency (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp-read</td>
<td>0.028</td>
</tr>
<tr>
<td>ftp-write</td>
<td>0.026</td>
</tr>
<tr>
<td>gass-read</td>
<td>0.087</td>
</tr>
<tr>
<td>gass-write</td>
<td>0.147</td>
</tr>
<tr>
<td>legion-read-udp</td>
<td>0.325</td>
</tr>
<tr>
<td>legion-write-udp</td>
<td>0.487</td>
</tr>
<tr>
<td>lio-read-udp</td>
<td>0.507</td>
</tr>
<tr>
<td>lio-write-udp</td>
<td>0.631</td>
</tr>
<tr>
<td>lnfs-read-udp</td>
<td>0.454</td>
</tr>
<tr>
<td>lnfs-write-udp</td>
<td>0.681</td>
</tr>
<tr>
<td>legion-read-tcp</td>
<td>0.430</td>
</tr>
<tr>
<td>legion-write-tcp</td>
<td>0.632</td>
</tr>
<tr>
<td>lio-read-tcp</td>
<td>0.501</td>
</tr>
<tr>
<td>lio-write-tcp</td>
<td>0.677</td>
</tr>
<tr>
<td>lnfs-read-tcp</td>
<td>0.403</td>
</tr>
<tr>
<td>lnfs-write-tcp</td>
<td>0.422</td>
</tr>
</tbody>
</table>

Table 1: File Open Overhead

Cache locking and management contribute overhead to the GASS protocol [2].

\[1\] In this and subsequent figures, legion refers to the Legion basic I/O interface (Section 4.2), lio denotes the Legion low impact interface (Section 4.2.2), and lnfs signifies Legion-aware NFS (Section 4.3).
The overhead of the Legion mechanisms is significant, and attributable primarily to Legion’s location-independent naming scheme. Several context name/LOID translations and LOID/DA bindings are required during this test. In a longer-lived application, these translations and bindings would likely be cached and amortized over a number of Legion RPCs.

In addition to this naming overhead, the Legion basic I/O and low impact interfaces make an expensive (and superfluous) call to the remote BasicFileObject at NCSA. The Legion NFS implementation performs a GETATTR RPC to retrieve the BasicFileObject’s attributes. It also performs a less expensive invocation within the LAN to determine the attributes of the /tmp context.

### 6.2. Bandwidth measurements

In this section, we present file transfers of various sizes. Through these experiments we gain an appreciation for the throughput achieved by each mechanism.

The first set of figures compare UDP- and TCP-based Legion mechanisms. Figure 3 clearly shows each TCP mechanism outperforming its UDP counterpart for read operations. In each case, the low impact interface performs similarly to the basic I/O interface. This is not surprising as each mechanism is simply reading data in one megabyte chunks. Governed by the NFS protocol, Infsd periodically queries the remote BasicFileObject to satisfy GETATTR requests. Further, the maximum transfer size requests by the NFS kernel client corresponds to a page (4K on a Pentium II). Thus throughput between the kernel and Infsd is limited. For these reasons, Legion NFS performance suffers.

Figure 4 highlights Legion write performance. Unlike the read case, we notice that the low impact interface performs significantly worse than the basic I/O interface under both TCP and UDP. This is to be expected; the file must first be fetched from the remote server, modified locally, and finally written back. Because of this additional data copying, Legion NFS outperforms the low impact interface over TCP.

In Figures 3 and 4, the performance of TCP implementations surpasses the corresponding UDP mechanism. Though TCP outperforms UDP in this wide-area environment, since TCP entails a higher overhead than UDP on a per-connection basis, we would not want to blindly deploy TCP in local-area networks. Future releases of Legion will support dynamic selection of the network transport (TCP or UDP) based on IP addresses.

A comparison of Figure 3 with Figure 4 and Figure 5 with 6 show that writes outstrip reads over TCP. From a file system perspective, a read access may be semantically quite different from a write operation. However informal experiments removed all disk accesses, so that reads and writes were simple network transfers differing only in transfer direction. The anomalies witnessed above did not disappear. This manifests the potential instability and non-uniformity of the environments in which wide-area file systems will need to thrive.

Figure 5 compares Legion read mechanisms against the corresponding GASS and ftp operations. Figure 6 treats writes similarly.

In the case of reads, GASS closely mirrors ftp. This is expected as GASS is a thin veneer over the simple HTTP protocol. The Legion basic I/O and the copy-in/copy-out mechanisms approach the bandwidth of ftp to within 70-85%. Legion’s degraded performance is likely attributable to its extendible protocol stack.

Legion NFS performance lags significantly, achieving 30% of ftp’s throughput for transfers greater than a megabyte. For small transfer sizes, Legion NFS latencies are dominated by LOOKUPS of directory path components and GETATTR RPC calls. Given its readahead potential, we would not expect Legion NFS performance to be so abysmal under mass transfers. We hypothesize that as Infsd’s block cache saturates, the implementation is burdened by unoptimized and frequent cache lookups.

For writes, the Legion basic I/O interface outperforms both GASS and the Legion low impact interface. As described above, this behavior is to be expected as the Legion basic I/O interface avoids the copy-in/copy-out semantics of the other two mechanisms.

Legion NFS write performance is consistent with its read bandwidth. A sophisticated block cache contributes considerable overhead and once again appears to be the source of poor write throughput. With a stream-lined cache implementation, Legion NFS should be able to leverage asynchronous write-behind to achieve performance similar to that exhibited by the Legion I/O interface.

Once again, it is interesting to note that ftp writes are significantly more efficient than ftp reads. This suggests that the asymmetric anomaly described above is not a Legion artifact, but rather attributable to the network configuration of our heterogeneous test environment.

The graphs show that mass transfers are more efficient because they incur fewer invocations of the respective protocol and its associated overhead. For example, the overhead of a NULL RPC in Legion has been measured to be 8 milliseconds. This makes small data transfer costly.

### 7. Conclusions

We presented the Legion I/O interfaces, highlighting their flexibility and intended audiences. The low impact interface and NFS implementations provide support for legacy codes. The Unix-like I/O library and command line
utilities provide a familiar model for many users, easing their transition to a grid environment.

A presentation of Legion and GASS mechanisms compared against a baseline allows us to judge the current performance of grid-based I/O systems. We find that the Legion basic I/O interface (TCP implementation) achieves 55-65% of ftp's efficiency for writes and 70-85% for writes. While NFS Legion performance lags, the ease of use it affords may make it a viable option. Further, the implementation is immature, with several optimizations planned.

Equipped with an understanding of grid-based I/O performance relative to more traditional remote access, we expect to attack the I/O bottleneck from various angles. Our first efforts will quantify the time spent in the Legion protocol. The Legion I/O model offers an opportunity to tailor files to specific file access patterns. This could easily be supported by attribute value tags, such as 'read-only' or 'single-writer'. Taking such properties into account will have a significant performance impact.

Acknowledgments

This work was partially supported by National Computational Science Alliance under grant S-26025 and utilized the NCSA SGI/CRAY Origin2000. We would also like to thank Mark Morgan for answering a constant stream of questions and John Karpovich who implemented the dynamic network protocol selection in Legion. Marty Humphrey, Fritz Knabe, and Robert Schutt provided valuable insights regarding security in nfsd.

References

Figure 4. Legion Write Bandwidth


Figure 5. Read Bandwidth

Figure 6. Write Bandwidth
Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem

Holly Dail*  Graziano Obertelli*
Francine Berman*  Rich Wolski†  Andrew Grimshaw‡

* Computer Science and Engineering Department
University of California, San Diego
[hdail, graziano, berman]@cs.ucsd.edu

† Department of Computer Science
University of Tennessee
rich@cs.utk.edu

‡ Department of Computer Science
University of Virginia
grimshaw@virginia.edu

Abstract

Computational Grids have become an important and popular computing platform for both scientific and commercial distributed computing communities. However, users of such systems typically find achievement of application execution performance remains challenging. Although Grid infrastructures such as Legion and Globus provide basic resource selection functionality, work allocation functionality, and scheduling mechanisms, applications must interpret system performance information in terms of their own requirements in order to develop performance-efficient schedules.

We describe a new high-performance scheduler that incorporates dynamic system information, application requirements, and a detailed performance model in order to create performance-efficient schedules. While the scheduler is designed to provide improved performance for a magnetohydrodynamics simulation in the Legion Computational Grid infrastructure, the design is generalizable to other systems and other data-parallel, iterative codes. We describe the adaptive performance model, resource selection strategies, and scheduling policies employed by the scheduler. We demonstrate the improvement in application performance achieved by the scheduler in dedicated and shared Legion environments.

1. Introduction

Computational Grids [7] are rapidly becoming an important and popular computing platform for both scientific and commercial distributed computing communities. Grids integrate independently administered machines, storage systems, databases, networks, and scientific instruments with the goal of providing greater delivered application performance than can be obtained from any single site. There are many critical research challenges in the development of Computational Grids as an effective computing platform. For users, both performance and programmability of the underlying infrastructure are essential to the successful implementation of applications in Grid environments.

The Legion Computational Grid infrastructure [11] provides a sophisticated object-oriented programming environment that promotes application programmability by enabling transparent access to Grid resources. Legion provides basic resource selection, work allocation, and scheduling mechanisms. In order to achieve desired performance levels, applications (or their users) must interpret system performance information in terms of requirements specific to the target application. Application Level Scheduling (AppLeS) [3] is an established methodology for developing adaptive, distributed programs that execute in dynamically changing and heterogeneous execution settings. The ultimate goal of this work is to draw upon the AppLeS and Legion Computational Grid research efforts to design an adaptive application scheduler for regular iterative stencil codes in Legion environments.

We consider a general class of regular, data-parallel stencil codes which require repeated applications of relatively...
constant-time operations. Many of these codes have the following structure:

Initialization

Loop over an n-dimensional mesh

Finalization

in which the basic activity of the loop is a stencil based computation. In other words the data items in the n-dimensional mesh are updated based on the values of their nearest neighbors in the mesh. Such codes are common in scientific computing and include parallel implementations of matrix operations as well as routines found in packages such as ScaLA-PACK [18].

In this paper we focus on the development of an adaptive strategy for scheduling a regular, data-parallel stencil code called PMHD3D on the Legion Grid infrastructure. The primary contributions of this paper are:

- We describe an adaptive performance model for PMHD3D and demonstrate its ability to predict application performance in initial experiments. The performance model represents the application’s requirements for computation, communication, overhead, and memory, and could easily be extended to serve more generally as a framework for regular iterative stencil codes in Grid environments.

- We couple the PMHD3D performance model with resource selection strategies, schedule selection policies, and deployment software to form an AppLeS scheduler for PMHD3D.

- In order to satisfy the requirements of the PMHD3D performance model we implement and utilize a new memory sensor as part of the Network Weather Service (NWS) [22]. The sensor collects measurements and produces forecasts of the amount of free memory available on a processor.

- We demonstrate the ability of the AppLeS methodology to provide enhanced performance for the PMHD3D application, using the Legion software infrastructure as a platform for high-performance application execution.

In the next section we discuss the structure of the target application and the environment that we used as a test-bed. In Section 3, we discuss the AppLeS we have designed for PMHD3D and provide a generalizable performance model. Section 4 provides experimental results and demonstrates performance improvements we achieved via AppLeS using Legion. In Sections 5 and 6 we review related work and investigate possible new directions, respectively.

2. Research Components: AppLeS, NWS, PMHD3D and Legion

In order to build a high-performance scheduler for PMHD3D we leveraged application characteristics, dynamic resource information from NWS, the AppLeS methodology, and the Legion system infrastructure. In this section we explain each of these components in detail.

2.1. AppLeS

The AppLeS project focuses on the development of a methodology and software for achieving application performance via adaptive scheduling [1]. For individual applications, an AppLeS is an agent that integrates with the application and uses dynamic and application-specific information to develop and deploy a customized adaptive application schedule. For structurally similar classes of applications, an AppLeS template provides a “pluggable” framework which comprises a class-specific performance model, scheduling model, and deployment module. An application from the class can be instantiated within the template to form a performance-oriented self-scheduling application targeted to the underlying Grid resources.

AppLeS schedulers often rely on available tools in order to deploy the schedule or to gather information on resources or environment. AppLeS commonly depends on the Network Weather Service (NWS) (see Section 2.4) to provide dynamic predictions of resource load and availability. Together, AppLeS and the Network Weather Service can be used to adapt application performance to the deliverable capacities of Grid resources at execution time. In this project AppLeS uses Legion to execute a schedule and the Internet Backplane Protocol (IBP) [13] to effectively cache the data coming from NWS.

2.2. PMHD3D

The target application for this work, PMHD3D [12, 15], is a magnetohydrodynamics simulation developed at the University of Virginia Department of Astronomy by John F. Hawley and ported to Legion by Greg Lindhal. The code is an MPI FORTRAN stencil-based application and shares many characteristics with other stencil codes. The code is structured as a three-dimensional mesh of data, upon which the same computation is iteratively performed on each point using data from its neighbors. PMHD3D alternates between CPU-intensive computation and communication (between “slab” neighbors and for barrier synchronizations).

At startup PMHD3D reads a configuration file that specifies the problem size and the target number of processors. Since the other two dimensions are fixed in PMHD3D’s three-dimensional mesh, we refer to the height of the mesh
as the *problem size*. In order to allocate work among processors in the computation the mesh is divided into horizontal slabs such that each processor receives a slab. For load balancing purposes each processor can be assigned a different amount of work (by dividing the work into slabs of varying height). The AppLeS scheduler determines the optimal height of each slab depending on the raw speed of the processor and on NWS forecasts of CPU load, the amount of free memory, and network conditions. AppLeS is dynamic in the sense that the data used by the scheduler is computed and collected just before execution, but once the schedule is created and implemented, the execution currently proceeds without interaction with the AppLeS.

2.3. Legion

Legion, a project at the University of Virginia, is designed to provide users with a transparent, secure, and reliable interface to resources in a wide-area system, both at the programming interface level as well as at the end-user level [9, 14]. Both the programmer and the end-user have coherent and seamless access to all the resources and services managed by Legion. Legion addresses challenging issues in Computational Grid research such as parallelism, fault-tolerance, security, autonomy, heterogeneity, legacy code management, resource management, and access transparency.

Legion provides mechanisms and facilities, leaving to the programmer the implementation of the policies to be enforced for a particular task. Following this idea, scheduling in Legion is flexible and can be tailored to suit applications with different requirements. The main Legion components involved in scheduling are the collection, the enactor, the scheduler, and the hosts which will execute the schedule [5]. The collection provides information about the available resources and the scheduler selects the resources to be used in a schedule. The schedule is then given to the enactor, which contacts the host objects involved in the schedule and attempts to execute the application. This scheme provides scheduling flexibility; for example, in case of host failures, the enactor can ask the scheduler for a new schedule and continue despite the failure, the collection can return subsets of the resources depending on the user and/or the application, or the hosts can refuse to serve a specific user.

Legion currently provides default implementations of all the objects described herein. Moreover, new objects can be developed and used rather than the default ones. Note that the PMHD3D AppLeS is developed "on top" of Legion, and uses default Legion objects. We would expect the performance improvement for such a code to conservatively bound from below that which would be achievable if the AppLeS were structured as a Legion object. We plan to eventually develop the AppLeS described here as a Legion scheduling object for a class of regular, iterative, data-parallel applications.

2.4. Network Weather Service

The Network Weather Service [17, 22] is a distributed system that periodically monitors and dynamically forecasts the performance various network and computational
resources can deliver. NWS is composed of sensors, memories and forecasters. Sensors measure the availability of the resource, for example CPU availability, and then record the measurement in a NWS memory. In response to a query, the NWS software will return a time series of measurements from any activated sensor in the system. This time series can then be passed to the NWS forecaster which predicts the future availability of the resource. The forecaster tests a variety of predictors and returns the result and expected error of the most accurate predictor. To obtain better performance for PMHD3D we developed a memory sensor that measures the available free memory of a machine. The sensor has been extended and is now part of NWS.

2.5. Interactions Among System Components

PMHD3D can directly access Legion’s scheduling facilities or can use AppLeS to obtain a more performance-efficient schedule. Figure 1 shows the interactions among components in each of these scenarios. The dotted line represents the scheduling of a PMHD3D run without AppLeS facilities: the user supplies the number of processors, the processor list, and the associated problem size per processor and the rest of the scheduling process is supplied by a default scheduler within the Legion infrastructure.

When the application uses AppLeS for scheduling, the interactions among components can instead be represented by the solid lines in Figure 1. In this case the user supplies only the problem size of interest. AppLeS collects the list of available resources from the environment (via the Legion collection object or, in our case, via the Legion context space), and then queries NWS to obtain updated performance and availability predictions for the available resources. As the figure shows, AppLeS collects the NWS predictions as an IBP client: the predictions are pushed into the IBP server by a separate process.

AppLeS then creates a performance-promoting adaptive schedule and asks the Legion scheduler to execute it. The schedule is adaptive because AppLeS assigns a different amount of work to each processor depending on their predicted performance. As is suggested by the figure, the PMHD3D AppLeS is built on top of Legion facilities. A future goal is to integrate the AppLeS as an alternative scheduler in Legion for the class of regular, data-parallel, stencil applications.

3. The PMHD3D AppLeS

The general AppLeS approach is to create good schedules for an application by incorporating application specific characteristics, system characteristics, and dynamic resource performance data in scheduling decisions. The PMHD3D AppLeS draws upon the general AppLeS methodology [3] and the experience gained building an AppLeS for a structurally similar Jacobi-2D application [2].

Conceptually, the PMHD3D AppLeS can be decomposed into three components:

- a performance model that accurately represents application performance within the Computational Grid environment;
- a resource selection strategy that identifies potentially performance-efficient candidate resource sets from those that are available at run time;
- a schedule creation and selection strategy that creates a good schedule for each of the various candidate resource sets and then selects the most performance-efficient schedule.

The overall strategy and organization of the scheduler will be discussed here but the details of each component are reserved for the following sections.

An accurate performance model (Section 3.1) is fundamental for the development of good schedules. The performance model is used in two important ways, the first of which is to guide the creation of schedules for specific resource sets. For example, load balancing is a necessary condition developing an efficient schedule but is difficult or impossible to achieve without an estimate of the relative costs of computation on various resources. An accurate performance model is also necessary for selection of the highest performance schedule from a set of candidate schedules.

The resource selection strategy (Section 3.2) produces several orderings of available resources based on different concepts of “desirability” of resources to PMHD3D. Our definitions of desirability incorporate Legion resource discovery results, dynamic resource availability from NWS, dynamic performance forecasts from NWS, and application-specific performance data for each resource. Once complete, the ordered lists of resources are passed on to the schedule creation and selection component of the AppLeS.

The schedule creation step (Section 3.3) takes the proposed resource lists and creates a good schedule for each based on the constraints the system and application impose. System constraints are characteristics such as available memory of the resources while the application constraints are characteristics such as the amount of memory required for the application to remain in main memory. Once all schedules have been created the performance model is used to select the highest performance schedule (the one in which the execution time is expected to be the lowest).

The decomposition of the scheduling process into these disjoint steps provides an overly simplistic view of the interactions between steps. In reality the scheduling process
1 Rset = getResourceSet()
2 NWS_data = NWS(Rset)
3 C = getScheduleConstraints()
4 for (balance = [0, 0.5, 1])
5    S = sort(Rset, balance, maxP)
6 for (n = 2..maxP)
7      sched = findSched(n, S, NWS_data, C)
8      while (sched is not found)
9         "Schedule constraints are too restrictive"
10         relaxConstraints (C)
11         sched = findSched(n, S, NWS_data, C)
12         endwhile
13      if (cost(sched) < best)
14         best = sched
15      endif
16  endfor
17 endfor
18 run(best)
\% Available resources obtained from Legion
\% NWS forecasts of resource performance
\% Obtain scheduling constraints for simplex
\% Select for CPU power, connectivity, both
\% Returns list of hosts sorted by desirability
\% Searching for correct number of processors
\% Use simplex to find schedule on S using C
\% Simplex was unsolvable with S and C
\% More schedule flexibility, more possible error
\% Try to find schedule again
\% Found a feasible schedule
\% If best one so far keep it, else throw away
\% Best schedule found, run it

Figure 2. PMHD3D AppLeS pseudo-code.

requires more complicated interactions. To accurately represent the true interaction of the scheduling components we present a pseudo-code version of the PMHD3D AppLeS strategy in Figure 2. The steps shown in Figure 2 will become clearer in the following sections.

3.1. Performance Model

The goal of the performance model is to accurately predict the execution time of PMHD3D. Since the run-time may vary somewhat from processor to processor, we take the maximum run-time of any processor involved in the computation as the overall run-time. During every iteration each processor computes on its slab of data, communicates with its neighbors, and synchronizes with all other processors.

Formally, the running time for processor $i$ is given by:

$$T_i = Comp_i + Comm_i + Over_i$$

where $Comp_i$, $Comm_i$ and $Over_i$ are the predicted computation time, the predicted communication time, and the estimated overhead for $P_i$, respectively.

Computation time is directly related to the units of work assigned to a processor (in other words the height of the slab) and to the speed of that processor. The computation time for $P_i$ is:

$$Comp_i = \frac{x_i \times BM_i}{Avail_i}$$

where $x_i$ is the amount of work allocated to processor $P_i$ (dynamically determined by the scheduling process), $BM_i$ is a benchmark for the application-specific speed of $P_i$'s processor configuration, and $Avail_i$ is a forecast of the CPU load on processor $P_i$ (obtained from dynamic NWS forecasts). To obtain the benchmarks, we run PMHD3D on dedicated machines with various problem sizes and variable number of hosts. Execution times were proportional to problem size and are given in terms of seconds per point on each platform.

Communication time is modeled as the time required for transferring data to neighboring processors across the available network. This represents communication for all iterations and accounts for both the time to establish a connection and the time to transfer the messages. To simplify the communication model, we have not attempted to directly predict synchronization time or the time a processor waits for a communication partner. We hope instead to capture the effect of these communication costs in our estimate of overhead costs, which we discuss shortly. Communication time is then:

$$Comm_i = MB/(b_{i,i+1} + b_{i,i-1}) \times \frac{(i_{i+1} + i_{i-1})}{M}$$

where $MB$ is the total megabytes transferred, $M$ is the number of messages transferred, and $b_{ij}$ and $l_{ij}$ are predictions of available bandwidth and latency from $P_i$ to $P_j$, respectively. Predictions of available bandwidth and latency between pairs of processors are obtained from dynamic NWS forecasts. To provide an estimate of the number of messages transferred ($M$) and the megabytes transferred ($MB$) we examined post-execution program performance reports provided by Legion. For a variety of problem sizes and resource set sizes the number of megabytes transferred varied by less than 5% so we used an average value for all runs. Data transfer does not significantly vary with problem size because the problem size affects only the height of the grid while the decomposition is performed horizontally. Data transfer costs also do not vary with number of processors because each processor must communicate with only its neighbors, regardless of the total number of processors. Although the number of messages transferred varied more
significantly from run-to-run we also used an average value for this variable. This approximation did not adversely affect our scheduling ability in the environments we tested; in cases where communication costs are more severe a model could be developed to approximate the expected number of messages transferred.

The overhead factor $\text{Over}_1$ is included in the performance model to capture application and system behavior that cannot be accounted for by a simple communication/computation model. For example, a processor will likely spend time synchronizing with other processors, waiting for neighbor processors for data communication, and waiting for system delays. System overheads are associated with specifics of the hardware and Legion infrastructure such as the time required to resolve the physical location of a data object needed by the application. The overhead for PMHD3D can be estimated by:

$$\text{Over}_1 = 16 - 1.5 \times \text{probSize}/1000 + 0.094P^2$$

where $P$ is the number of processors involved in the computation and $\text{probSize}$ is the height of the PMHD3D mesh.

$\text{Over}_1$ was estimated empirically using data from 106 individual application executions with problem sizes varying from 1000 to 6000 and with resource set sizes varying between 4 and 26. To determine the effect of the number of processors on overhead runs, runs were grouped by problem size and the corresponding execution times plotted against number of processors. For each set of runs performed with the same problem size, a quadratic fit was performed on the difference between the actual execution time and the predicted execution time (without the overhead factor). The quadratic factor varied between 0.090 and 0.096 with a mean of 0.094 (standard deviation of 0.0022). To determine the effect of problem size on overhead we used the same runs but did a linear datafit on the predicted/actual execution time difference with problem size.

### 3.2. Resource Selection

Resource selection is the process of selecting a set of target resources (processors in this case) that will be performance-efficient. Finding the optimal set of resources requires comparing all possible schedules on all possible subsets of the resource pool - clearly an inefficient process as the resource pool becomes large. Instead, we create several ordered lists of resources by employing a heuristic to sort candidate resources in terms of several definitions of resource desirability. Resource desirability is based on how resource characteristics such as computational speed and network connectivity will affect the performance of PMHD3D.

The resource selection process begins by querying Legion to discover the available set of resources. Effective evaluation of the desirability of each resource requires application-specific performance information as well as dynamic resource performance information. As of this writing, Legion collection objects report available resources and their static configurations but do not provide up-to-date dynamic information on availability, load, or connectivity. Accordingly, the list of available resources reported by Legion is used to query NWS for dynamic forecasts of resource availability, CPU load, and free memory for each host and of latency and bandwidth between all pairs of hosts. To obtain the computational cost per unit of the PMHD3D grid on each type of resource we used the benchmarking method described in Section 3.1.

Once the available resource lists and the dynamic system characteristics are collected, the list can be ordered in terms of desirability. We use three definitions of desirability of a resource: desirability based on connectivity, desirability based on computational power, and desirability based equally on the two characteristics. Connectivity is approximated by computing the latency and bandwidth between the resource in question and all other resources in the resource pool: as a metric we calculate the amount of time (seconds) it would take for the resource in question to exchange a packet of size 1 byte to and from every other host. Computational power is measured by the time (seconds) it would take the host to compute 1 point for 1 iteration based on the NWS predictions and the benchmarks we discussed earlier. The balanced strategy orders the resources based on an average of computational power and connectivity.

The resource set is sorted into 3 resource lists using the 3 notions of resource desirability. We then create subsets of the lists by selecting the $n$ most desirable hosts from each list where $n = \max P$ and $n$ is even. We select multiple subsets from each list because it is often impossible to know the optimal number of hosts a priori. Once the subsets have been created the resulting group of proposed resource sets are passed on to the schedule creation step described in the next section. Although the approach described here is not guaranteed to find the optimal resource set, the methodology provides a scalable and performance-efficient approach to resource selection.

### 3.3. Schedule Creation and Selection

For each of the proposed resource sets, a schedule is developed. Essentially, schedule development on a given resource set for PMHD3D reduces to finding a work allocation that provides good time balancing. As in Section 3.1 work allocation is represented by $z_i$ and is the height of the slab given to processor $P_i$.

One of the most important characteristics for any solution to this problem is time balancing: all processors should finish at the same time. Using the notation from Section 3.1,
\( T_i = T_{i+1}, \ i \in \{1 \ldots (n - 1)\} \) and, since all of the work must be allocated, we also have \( \sum_i x_i = \text{probSize} \).

Taken together we have \( n \) equations in \( n \) unknowns and the problem can be solved with a basic linear solver. This approach was successful for the Jacobi-2D AppLeS [2] but is not powerful enough to incorporate several additional constraints required to develop good schedules for PMHD3D.

One of the important constraints for PMHD3D performance is the amount of memory available for the application. There is a limit to the size of problem that can be placed on a machine because if the computation spills out of memory, performance can drop by two orders of magnitude. To quantify this constraint a benchmark for application memory usage must be obtained by observing memory usage for varying problem sizes on each type of resource. Formally, this constraint becomes:

\[ BM_{\text{mem}_i} \times x_i < Mem_{\text{Avail}_i} \]

where \( Mem_{\text{Avail}_i} \) is the available memory for processor \( i \) (provided by the NWS memory sensor) and \( BM_{\text{mem}_i} \) is the memory benchmark (megabytes/unit) recorded for processor \( i \)'s architecture.

We formalize the work allocation constraints as a Linear Programming problem (from now on simply LP), solvable with the simplex method [6]. In short, LP solves the problem of finding an extreme (maximum or minimum) of a function \( f(x_1, x_2, \ldots, x_n) \) where the unknowns have to satisfy a set of constraints \( g(x_1, x_2, \ldots, x_n) \geq b \) and both the objective function and the constraints are linear. The simplex is a well-known method used to solve LP problems. The simplex formulation requires that constraints are expressed in standard form; that is the constraints must be expressed as equalities and each variable is assigned a non-negativity sign restriction. There is a simple procedure that can be used to transform LP problems into a standard form equivalent.

We modified the time balancing equations to provide some flexibility for the constraints specification: expected execution time for any processor in the computation must fall within a small percentage of the expected total running time. This flexibility is beneficial, especially as additional constraints such as memory limits are incorporated into the problem formulation. The constraints are initially very rigid but can be relaxed in cases where no solution can be found given the initial constraints. The time balancing equations and the application memory requirements form the application constraints on which the simplex has to operate. The simplex formulation also requires specification of an objective function where the goal of the solver is to maximize the objective function while satisfying the simplex constraints.

We use \( \sum_i x_i \) as the objective function and search for a solution where all work is allocated.

For each of the proposed resource sets the simplex is used to create the best schedule possible for that resource set. We use a library [16] which provides a fast and easy to use implementation of the simplex. There are several benefits of using linear programming and the simplex method to create a good schedule:

- Linear programming is well known and commonly used so that fast and reliable algorithms are readily available.
- Once the constraints are formalized as a linear programming problem, adding additional constraints is trivial. For example, the FORTRAN compiler used to compile PMHD3D enforced a limit on the maximum size of arrays, therefore limiting the maximum units of work that could be allocated to any processor. This constraint was easily added to the problem formalization.
- The linear programming problem can be extended to give integer solutions, although the problem then becomes much more difficult. Currently the solver computes real values for work allocation and we redistribute the fractional work portions. In some problems a linear solution may be required for additional accuracy.
- In the case that a solution cannot be found, the simplex method provides important feedback. For this application, the simplex could not find a solution if the constraints were too restrictive. In this case the simplex is reiterated with successively relaxed constraints until a solution can be reached.

Once the proposed schedules are identified, schedule selection is surprisingly simple. The performance model is used to evaluate the expected execution time of each proposed schedule, and the schedule with the lowest estimated execution time is selected and implemented.

4. Results

The PMHD3D AppLeS has been implemented and we present results to investigate the usefulness of the methodology. The goals of these experiments were to:

- Evaluate the accuracy of our performance prediction model.
- Evaluate the ability of the PMHD3D AppLeS to promote application performance in a multi-user Legion environment.

The previous sections stressed the importance of the performance model for effective scheduling. In Section 4.2 we explain in detail results demonstrating the accuracy of the
performance model. In Section 4.3 we present evidence that the scheduling methodology and implementation are effective in practice. Before discussing these results we first outline our experimental design.

4.1. Experimental Design

To evaluate the PMHD3D AppLeS, we conducted experiments on the University of Virginia Centurion Cluster, a large cluster of machines maintained by the Legion team (see [4] for more information on the cluster). The Centurion Cluster is continuously upgraded for new Legion version releases; during the 3-month period of the experiments, we used Legion versions 1.5 through 1.6.1. The cluster itself is composed of 128 Alphas and 128 Dual-Pentium II PCs; 12 fast Ethernet switches and a gigaswitch connect the whole cluster. Although we employed both Alphas and Pentiums during the development and initial testing process, we had multiple difficulties with Alpha Linux kernel instabilities and a faulty network driver which made our data for the Alphas machines unreliable. The results presented here are based only on the 400 MHz Dual Pentium II machines. We didn’t employ the second processor on the Dual Pentium: therefore when we think of host or machine we consider the machines to be uniprocessors. It is worth noting that many users only use one processor per node so that even a computationally intensive user will not affect CPU availability as much as might be expected. However, the two processors on each Dual Pentium machine utilize the same memory, sometimes leading to performance degradation due to overloaded memory systems. Inclusion of memory constraints in the performance model helped the AppLeS scheduler avoid overloaded memory systems.

We restricted our experiments to 34 machines for practical reasons: the dynamic information collected from NWS includes a large amount of data, even for a relatively small cluster. Limiting the resource pool did not impact investigations of application performance or schedule efficiency because, as will become clear, the parallelism available in PMHD3D for the problem sizes studied here is well below the 34 machine limit. As explained in Section 2.5 we used an IBP server running at all times at UCSD, while AppLeS acted as an IBP client retrieving the forecasts. This setup allowed us to obtain updated predictions for a large number of resources in a reasonable amount of time. On average it took less than 4 seconds to retrieve the data, with a minimum of 2.5 seconds and a maximum of 8.5 seconds.

To test the performance of PMHD3D under a variety of conditions, experiments were typically performed with maximum resource set sizes (from now on called resource pool or simply pool) of 4, 6...26 and problem sizes of 1000, 2000...6000. Problem size is the height of the data grid used by PMHD3D. The pool is the maximum number of machines the scheduler is allowed to employ. We test varying pool sizes to simulate conditions under which a user may be limited to a certain number of resources by cost or access considerations. Although our overall resource pool contains 34 machines in total, the maximum pool size we simulate is only 26. This choice was practical: we frequently found unavailable or inaccessible machines in our overall resource pool and so were never able to access all 34 machines at one time. Note also that the scheduler may determine that utilizing the entire pool is not the most performance efficient choice. In this case the pool is larger than the number of target resources.

The experiments presented in Section 4.2 were conducted under unloaded conditions while those presented in Section 4.3 were conducted under loaded conditions. The ambient load present during most of our loaded runs consisted of heavy use of some machines and light use of others. In order to investigate application performance we report performance results based on application execution time. However, there is a cost associated with using AppLeS to develop a schedule. We analyzed 43 runs in detail and the dominant scheduling cost is associated with querying the Legion Collection and the Legion context space. The time required to access NWS and IBP is on average less than 4 seconds. Once the system and performance information has been collected, the AppLeS required on average roughly 1 second to order the resources, create schedules, and select the best schedule.

4.2. Performance Model Validation

The performance model is the basis for determining a good work allocation and, more importantly, provides the basis for selecting a final schedule among those that have been considered. We tested model accuracy for a variety of problem sizes and target resource sets (see Figure 3). For the 62 runs shown in this figure the model accurately predicts execution time within 1.5%, on average. The performance model consistently achieved this level of accuracy for other runs taken under similar conditions. Notice that as the problem size becomes larger, the smallest pool that we test also increases (i.e. the smallest pool for a problem size of 2000 is of size 4 while for a problem size of 6000 it is 12). This experimental setup was required by a limit in the g77 FORTRAN compiler we employed: no more than 507 work units could be allocated to any one processor during the computation.

Figure 3 demonstrates the importance of selecting an appropriate number of target resources for PMHD3D. For example, for a problem size of 1000 the minimal execution time is achieved when the application is run on 10 processors. If fewer processors are used, the amount of work per processor is high and the overall execution time is higher.
Figure 3. Model predictions (dashed lines) and observed execution time (solid lines) for a variety of problem sizes and pool sizes.

Table 1. Number of resources to target for various problem sizes under unloaded conditions. Optimal is the best choice, range indicates close to optimal choices.

<table>
<thead>
<tr>
<th>Size</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
<th>5000</th>
<th>6000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hosts</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Range</td>
<td>8-12</td>
<td>12-14</td>
<td>14-16</td>
<td>14-18</td>
<td>16-18</td>
<td>18-20</td>
</tr>
</tbody>
</table>

If more processors are used, the added communication and system overheads cannot be offset by the advantage of the additional computational power. Significantly, the performance model accurately tracks the knee (i.e. inflection point) in the curve and is thus capable of predicting the correct number of target resources, at least under these conditions. We report the optimal number of target resources for all problem sizes tested in Table 1. As will be obvious in Section 4.3, the optimal number of processors may vary with resource performance and dynamic system conditions as well as with problem size.

Figure 4 demonstrates the scheduling advantage of accurately predicting the correct number of processors to target. In these experiments the PMHD3D AppLeS was allowed to select any number of processors up to the maximum pool size. The PMHD3D AppLeS selects the maximum number of resources for each resource pool up to and including a size of 18. For resource pools of size 20 and larger the optimal number of hosts is 18 and the PMHD3D AppLeS correctly selects only 18 hosts.

Figure 4. PMHD3D AppLeS predicted and actual execution times for a problem size of 5000.

4.3. Performance Results

Once we verified that the performance model is accurate in a predictable environment (i.e. where resources are dedicated), we turned our attention to considering the
performance of the AppLeS in a more dynamic, unpredictable, multi-user environment. We begin by investigating the ability of PMHD3D AppLeS to compare available resources and select desirable hosts (computationally fast, well-connected, or both). To provide a comparison point we test the performance of another available scheduler, namely the default Legion scheduler. We conducted experiments in runs, namely back-to-back PMHD3D executions using the same resource pool and the same problem size but utilizing the PMHD3D AppLeS scheduler first and the default Legion scheduler second.

Figure 5. PMHD3D performance attained with and without the AppLeS scheduler for a problem size of 1000.

In Fig. 5 we show a series of runs comparing the two schedulers for a problem size of 1000. Clearly, the PMHD3D AppLeS provides a performance advantage for all resource set sizes tested. However, it is notable that the two execution time curves follow the same trend only when the resource pool is in the range of 4-12 hosts. When more resources are added to the pool the execution time achieved with the PMHD3D AppLeS remains constant while the default Legion scheduler execution time diverges. The default Legion scheduler allocates all available resources, a less than optimal strategy for PMHD3D. In Table 2 we report the typical number of processors selected by AppLeS for different problem sizes and resource set sizes.

For pool sizes of 4 - 12 performance achieved via the PMHD3D AppLeS is consistently 20 - 25 seconds lower than that achieved via the default scheduler. In this range of pool sizes, the PMHD3D AppLeS selects the maximum number of hosts available and so uses the same number of resources as the default Legion scheduler. The performance advantage is achieved by selecting "desirable" resources, i.e. resources that are computationally fast and/or well-connected. Figure 6 illustrates the load of all available machines just before scheduling occurred for the 18-processor run shown in Figure 5. Clearly, the PMHD3D AppLeS selects lightly loaded hosts (i.e. those hosts with high availability) while the default scheduler selects several loaded hosts. It is the load on these selected machines that causes a performance disadvantage for the default scheduler. In a more heterogeneous network environment the connectivity of the hosts would also play an important role in host selection and resulting performance.

We obtained 83 runs comparing the default Legion scheduler to the PMHD3D AppLeS for a variety of problem sizes (1000-6000) and pool sizes (4-26). Figure 7 shows a histogram of the percent improvement the PMHD3D AppLeS achieved over the default Legion scheduler for the 83 runs (the average improvement was 30%).

Note that in a few runs there was little or no advantage to using the PMHD3D AppLeS. In these cases the processors were essentially idle and the pool size was below the optimal number so that the schedulers selected the same number of processors. In one run the PMHD3D AppLeS-determined schedule was considerably slower than that determined by the default Legion scheduler. In this case the scheduler created a schedule based on incorrect system information: NWS forecasts of CPU availability were unable to predict a sudden change in load on several machines and the resulting schedule was poorly load balanced.

The Legion default scheduler was designed to provide general scheduling services, not the specialized services we include in the PMHD3D AppLeS. It is therefore not surprising that the AppLeS is better able to promote application performance. In fact, the PMHD3D AppLeS could be developed as a Legion object for scheduling regular, iterative, data-parallel computations, and this is a focus of future work. Using the PMHD3D AppLeS and the Legion default scheduling strategy as extremes, we wanted to explore a third alternative for scheduling -- that of what a "smart user" might do: In a typical user scenario for a cluster of machines a user will have access to a large number of machines and will typically do a back-of-the-envelope static

### Table 2. Hosts chosen by PMHD3D AppLeS. The Legion default scheduler always selects the maximum number of hosts.

<table>
<thead>
<tr>
<th>Max Hosts</th>
<th>1000</th>
<th>2000</th>
<th>4000</th>
<th>5000</th>
<th>6000</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
calculation to determine an appropriate number of target resources given the granularity of the application. Although a user may correctly determine the number of hosts to target, accurate information on resource load and availability will be difficult or impossible to obtain and interpret prior to or at compile-time.

To simulate this user scenario, we developed a third scheduling method called the smart user. The smart user selects an appropriate number of hosts but does not select hosts based on desirability. Experiments were performed for problem sizes ranging from 1000 to 6000 with a pool size of 26 hosts. Figure 8 shows the performance obtained by the PMHD3D AppLeS, the default Legion scheduler, and that obtained by the smart user. In these experiments, the PMHD3D AppLeS provides a significant performance advantage over both alternatives.

5. Related Work

The PMHD3D AppLeS is an adaptation and extension of previous work targeting the structurally similar Jacobi-2D application ([2],[3]). Jacobi-2D is a data-parallel, stencil-based iterative code, as is PMHD3D. Both applications allow non-uniform work distribution, however Jacobi-2D employs strip decomposition (using strip widths) for its 2-dimensional grid while PMHD3D employs slab decomposition (using slab height) for its 3-dimensional grid. While the applications are structurally similar, PMHD3D required tighter constraints on memory availability and a more complex performance model. Additionally, PMHD3D was targeted for a much larger resource set (34 machines vs. 8). The availability of a larger resource pool for this work motivated the introduction of the quadratic overhead term in the PMHD3D performance model. Previous AppLeS work has not included the additional overhead of using extra machines in scheduling decisions.

As part of our previous work, we developed an AppLeS for Complib and the Mentat distributed programming environment. Complib implements a genetic sequencing algorithm for libraries of sequences. It is particularly difficult to schedule because of its highly data dependent execution profile. The implementation of Complib we chose was for Mentat [8] which is an early prototype of the Legion Grid software infrastructure. By combining a fixed initial distribution strategy (based on a combination of application characteristics and NWS forecasts) with a shared work-queue distribution strategy, the Complib AppLeS was able to achieve large performance improvements in different Grid settings [20]. In addition to AppLeS for Legion/Mentat applications, we have developed AppLeS for a variety of Grid infrastructures and applications [19, 21, 7].

In [10], the authors describe a scheduler targeting data parallel "stencil" applications that use the Mentat programming system. They specifically examine Gaussian elimination using a master/slave work-distribution methodology. While it is difficult to compare the performance of each system, their approach differs from AppLeS in that it requires more extensive modification of the application and it does not incorporate dynamic information.
6. New Directions

An ultimate goal is to offer the PMHD3D AppLeS agent within the Legion framework as a default scheduler for iterative, regular, stencil-based distributed applications. In particular, the scheduler’s performance model is flexible enough to incorporate the requirements and constraints of other stencil applications and the characteristics of other platforms. To use this model for other appropriate applications, good predictions of megabytes transferred, number of messages initiated, overhead factor, benchmarks for program CPU and memory utilization over the different target architectures, as well as access to dynamic system information from NWS or a similar system would be required. Once obtained, these characteristics are used as inputs to the model without changing the model structure.

Portability and heterogeneity are also important. The AppLeS itself is written in C and Perl and has been compiled successfully and executed on various architectures and systems (Pentium, Alpha, Linux and Solaris). Initial results indicate that the scheduler can be used effectively on different target environments without changes to the structure of the performance model. For example, we used mpich on a local cluster for initial development and debugging. The schedule worked well with only the previously described changes in model input parameters.

Acknowledgements

The authors would like to express their gratitude to the reviewers for their comments and suggestions. The insight and focus provided by their comments improved the paper greatly. We thank the NWS team and Jim Hayes in particular for sharing their NWS expertise with us. We also thank the Legion team and Norman Francis Beekwilder in particular for sharing their Legion expertise with us.

References


Holly Dall is currently a M.S. student in the Department of Computer Science and Engineering at the University of California San Diego. She received a B.S. in Physics and a B.S. in Oceanography from the University of Washington in 1996. Her current research interests focus on achieving application performance in Computational Grid environments.

Graziano Obertelli is currently an Analyst Programmer in the Department of Computer Science and Engineering
at the University of California, San Diego. He received his Laurea in Computer Science at Università degli Studi, Milano.

Francine Berman is a Professor of Computer Science and Engineering at the University of California, San Diego. She is also a Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, her M.S. and Ph.D. from the University of Washington.

Rich Wolski is an Assistant Professor in the Department of Computer Science at the University Tennessee, Knoxville and a partner in the National Partnership for Advanced Computational Infrastructure. His research interests include parallel and distributed computing, on-line performance analysis techniques and software, compiler runtime system, and dynamic scheduling. He received his B.S. from the California Polytechnic University, San Luis Obispo and his M.S. and Ph.D. from the University of California at Davis/Livermore Campus.

Andrew S. Grimshaw is an Associate Professor of Computer Science and director of the Institute of Parallel Computation at the University of Virginia. His research interests include high-performance parallel computing, heterogeneous parallel computing, compilers for parallel systems, operating systems, and high-performance parallel I/O. He is the chief designer and architect of Mentat and Legion. Grimshaw received his M.S. and Ph.D. from the University of Illinois at Urbana-Champaign in 1986 and 1988 respectively.
Accountability and Control of Process Creation in the Legion Metasystem*

Marty Humphrey
humphrey@cs.virginia.edu

Frederick Knabe
knabe@virginia.edu

Adam Ferrari
ferrari@virginia.edu

Andrew Grimshaw
grimshaw@cs.virginia.edu

Abstract

A metacomputing environment, or metasystem, is a collection of geographically separated resources (people, computers, devices, databases) connected by one or more high-speed networks. The distinguishing feature of a metasystem is middleware that facilitates viewing the collection of resources as a single virtual machine. The traditional requirements of security mechanisms and policies in a single physical host is exacerbated in a metasystem, as the physical resources of the metasystem exist in multiple administrative domains, each with different local security requirements. This paper illustrates how the Legion metasystem both accommodates and augments local security policies specifically with regard to process creation. For example, Legion configurations for local sites with different access control mechanisms such as standard UNIX mechanisms and Kerberos are compared. Through analysis of these configurations, the inherent security trade-offs in each design are derived. These results have practical importance to sites considering any future inclusion of local resources in a global virtual computer.

1 Introduction

The emerging widespread introduction and use of gigabit wide area and local area networks have the potential to transform the way people compute, and more importantly the way they interact and collaborate with one another. The increase in bandwidth will enable the construction of wide area virtual computers, or metasystems (Figure 1). However, in the face of the onrush of hardware, the community has tried to stretch an existing paradigm—interacting autonomous hosts—into a regime for which it was not designed. The challenge is to provide a solid, integrated, conceptual model on which to build applications that unleash the potential of so many diverse resources. The foundation must at least hide the underlying physical infrastructure from users and from the vast majority of programmers. It must support access-, location-, and fault-transparency, enable interoperability of components, support construction of larger integrated components using existing components, scale to millions of autonomous hosts, and provide a secure environment for both resource owners and users.

Security services provided by the metacomputer software infrastructure are not unlike traditional services required by uniprocessor operating systems. Users must authenticate themselves

---

*This work was funded by DARPA contract N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, and DOE D459000-16-3C.
to the "system". Users must not be allowed to access arbitrary resources. Logging mechanisms are used to hold users accountable and to track intruders.

However, a solution to the metacomputer security problem requires significant additions over the uniprocessor security solution [3], particularly because the machines that comprise the metacomputer may exist in different administrative domains. In a metacomputing environment, the security problem can be divided into two main concerns:

1. Protecting the metacomputer's high-level resources, services, and users from each other and from possibly corrupted underlying resources (within the metasystems layer in Figure 1)

2. Preserving the security policies of the underlying resources that form the foundation of the metacomputer and minimizing their vulnerability to attacks from the metacomputer level (the vertical arrows in Figure 1)

For example, restricting who is able to configure a metacomputer-wide scheduling service would fall in the first category—it's solution requires metacomputer-specific definitions of identity, authorization, and access control. Meanwhile, enforcing a policy that permits only those metacomputer users who have local accounts to run jobs on a given host falls in the second category, and it might require a means to map between local identities and metacomputer identities.

The security infrastructure of the Legion metasystem was created to solve these two problems. Legion [5, 6] is an object-based metasystem with a goal of supporting millions of objects spread across thousands of machines. Legion executes as middleware—below the end-user applications but above the operating system (thus, the Legion infrastructure itself is not "privileged code"). Legion provides process management, inter-process communication, persistent storage, a single unified file system, and security services. Legion supports PVM, MPI, C, Fortran, a parallel C++, Java and the CORBA IDL. Legion can select resources for use by applications and securely coordinate large-scale application execution, eliminating the need for the end-user to explicitly log on to each machine, FTP files, create processes, create temporary files, etc. A project of Legion's size and scope is faced with numerous technical challenges, all related to managing the potentially huge number of underlying heterogeneous hardware and software platforms.

In the Legion system, host objects represent processing resources. When a Legion object is instantiated, it is a Host Object that actually creates a process to contain the newly activated object. The Host Object thus controls access to its processing resource and can enforce local policies, e.g.,
ensuring that a user does not consume more processing time than allotted. *Vault objects* in Legion represent stable storage available within the system for containing Object Persistent Representations (OPRs). Just as Host Objects are the managers of active Legion objects, Vault Objects are the managers of inert Legion objects. For example, Vaults are the point of access control to storage resources, and can enforce policies such as file system allocations.

In a metasystem, arbitrary users must not be allowed to allocate resources on arbitrary machines. In contrast, allowing this is analogous to a local sysadmin granting an account to any person without an evaluation of what the person might do on the machine if an account were granted. Sysadmins rarely allow this in order to protect their integrity of the data on their machines and to ensure that resources will be available to "legitimate" users. Similarly, local sites will choose to participate in a metasystem, but contingent on assurances that their local security policies will not be violated.

The specific requirement of Legion that this paper addresses is that Host Objects must only create processes for objects for users that have been authenticated and are authorized to use the underlying resources. Local system administrators will allow their systems to participate in a Legion network (i.e., execute Legion-related processes and store Legion-related objects) only if these processes and files are created according to local policy. For example, a site's security requirements could range from not restrictive (allowing any Legion user to create processes, even if the particular Legion user does not have an account on the computers at the site) to very restrictive (allowing only those users who have personal accounts on the machines on their site, and only after Kerberos authentication). Note that in all cases, only users who have authenticated themselves to Legion are allowed to create processes. Legion must be made to accommodate local policy, over which Legion designers have little or no influence. That is, it is not the intent of the Legion developers to mandate underlying security policy; the challenge in Legion is to adhere to and support local security policies while creating an easy-to-use and secure environment for end users.

The contribution of this paper is that it reveals how this metacomputing problem manifests itself and is solved in the context of Legion. It presents the Legion solution to three site-specific requirements to authentication and authorization for the purposes of process creation. In the first scenario, the site requires standard UNIX access control (similar to most UNIX installations at universities). In the next two scenarios, the sites require Kerberos authentication but differ in their specific security policies regarding process creation after Kerberos authentication. The "level of security" as compared to ease of use of each scenario is discussed in terms of the likelihood and ramifications of the compromise of user credentials. The importance of this work is that it describes the solution to real-world security problems that exist as a result of users who increasingly create programs that require transparent, secure access to multiple, geographically dispersed resources. Sites are eager to deploy the Legion metasystem software, but only after receiving assurances that their local security policies are satisfied. Without solutions such as these described in this paper, Legion cannot be deployed on a large-scale basis.

The organization of this paper is as follows. Section 2 presents the Legion mechanisms for security that are independent of any underlying host system. This section covers identity, authentication, authorization, and accountability. Section 3 describes how processes are started and controlled, and how files are written, on a system that employs standard UNIX access control. Section 4 describes two configurations in which Legion integrates with a Kerberized host. Section 5 describes projects that are related to this work. Section 6 contains the conclusions.
2 Security Provided by Legion Mechanisms

The security model for Legion differs significantly from that of conventional systems. A Legion "system" is really a federation of resources from multiple administrative domains, each with its own separately evaluated and enforced security policies. As such, there is no central kernel or trusted code base that can monitor and control all interactions between users and resources. Nor is there the concept of a superuser—no one person or entity controls all of the resources in a Legion system.

Legion programs and objects run on top of host operating systems, in user space. They are thus subject to the policies and administrative control of the local OS. Not surprisingly, the Legion objects running on a particular host must trust that host. This trust does not necessarily extend to objects running elsewhere, however. A critical aspect of Legion security is that the security of the overall Legion system cannot rely on every host being trustworthy. A large Legion system will span multiple trust domains, and even within one trust domain, some of the hosts may be compromised or may even be malicious. For example, two organizations might use Legion to share certain resources in specifically constrained ways. Such sharing would clearly not be acceptable if one organization could subvert the other's objects through its ownership of some part of a Legion system.

The purpose of this section is to provide a discussion of Legion security mechanisms that independent of the underlying security mechanisms and policies of the host systems. The material in this section is presented primarily to establish the context for the remaining sections. For more details regarding this mechanisms, see [3].

2.1 Identity

Identity is fundamental to higher-level security services such as access control. Every Legion object is identified by a unique, multi-field, location-independent Legion Object Identifier, or LOID. One of the LOID fields contains security information including an RSA public key. By including the public key in an object's LOID, it is easy for other objects to encrypt communications to that object or to verify messages signed by it. Objects can just extract the key from the LOID, rather than looking it up in some separate database. By making the key an integral part of an object's name, we eliminate some kinds of public key tampering. An attacker cannot substitute a new key in a known object's id, because if any part of the LOID is altered, including the key, a new LOID is created that will not be recognized by certain core system objects. One drawback in this design is that there is no mechanism for revoking an object's key and issuing a new one, as this step implies a complete change of the object's name.

The default security field of a LOID is more than just an RSA public key. It is actually an X.509 certificate that contains the key. In general, an X.509 certificate pairs a public key with a person's name, organization, identification of the public key algorithm, and other information. A certificate may be signed by a certification authority (CA) that vouches for the association of the key with the identifying information. To cover the case where a recipient doesn't recognize the CA, the CA's own certificate can be chained onto the certificate, allowing the CA's CA to be the basis of authority. The user's X.509 certificate is propagated with requests and method calls made directly or indirectly on behalf of the user. The information in the certificate is used when making entries to access logs.

In a distributed object system such as Legion, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf. One way in which to allow intermediate objects to request services on behalf of an originating object is to give the intermediate objects
a copy of the private key of the originating object, thus proving authentication. This approach is clearly insecure, as intermediate objects could then maliciously originate operations on false behalf of the originating object. An alternative approach is to have intermediate objects call back to the user or his trusted proxy when they receive access requests in the user’s name. This step puts control back in the user’s hands. There are several drawbacks to this approach, though. First, the fine-grain control afforded by authorization callbacks may be mostly illusory. It can be very difficult to craft policies for a user proxy (or even the real user himself!) that are much more than “grant all requests”—too much contextual and semantic information is generally missing from the request. Beyond this barrier, callbacks are expensive and do not scale well. In Legion, after all, every object represents a resource of some type, and a callback on every method call would be a crippling performance hit.

The intermediate solution between these approaches is to issue credentials to objects. A credential is a list of rights granted by the credential’s maker, presumably the user. They can be passed through call chains. When an object requests a resource, it presents the credential to gain access. The resource checks the rights in the credential and who the maker is, and uses that information in deciding to grant access. There are two main types of credentials in Legion: delegated credentials and bearer credentials. A delegated credential specifies exactly who is granted the listed rights, whereas simple possession of a bearer credential grants the rights listed within it. A Legion credential specifies the period the credential is valid, who is allowed to use the credential, and the rights—which methods may be called on which specific objects or class of objects. The credential also includes the identity of its maker, who digitally signs the complete credential.

2.2 Access Control

Each Legion object is responsible for enforcing its own access control policy. The general model for access control is that each method call received at an object passes through a MayI layer before being serviced (see Figure 2). MayI decides whether to grant access according to whatever policy it implements. If access is denied, the object will respond with an appropriate security exception, which the caller can handle any way it sees fit.

MayI can be implemented in multiple ways. The trivial MayI layer could just allow all access. Most objects, however, use the Legion-provided default MayI implementation, which essentially defines, on a per-method basis, an allow list and a deny list. The entries in the lists are the LOIDs of callers that are granted or denied the right to call the particular method. Default allow and deny lists can be specified to cover methods that don’t have their own entries.

When a method call is received, the credentials it carries are checked by MayI and compared against the access control lists. For example, in the case of a delegated credential, the caller must have included proof of his identity in the call so that MayI can confirm that the credential applies. Multiple credentials can be carried in a call; checking continues until one provides access.
2.3 Communication between Legion Objects

A method call from one Legion object to another can consist of multiple Legion messages. Because Legion supports dataflow-based method invocation, the various arguments of a method call may flow into the target as messages from several different objects. A message may be sent with no security, in *private mode*, or in *protected mode*. In both private and protected modes, certain key elements of a message (e.g., any contained credentials) are encrypted. In private mode the body of the message is encrypted, whereas in protected mode only a digest is generated to provide an integrity guarantee. Unless private mode is already on, protected mode is selected automatically if a message contains credentials. The mode selected for use by an originating object is applied for all messages indirectly generated as a result of the originating message. For example, a user can select private mode when calling an object. The calls that the object makes on behalf of the user will also use private mode, and so on down the line. Currently, encryption is based on the RSA toolkit (RSAREF 2.0).

In addition to protecting credentials, both protected mode and private mode encrypt a *computation tag* contained in every Legion message, a random number token that is generated for each method call. All the messages that make up a given method call contain the same computation tag. The tag is used to assemble incoming messages from multiple objects into a single method call and to identify the return value for a call made earlier. If an attacker knows the computation tag for a method call, he can forge complete messages containing arguments or return values, even without holding any credentials. The computation tag is treated as a shared secret, and is never transmitted in the clear unless "no security" mode is selected.

The security layer does not provide mutual authentication. The sender can be assured of the identity of the recipient, because only the desired recipient can read the encrypted parts of the message. The recipient usually doesn't care who the actual sender is; its decisions are based solely on the credentials that arrived in the message.

3 Legion Integration with Standard UNIX Access Control

The previous section described object interactions at the logical level of the metasystem in Figure 1—specifically, how one Legion object can authenticate with another Legion object and exchange secure communication. However, Legion objects must physically exist on a host that is part of the metasystem. This section describes how an object is instantiated on a host that requires only standard UNIX access control.

Our general strategy for isolating Legion objects from one another is to run them in separate accounts on the host system. The accounts that can be used for this purpose fall into two categories:

- For those Legion users who happen to have accounts on the system, objects can run on their normal user accounts.

- For other users, there is a pool of generic accounts that are assigned for Legion use.

The generic accounts usually have minimal permissions. The local Host Object and Vault Object also have their own accounts.

Object creation requests arrive at the Host Object as normal method invocations, and can thus be controlled using the standard Legion access control mechanism for methods. For each request, the host checks the credentials against the user LOIDs and groups that are allowed to create objects on it. If everything is acceptable, it next selects an account for the new object to run in; depending
on the credentials in the creation request and its local configuration, it may choose a local user account or one of the generic accounts. The accounts are subject to scheduling and resource control just like CPU time, memory usage, and so on; an object’s lease on an account, especially a generic account, is usually limited.

All Legion objects are associated with some persistent storage, typically in the form of a directory in the local file system managed by the Object Vault. Before starting an actual process for the new object in the allocated account, the host needs to change the ownership of the object’s directory from the vault user-id to the newly allocated user-id. The location of the directory that will contain the new object’s persistent state is passed to the host as part of the activation request (this location was obtained through a method on the local vault performed by the object’s creator, likely its class). Ownership of this directory must be changed to both protect the object’s state from access by other objects (which will run under different user-ids), and to make the state accessible to the new object.

Finally, the host needs to spawn the actual process that will execute the object on the appropriate account. To carry out this step, and to change ownership of the object’s persistent state, the host requires access to some privileged operations. However, the host does not execute with root permissions. Access to these required privileged operations is encapsulated in a process control daemon (PCD) that executes on the host, providing services to the Host Object in a controlled fashion. The PCD is a small, easily vetted program that runs with root permissions. It is configured only to allow access by the host account. Two of its key functions are to permit changing directory ownership and to create new processes on a designated account. The PCD limits the accounts for which this can be done to a set configured by the local system administrator. The set includes the generic Legion accounts and potentially the accounts of local Legion users.

A simplified example of this operation is shown in Figure 3. In this example, Object 12, which is executing on Fred’s account on some machine in the metasystem, and has access to Fred’s Legion credentials, wants to create an object on the host Elmer.virginia.edu. To do this, Object 12 asks the Host Object executing on Elmer to start a process for the new object. First, the Host Object confirms that Object 12 is acting on Fred’s behalf by looking at the credentials contained in the request. Then, the Host Object maps a request by Fred to a generic account (Legion-generic-1) on Elmer.virginia.edu that has been established at the time that Legion was installed on Elmer.virginia.edu. Finally, the Host Object asks the PCD to spawn a new process as Legion-generic-1 for the new object.
As the PCD starts the object running, the host logs an audit trail using the X.509 information for the user whose credentials accompanied the request. The audit trail provides essential information if the new object misuses local resources. If the object has exceeded its use of local resources, the host can request that the PCD kill it directly. When an object loses or relinquishes its use of an account, the Host Object uses the PCD to change the ownership of its persistent state back to the Vault Object. If the object is reactivated later on a different account, ownership of the state can be changed to the appropriate user-id. After an account is reclaimed, the PCD terminates all processes running on it and generally cleans it up.

Security Analysis. In accepting this approach, a sysadmin at a local site is trusting the Legion software to legitimately map Legion identities to local accounts (if the PCD is configured to map to non-generic local accounts). If the Legion credentials for a particular user are stolen, the risk to the system is less when configured for generic accounts than with non-generic accounts (by their nature, when a user is finished with a generic account, no persistent state remains). Clearly, a negative aspect of this approach is that a site must install the PCD as privileged code, creating a potential point of attack for intruders. However, this code has been vetted by numerous experts, increasing the confidence on the part of local site regarding the safety of this code. Of course, if the Legion Host Object account ("Legion-Account" in Figure 3) is cracked, the intruder can create processes under the accounts of any local Legion users.

Overall. The PCD-based implementation is sufficient for many local system administrators. Legion authentication is used to determine who gains access to local resources, and the resources made available are also constrained to those usable from a limited set of accounts. Detailed logging provides accountability. The safety of credentials is a chief design goal in the security architecture and mechanisms of Legion. An alternative, simple approach is to have all Legion objects execute under the "Legion-Account" account. In general, we have found that sysadmins do not like this approach because of limited accountability—as far as they see, only one account "does anything" with regard to Legion. We (the Legion designers) do not advocate this approach, because it does not provide the necessary isolation, as all files and processes are owned by one oner id (meaning that one Legion user can use UNIX mechanisms to destroy or subvert another Legion user on the same host).

4 Legion Integration with a Kerberized Host

Increased security concerns have caused many sites to switch from standard UNIX access control to the use of Kerberos [9]. Kerberos is a trusted third-party authentication, in which users and services register their keys. In this paper, familiarity with the basic Kerberos protocols are assumed.

It is important to understand that the "Kerberized Host" in this section refers to a host that is executing the MIT source code distribution [7]. This paper does not discuss efforts to integrate Kerberos directly with public key cryptography [10], because this paper focuses on integration with widely-deployed Kerberos systems. Similarly, the use of Proxiable tickets in Kerberos is not discussed, because their usage is not widely supported. While its support is not directly discussed, much of the discussion is applicable to AFS.

In this section, assume that a simple Legion metacomputer is being constructed. There are only two machines involved:
Khost The Kerberized Host machine

NKhost A machine that does not require Kerberos authentication (instead, it uses only password-based authentication)

Additionally, it is useful to define the following entities:

L creds credentials that are necessary to function in the Legion virtual computer

K creds Kerberos credentials; obtained via Kerberos kinit either explicitly or implicitly

admin Legion user with some administrative duties; “admin” does not have an account on either Khost or NKhost; has L creds; may have K creds (if admin logged into Legion virtual machine via some account on NKhost, then does not have K creds; if via Khost, it has K creds, though not as “admin”, because “admin” does not exist as a Kerberos principal on Khost)

Legion-Khost Kerberos principal with account on Khost solely to execute processes related to Legion virtual computer; has K creds; does not have L creds (there is no Legion user named “Legion-Khost”)

Alice-KL has an account on Khost and wishes to participate in Legion virtual machine executing on Khost; has K creds; has L creds; has an account on NKhost

Bob-L has an “account” on Legion virtual machine but no account on Khost; has L creds; does not have K creds; has a UNIX account on NKhost

4.1 Kerberos Background: .k5login and .k5users

In Kerberos, there is the capability to allow one principal to grant access to another principal (after the other principal has authenticated). The file .k5login allows one user to unconditionally grant another user the ability to spawn processes as the first user. For example, if “bob/.k5login contained “jim”, jim could “ksu bob” and have a running shell whereas new processes are tagged as being owned by bob. Note that in this case, jim does not acquire bob’s credentials; rather, in this case, jim, executing as UID bob, has a copy of the Kerberos credentials jim had immediately prior to executing ksu. .k5users is more restrictive than .k5login; .k5users lets bob allow jim to execute only certain binaries as bob, for instance “/bin/ls”. The entry in “bob/.k5users in this case is “jim /bin/ls”. In this case, jim cannot execute a shell as bob; jim could only “ksu bob -e /bin/ls ”.

4.2 Kerberos Solution #1: k5login

A simple solution to allow a user such as Alice-KL with an account on Khost to access the Legion virtual machine is to add “Legion-Khost” to her .k5login file. This approach allows Legion-Khost to execute “binary1” as Alice-KL by invoking “ksu Alice-KL -e full-path-to-binary1”. For Alice-KL to start a process on Khost, essentially the same steps as in Figure 3 are taken. The difference in this situation is that the PCD does not exist, as the Host Object can create processes directly as Alice-KL via invocations of ksu. In this configuration, Legion mechanisms will ensure that Bob-L will not be able to start new processes on Khost, but Bob-L will be able to use services (processes) of Alice-KL, but only if Alice-KL has configured Legion authorization mechanisms to let Bob-L.
Security Analysis. The analysis consists of a number of cases:

If the Legion-Khost account is compromised (i.e., an attacker obtains Legion-Khost Kerberos credential cache, or breaks into the Host Object and, for example, causes the Host Object to execute a binary that allows the credential cache to be read), then all Legion users on Khost, even if they have never used Legion, have been compromised. The attacker cannot get their K-creds, but can start processes on their accounts. There is a variation of this approach, beyond the scope of this paper, that uses .k5users instead of .k5login in order to reduce the scope of attack is the Legion-Khost is compromised. However, this approach is substantially more complicated to implement and analyze.

If Alice-KL’s K-creds are stolen (i.e., either by some activity irrespective of Legion, or if Alice-KL gives her K-creds to a Legion “con-artist” object), then only Alice-KL’s account is compromised. In this case, an attacker can replace legitimate Alice-KL service with corrupted service, thus tricking Bob-L if Bob-L wants to use the service.

If Alice-KL’s L-creds are stolen Alice-KL’s account is compromised because the Host Object can be asked to start jobs on her account.

If admin’s L-creds are stolen an attacker can effectively shut down the Host Object but not break security. Note: admin has no special privileges with regard to the host object, beyond being able to change the ACLs on the Host Object.

Overall. The fact that Alice-KL can start a process on the Khost without directly obtaining and presenting K-creds is both positive and negative: The use of the Legion virtual machine by Alice-KL is easier and perhaps more secure because she does not need to directly acquire Kerberos credentials. A potential problem is that the Legion-Khost has unlimited access to the Khost account of Alice-KL. For this reason, Alice-KL and/or the sysadmins of Khost might require that Alice-KL get a separate account on Khost for use with the Legion virtual machine. For many installations, this approach is sufficient, satisfying the requirement of Kerberos authentication before use of the physical resources (the authentication in this case is by the Legion-Khost principal). A sense of security is provided by the Legion mechanisms based on L-creds that have timeouts, recovery mechanisms, and potentially very specific scope and privilege. It is also very easy to implement.

4.3 Kerberos Solution #2: KProxy Object

In general, a problem of the k5login approach is that a user must grant unlimited access to her account by the Legion-Khost principal. A second problem is that a user such as Alice-KL (or more precisely a person impersonating Alice-KL) does not have to authenticate to the KDC of the Khost Kerberos realm in order to use the physical resources. A second approach eliminates these problems, but at the cost of simplicity.

The essential component of the design is a Legion KProxy Object for each user. This KProxy Object securely holds the Legion user’s Kerberos credentials. The KProxy Object for user Fred executes under Fred’s uid on a machine upon which Fred has an account. Whenever a Host Object anywhere in the metasystem wants to create an object on Fred’s behalf on its associated physical machine, the Host Object performs a call back to the KProxy Object for Fred to obtain a valid ticket for that particular host. Fred’s KProxy Object will only issue Fred’s Kerberos credentials if Fred’s valid Legion credentials are presented in the request (more generally, the access control

---

1The attacker can get their K-creds if the user is logged in.
mechanisms of Fred’s KProxy Object can be configured to issue Fred’s Kerberos credentials to any object that presents valid credentials on behalf of any user with whom Fred has previous established a trust relationship. The Host Object creates the new object via a call to ksu, without requiring the use of the .k5login file.

Note that all communication of secret information is either done via Kerberos mechanisms (DES) or Legion mechanisms (RSAREF). Cross-realm authentication is immediately and transparently supported in this design: kinit only has to be performed once for each group of Kerberos realms that support cross-realm authentication with each other. The Legion KProxy object will automatically obtain TGTs for the other realms based on the existence of a valid TGT for a given host.

The steps that Alice-KL must take in order to create her KProxy object are shown in Figure 4. First, Alice-KL obtains her K-creds from machine NKhost (this can also be performed on Khost, although it doesn’t have to be). By default, these K-creds are tied to the IP address of NKhost (if not—if the KDC supports kinit -A—many aspects of this approach are simplified). On NKhost, Alice-KL then executes legion_create_kproxy, which asks the Host Object on Khost to create an instance of KProxy_class on Khost. As part of this (not shown), Alice-KL interacts with the KDC to obtain a ticket that is usable from Khost. These K-creds are then used by the Host Object in an invocation of ksu to actually create the KProxy object that will hold Alice-KL’s Kcreds. This KProxy object will execute on this machine under Alice-KL’s account. Note that neither Alice-KL’s .k5login nor .k5users contains an entry that directly allows Legion-Khost to execute this ksu; instead, ksu is invoked with an explicit copy of Alice-KL’s K-creds. Now, anytime in the future that the Host Object on Khost wants to create an object on Alice-KL’s behalf, it interacts with this KProxy object to obtain a valid ticket for use in the ksu invocation. Additionally, any Host Object in the realm will interact with Alice-KL’s KProxy object when creating objects.

Why can’t the Legion user obtain K-creds for a particular computation before starting the computation, thus eliminating the need for the KProxy object? Legion is an object-based system in which the necessary functionality to start and coordinate large-scale computation may be spread across hundred or thousands of objects. When a user originates computation, the user has no idea on what machines processes ultimately will be started either directly or indirectly on behalf of this user request. For example, the user may attempt to “run discreet-simulation-1 1000 times”. At this point, the user has no idea (he may not care) which machines are then selected by scheduler objects. Thus, when the user originates computation, he cannot obtain tickets (which are tied to IP addresses) for all of the machines on which processes will be started. Our approach uses a call back to the KProxy object to get the appropriate ticket for each machine.
A second, related note is that the user may not be present (i.e., logged onto a machine in the Kerberos realm) when the Legion software attempts to spawn processes on his behalf. For example, assume that the user starts a long-running activity on one machine, and then logs off. The Legion software might then need to spawn a process on another machine (if, for example, the computation is pipelined, where each element of the pipeline is a sufficiently long computation). Again, we need a call back to something (the KProxy object in our approach) in order to obtain the necessary ticket. One approach might be to somehow get a ticket from the process that already exists that is performing the active part of the pipelined computation (because the credentials cache is present for this process, and presumably it contains a TGT). This is essentially the approach of the KProxy object.

The key limitation that this approach overcomes is that a process can only be started on Khost as user Alice-KL if Alice-KL has previously authenticated herself to the KDC of the Kerberos realm. In addition, at any point, Alice-KL can control the creation of processes under her account by either limiting the lifetime of the ticket held by her KProxy Object, or eliminate her KProxy object completely. However, it is still the case that the Legion user must trust the Legion software to create processes only that she intended.

Security Analysis. The analysis consists of a number of cases:

If the Legion-Khost account is compromised If Alice-KL uses the subverted Khost, the Host Object can use her L-creds to obtain the K-creds from the KProxy. The K-creds can then be misused by the Host Object, for example, to spawn arbitrary processes under Alice-KL's user id (such as "rm "). Note that unlike the k5login approach, where a subverted Khost Legion account can immediately abuse all local Legion user accounts, this approach limits the attacker to misusing the accounts of users currently starting objects on the Khost. This allows intrusion detection as an approach for limiting the damage caused by an attacker.

If Alice-KL's K-creds are stolen The same ramifications as the k5login approach.

If Alice-KL's L-creds are stolen The same ramifications as the k5login approach.

If admin's L-creds are stolen The same ramifications as the k5login approach.

Overall. This approach trades off some additional complexity in terms of the systems structure and some extra effort on the part of users (who no longer get to execute just a single command to login) for an added measure of attack containment. This approach meets the requirement that the user actively authenticate through the Kerberos mechanism before using the local resources (unlike Kerberos k5login approach). This approach incurs added overhead due to the call-backs to the KProxy object; however, these call-backs only occur at the time of object creation, so the impact should not be significant.

5 Related Work

There are several projects being conducted to support inter-operability between a particular security infrastructure and Kerberos, for example supporting a single login for NetWare and Kerberos [1]. This project has similar goals to Legion's integration with Kerberos—in particular, no changes to Kerberos and no reduction in security in either security realm due to the single login. A significant difference between the Legion project and these projects is that Legion attempts to
build security mechanisms the can be viewed as being on top of underlying security mechanisms of host systems, whereas these projects generally attempt to support single sign-on of co-existing realms.

Minsky and Ungureanu address the need of unifying heterogeneous security policies in distributed systems by introducing a formalism that describes various security policies [8]. A unified mechanism is used to enforce the security schemes. This work is important for the construction and analysis of security policies in Legion, in that the Legion mechanism must support a wide variety of local policies. However, in Legion, the approach is that the local sites can have whatever security policy they want, and it is very likely that it will not be specified formally. Requiring every local site to specify their security policies in a single formalism is difficult if not impossible, severely impeding Legion’s deployment.

Yialelis and Sloman describe a security framework for object-based distributed systems [12]. This project is related to the work in Legion, because it attempts to allow the development of secure distributed applications on operating systems with varying degrees of security mechanisms built it. While this work is similar to the Legion mechanisms described in Section 2, the work of Yialelis and Sloman is CORBA-based and does not address the general metasystems requirements, such as hardware heterogeneity and multiple administrative domains.

Globus [4] is another metasystem research project, and as such is addressing many of the same issues as Legion. In many instances, convergent evolution has led to similar solutions to these problems. For example, Globus has a small, easily-verified module called the Gatekeeper that runs as root and is responsible for remote process management, in much the same manner as the PCD. The manner in which Globus integrates with Kerberos is through use of the Generic Security Services API (GSS-API [11]). The level of granularity of the GSS-API and the Legion object model are fundamentally different: In GSS-API, two applications such as FTP and FTPD establish a security context and then communicate based on the security context. In Legion, objects are significantly more fine-grained than applications such as FTP and FTPD—the overhead to establish contexts establishing and deleting security contexts for each pair of communicating objects is intuitively too expensive, as the number of object-object communications is potentially quite large in the life of a computation.

CRISIS [2] is the security architecture for the WebOS project at UC Berkeley. The WebOS provides many of the same high-level services as Legion. WebOS is fundamentally different than Legion in that while WebOS focuses on system-level support for building and running wide-area applications, Legion’s goals are to provide an object-based programming model suitable for such a wide-area application. The principle goals of CRISIS are similar to the goals of the security architecture in Legion: to use redundancy to reduce the likelihood of system compromise, cache whenever possible to improve performance, support fine-grained control over delegated rights, make extensive use of logging, support local autonomy, and to make the design as simple as possible. A difference between the two systems is a result of Legion’s support for autonomy which is not a focus of WebOS: Legion supports dynamically-configured local security mechanisms, and CRISIS supports uniform mechanism across all of the nodes of the wide-area system (although policy within each node may be separately defined).

6 Conclusions

We have introduced the Legion security model, and within that model we have presented three alternatives for adapting Legion to local access control and authentication policies. On a practical level, these designs are important because they are in active use at various local sites within
deployed Legion networks. Therefore, the security implications of each are of great interest to system administrators and users at such sites. On a more general level, these designs demonstrate the degree to which the Legion architecture can accommodate and adapt to site-local requirements.

The Legion system is currently widely deployed, incorporating diverse resources at Supercomputing Centers, Labs, and Universities. For more information about the current status of the Legion system, see http://legion.virginia.edu. The power of the environment increases with the scale and scope of the system. We continue to actively integrate new sites into Legion. A natural part of the evolution requires us to adapt the Legion security architecture to new site-local policies and mechanisms. The work presented here describes our current dominant site configurations. In the future we expect to see this set expand as Legion deployment increases.

References


Using Reflection for Flexibility and Extensibility in a
Metacomputing Environment

Anh Nguyen-Tuong, Steve J. Chapin, Andrew S. Grimshaw
{nguyen | chapin | grimshaw}@virginia.edu
http://legion.virginia.edu
Department of Computer Science
University of Virginia

Charlie Viles
viles@ils.unc.edu
School of Information and Library Science
University of North Carolina at Chapel Hill

Abstract

Legion is a large-scale metacomputing project at the University of Virginia. Legion
users have requirements in many dimensions, including scheduling, security, fault
tolerance, programming languages and environments, and performance. Not all
users have the same needs. Further, as higher levels of services generally imply
higher costs, users should be allowed to make tradeoffs and select the combination
of services that is best suited for their purpose. To support diverse requirements
Legion presents system developers with a reflective model, the Reflective Graph
and Event model (RGE), for building metacomputing applications. The RGE
model uses graphs and events to specify computations and enables first-class
program graphs as event handlers. We demonstrate the RGE model in several areas
of interest to metacomputing using Legion as our testbed. We unify the concepts of
exceptions and events; by making exceptions a special case of events. Furthermore,
using RGE, we demonstrate how to build generic, composable and reusable
components that can be shared across development environments such as MPI,
PVM, NetSolve, Ninf, C++, and Fortran.

Keywords: metasystems, metacomputing, graphs, events, exceptions,
reflection, reflective architecture, component reuse

*This work is partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-
96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE
D459000-16-3C and DARPA (GA) SC H607305A.
1 Introduction

The widespread deployment of gigabit networks will shrink the effective distance between computing resources and enable metasystems—wide-area distributed-object computing systems that consist of many heterogeneous, distributed, and unreliable resources. Without significant software support, metasystem users, e.g., resource owners, administrators, application writers, scientists, language designers, toolkit providers, corporations and government agencies, will not be able to manage the complexity of this environment. Metasystem software must meet users’ requirements in several dimensions, including scheduling, security, fault tolerance, programming languages and environments, accounting, and ease-of-use. As higher levels of service generally imply higher costs, a metasystem should allow users to make tradeoffs and select the combination of services that is best suited for their purpose. For example, a bank may opt for a secure system at the cost of performance, a resource owner may wish to provide access to his machines but only between midnight and six o’clock in the morning, and a scientist may demand performance over other concerns. The set of requirements may change over time. The scientist who previously prized only performance may be willing to sacrifice some performance when resource owners decide to charge for usage.

To meet our users’ expectations, we provided system developers—compiler writers, library writers, and toolkit developers—with a modular architecture that promotes flexibility, extensibility, reusability, and composability. By extensibility, we mean that developers can extend the functionality of the system within a consistent framework. By flexibility, we mean that developers can accommodate a vast range of user requirements.
By reusability, we mean that developers can encapsulate functionality into modular, reusable components. By composability, we mean that developers can compose components to meet users’ requirements, e.g., security and accounting. Our intent is to spur the development of higher-level abstractions, tools and services for application programmers.

Our architecture is based on a reflective model of computation [33]. The basic design philosophy behind a reflective architecture is to expose—instead of hide—the elements that make up the structure of the system to developers. A reflective system is introspective; the system has a representation of itself that it can observe—its self-representation. Often, the self-representation of a reflective architecture is expressed in terms of abstract entities that are manipulated to modify the behavior of the system. Thus a reflective system promotes the writing of generic and reusable components that manipulate the self-representation. Such components may be written by domain experts and incorporated transparently into user applications. For example, Fabre et al. used a reflective architecture to incorporate fault-tolerance techniques into non-fault-tolerant applications [10], thereby freeing application programmers from the complex and error-prone task of implementing fault-tolerance algorithms. The versatility of reflective architectures has been demonstrated in several contexts, such as operating systems [32], programming languages [18][19][20][24], real-time databases [26], agent-based systems [8], and dependable systems [1].

In this paper, we present the Reflective Graph and Event model (RGE) and its application in the Legion metacomputing system [12]. RGE enables the manipulation of user computations at an abstract level by representing them as events, event handlers and
program graphs. These data structures are the self-representation of our reflective architecture and manipulating them is the basis for achieving flexibility, extensibility, reusability, and composability. The advantages of using an event-based architecture are well-known: components are decoupled from one another spatially and temporally, and they may be added/removed dynamically. Developers may extend object functionality by registering handlers with the appropriate events and by defining new events. A novel feature of the RGE event mechanism is that handlers may be executable program graphs that specify method invocations on remote objects. Graphs may be bound with their associated events at run-time, enabling the dynamic composition of functionality to objects.

The primary contributions of this paper are to present a computational model and structural framework for designing objects—the Reflective Graph and Event model; to enable the dynamic binding of policies to objects using first-class executable graphs as event handlers; and to unify the concept of exceptions and events.

We provide an existence proof of the applicability of the RGE model to the solution of real problems by demonstrating its use in Legion, an object-based metacomputing system. We demonstrate the versatility of the model by using it in such diverse areas as building a protocol stack for objects, defining a novel event notification model that includes exception propagation as a special case of the model, implementing a simple bag-of-tasks scheduler, and shutting down distributed applications gracefully.

The paper is organized as follows. In order to frame the RGE model within the context of the Legion project, we briefly describe the Legion system model in Section 2. Then, we describe the Reflective Graph and Event model in Section 3. In Section 4, we present
several applications of the RGE model to support our thesis that RGE is a viable technology for building metacomputing applications, incorporating our design goals of flexibility, extensibility, reusability, and composability. In Section 5, we discuss our experiences working with the RGE model. In Section 6, we present related work. We conclude in Section 7 and present areas of future research.

2 System Model

Before discussing the RGE model and its application in Legion, it will help to place the model within context of the overall Legion system.

Legion is an object-based metacomputing system that has been deployed at more than a dozen sites on three continents. Legion objects encapsulate both hardware and software resources. Objects are logically independent collections of data and associated methods with disjoint address spaces. Objects can contain one or more associated threads of control, and communicate via asynchronous method invocations. Objects are named entities identified by a Legion Object IDentifier (LOID). Objects are persistent and can be in one of two states: active or inert. Active objects contain one or more threads of control and are ready to service method calls. Inert objects exist as passive object state representations on persistent storage. Legion moves objects between active and inert states to use resources efficiently, to support object mobility, and to enable failure resilience. For a detailed description of the Legion object model, please see [14].

Legion provides a variety of programming interfaces on several different levels. Some programmers use Legion by writing programs in high-level languages such as parallel versions of C++, e.g., MPL [13]. Other programmers use Legion by specifying an object interface in an Interface Description Language (IDL), using an IDL compiler to generate
client and server stubs, and then providing the method implementations in a sequential programming language, e.g. CORBA [23]. Others use standard message-passing facilities such as PVM or MPI. Still others may use specialized domain toolkits, e.g., NetSolve [7] and Ninf [25]. Finally, another set of users—toolkit and middleware developers—require direct access to reflective aspects of the Legion run-time system, possibly to add new features and encapsulate them in the form of reusable components.

The RGE model targets the last set of users—toolkit and middleware developers—and provides a unifying architecture for developing components. The building blocks available to developers are reflective graphs and events. RGE program graphs are data-flow graphs whose nodes represent method invocations on objects and whose arcs represent data dependencies. The most important property of graphs is that they are first-class entities, and thus may be manipulated and passed as arguments to other objects for execution. Events provide a structuring mechanism for configuring services in a modular fashion—components may be added easily via event or graph handlers to provide new functionality.

3 Reflective Graph & Event Model (RGE)

Graphs and events specify the computation as it unfolds. Graphs represent interactions between coarse-grain objects; a graph node (vertex) is either a member function call on an object or another graph, arcs model data dependencies, and each input to a node corresponds to a formal parameter of the member function. Events specify interactions between components within an object’s address space. A special kind of event, the exoevent, allows for graphs as event handlers. Thus, raising an exoevent results in remote method invocations, and enables remote objects to be treated as components. The ability
of events to regulate interactions both inside an address space and across address spaces is the foundation of the RGE model.

Below we first present graphs (Section 3.1), followed by events (Section 3.2) and exoevents (Section 3.3).

### 3.1 Graphs

Graphs\(^1\) are the mechanisms in Legion for composing and invoking coarse-grained method functions on objects. Graph nodes are called actors and represent method invocation on objects, arcs denote data-dependencies between actors, and tokens flowing across arcs represent data or control information. When an actor has a token on each of its input arcs, it may "fire", i.e., execute its corresponding method, and deposit a result token on each output arc. Figure 1 illustrates a fragment of code written in C++-like syntax and the corresponding graph representation.

```
(1) main() {
(2)   int a = 10, b = 15, x, y, z;
(3)   MyObject A, B;
(4)   x = A.op1(a);
(5)   y = B.op1(b);
(6)   z = A.op2(x, y);
(7)   printf("z=%d\n", z);
(8) }
```

---

\(^1\) Our use of graphs originated in the Mentat project [13][15], a high-performance object-oriented parallel processing system.
Unlike a traditional client/server model, the results from the method invocations on lines 4 and 5 do not return to the Main object. Instead they are forwarded directly to A.opp. Upon executing the graph, Main sends each node a list of objects that should receive the return values and any out parameters.

Graphs are first-class entities and may be assembled at run-time, transformed, passed as arguments to other objects, and executed remotely. The interface to the graph facilities consists of library routines to build graph nodes, add tokens, add arcs, and annotate nodes, arcs, and tokens. Calls to these can be hand-coded or generated by a compiler front-end or other automated tool. The library also provides routines to execute graphs, probe graph status, and wait on return values. A sample of these features is illustrated in Figure 2, where we show a typical compiler transformation of the code in Figure 1.

In addition to the basic graph operations shown in Figure 2, graphs may be annotated with <name, type, value> triples. The name field is an arbitrary string, the type field is a string that indicates the type, and the value field consists of arbitrary data. The name and type fields dictate the interpretation of the value field. Annotations are properties tied to individual arcs and nodes, e.g. to aid in scheduling we may annotate a node with information such as "Architecture=C90", "Memory Usage=20MB", to indicate architectural restrictions and resource requirements. Similarly we might annotate a node with "Semantic Property=Stateless" to indicate that the function is a pure function – the result depends only on the inputs.

Annotations may propagate through the object method invocation chain, in which case we call them implicit parameters. For example, if object A annotates its graph with an

---

2 A client/server call is a special case of a graph with two nodes: one for the server and the other for the return
implicit parameter, invokes a method on object B, and B invokes a method on object C, A’s implicit parameter propagates to C. Implicit parameters provide a mechanism for adding meta-level information transitively.

As a more concrete example, to monitor message flow dynamically, an application can propagate the identity of a logger object to all objects. Subsequently, each object may build a graph and execute a method on the logger object to pass status information pertaining to the message stream. Implicit parameters are similar to CORBA’s contexts in that they denote meta-level information and are part of the environment when executing a method. The primary difference with CORBA’s contexts is that implicit parameters propagate automatically through the method invocation call chain.

---

3 Implicit parameters are also reminiscent of Unix environment variables.
Figure 2. Example of graph API. Lines 1-32 implement the graph shown on the left. The first step before building the graph is to create and provide handles to our objects (lines 5-11). For each graph node, we specify the method to invoke, and the number of input and output arcs. A graph node is represented by a LegionInvocation (lines 13, 18, 23). Then we create the tokens (arguments) and attach them to the appropriate graph nodes (lines 15-16, 20-21, 25-26). There are two kinds of tokens, constant tokens and invocation tokens. Constant tokens are those for which we have an actual value (lines 15-16, 20-21) whereas invocation tokens are those for which the values are results from other invocations (lines 25-26). Once the graph is built, we execute it (line 27) and block waiting on z, the final output value (line 30).
3.2 Events

There are several ways of structuring objects to support a variety of functions, ranging from the *ad hoc* gluing of components to the establishment of well-defined interfaces. A common way of structuring objects is to use a protocol stack—abstractions or functionality are layered on top of one another. If the set of functions supported by such a protocol stack is static, then hard-wiring components is a suitable approach. On the other hand, if the set of functions is expected to change, a flexible approach is required. In metacomputing systems, the latter approach is needed as the set of services supported by objects is driven by a wide set of user requirements.

We adopted an event-based paradigm for structuring objects. Events are introspective and specify the structural implementation of objects. Events provide a unifying mechanism for inter-component interaction; they are conceptually easy to understand and are familiar to programmers; and they allow the development of components in isolation from other components. Finally, they enable the easy addition or deletion of components, providing a basis for extending the functionality of objects. As described in Section 4.1, we use events to configure the Legion object protocol stack.

Our event model is defined by events, event kinds, event handlers and event managers. An *event* contains user-defined data and a tag that denotes its *event kind*. Each event has one or more associated *event handlers* that may be called whenever an event of that kind is announced. Handlers for a particular event kind are given priorities that determine their execution order. Any handler of a particular priority can postpone or prevent the execution of handlers with lower priorities. Events are announced, or raised, in one of two modes, asynchronous or synchronous. In the former case, an *event manager* stores
the event in an internal queue for later delivery. In the latter, the handlers are invoked immediately.

```
MethodReceive.addHandler(HandlerA, HIGH_PRIO);
data_ptr = ... // set according to application
myEvent = new LegionEvent(MethodReceive,data_ptr);
```

```
Handler List
SomeHandler()
HandlerForY()
MethodReceive
```

```
Handler List
SomeHandler()
HandlerForY()
MethodReceive
```

```
LegionEventManager.announce(myEvent);
```

```
LegionEventManager.flushEvents();
```

Figure 3. Component communication using events. (1) Y registers HandlerForY() with the MethodReceive event. (2) X creates an event. (3) X raises the event by invoking the Event Manager. (4) The Event Manager invokes the handlers.

For example, Figure 3 illustrates communication between two components X and Y within the same object: (1) Y registers a handler for the particular event kind that X will announce. (2) X creates an event using one of the provided event kind as a template. X may attach event-specific data as well. (3) X announces the event to an event manager, which enqueues events and ensures that its handlers are executed in priority order and only if preceding handlers have not prevented further execution. (4) The event manager dequeues and processes the event by calling the associated handlers. Note that apart from application-specific data manipulation, each of these actions requires developers to write only one or two lines of code.

The event facilities enable flexibility and extensibility by allowing components to add, modify and remove handlers. New event kinds may be added, and handler priorities maybe set or reset to affect the order in which handlers are processed.
3.3 Exoevents

In the previous section, events and their handlers resided in the same address space. Exoevents extend events to inter-address space communication. An exoevent is an event whose handler is represented by a program graph. Thus, raising an exoevent will result in a set of method invocations on remote objects. The set of method invocations is bound to the event dynamically and is specified by an executable first-class program graph, effectively implementing a "very long jump". The ability to defer the binding of graphs to events until run-time provides flexibility in implementing policy decisions. Object designers need not anticipate all the myriad ways in which their objects will be used. As an example, consider an object that raises a run-time exception. Where should the exception propagate? Should it be to the immediate caller or perhaps to an exception-monitoring object?

To indicate an interest in an exoevent, a remote object associates a program graph with the exoevent. Typically, the graph is a callback graph: when the exoevent is raised, the graph specifies a method invocation back on the remote object. More complex interactions are possible: the graph may specify a method invocation on a third-party object or a sequence of method invocations on several objects. The association of graphs with exoevents is performed at run-time, and may be set on a per object or per method basis. In the per object case, the association of graphs and exoevents persists across method calls. In the per method case, the association is temporary and valid only for the duration of a single method call.

Exoevents form a substantive portion of the Reflective Graph and Event model and embody a "mechanism, not policy" philosophy. RGE provides hooks to attach event
propagation policies dynamically. The benefit for object designers is that they need not anticipate all possible policies when building their objects. This is illustrated in Section 0, in which a single, shared, server object supports multiple exception propagation policies.

Before we proceed any further, we must first define the terms exoevent, exoevent interest, and exoevent interest set (EIS). An exoevent is a set of 3-tuple items <item-name, data-type, data-value>. The item-name field is a string to identify an item; the data-type specifies how to interpret the data-value field of an item. Items may be added or removed from an exoevent. Users may search for a specific item by using the name field as a key. By convention, all exoevents contain an item with item-name="ExoEventType". The data-type field is a string describing the type of exoevents. By convention, we classify exoevent types within broad categories and further divide them using a ":" to delineate subcategories, e.g., "Exception", "Warning", "Exception:Security", "Exception:Security:Access Control". An exoevent interest is a 2-tuple <exoeventType, notificationGraph> that associates an exoevent type with a computation graph. The exoevent type specifies the kind of exoevent of interest. The notificationGraph is a first-class program graph and specifies a computation to be executed if a match is made between an exoevent and an exoevent interest. An exoevent interest set (EIS) is a set of exoevent interests. An EIS may be used to specify interests in multiple exoevents or to specify multiple graphs for the same exoevent.

3.3.1 Specifying interest in an event (per method association)

To raise an exoevent, we use a RaiseNotification event. The data field of the event contains the raised exoevent. The handler for the event performs a matching function between the raised exoevent and each exoevent interest in the exoevent interest set. If the
handler finds a match, it extracts and executes the notification graph contained in the exoevent interest. If there are multiple matches, then all graphs are executed.

To specify interest in an exoevent for the duration of a single method call, an object creates an exoevent interest, inserts it into an exoevent interest set, (EIS) and annotates the program graph associated with the computation. Since implicit parameters propagate automatically, the EIS will be available to all objects in the call chain that raise an exoevent, i.e., if $S.service()$ invokes methods on other objects, the EIS will propagate to them as well (Figure 4). When an object raises an exoevent, it inspects the exoevent interests contained in the EIS to search for a match based on the exoeventType field. If a match is found, the corresponding notificationGraph is then executed.

![Figure 4. Graph annotated with exoevent interest set (EIS).](image)

Consider the example in Figure 4 that corresponds to the following code fragment ($C$ is the client object, $S$ is a server object):

At the client: $x = S.service();$

The exoevent interest set specified by $C$ is valid during the execution of $S.service()$. This is an important aspect of the model as it enables a single object to support multiple event notification policies, selecting among them on a per method invocation basis.
3.3.2 Specifying interest in an event (per object association)

Exoevent interests may also be specified persistently at the object level and be valid across all method calls to that object. To support this functionality an object uses the methods:

\[
\begin{align*}
\text{RegisterNotification(LOID, ExoeventInterest);} \\
\text{UnregisterNotification(LOID);} \\
\end{align*}
\]

The Legion Object Identifier (LOID) is used to identify the object that registers an interest and to unregister a previous set of registrations. Note that an object may register more than one ExoeventInterest, each with its own notification graph.

Object-level and method-level scoping of notification interests may be specified simultaneously: a single raised exoevent may result in several graphs being executed—some being specified via graph annotations and others via the \text{RegisterNotification()} method of the object in which the exoevent occurs.

4 Applications of the RGE model

We have presented events, exoevents and graphs, the basic building blocks of the RGE model. Next, we demonstrate their utility and versatility in:

- designing a configurable protocol stack (Section 4.1),
- supporting multiple exception propagation policy simultaneously (Section 4.2),
- implementing of a bag-of-task scheduler (Section 4.3),
- implementing a distributed application shutdown algorithm (Section 4.4).

The applications described in this section are examples of reusable and composable components. They have been implemented and are deployed in Legion across several development environments, including PVM, MPI, MPL, NetSolve and Fortran. These applications of the model are not meant to be an exhaustive list of the ways we have used
RGE in Legion. Instead, we illustrate the model's applicability to a variety of needs and informally show its application in other domains.

4.1 Configurable protocol stack

One of the primary applications of the RGE model is to implement a configurable protocol stack for Legion objects [31]. A striking feature of the protocol stack is that only a few events are employed. These events may be classified into three broad categories: message-related, method-related and object management-related events. Events reflect the fact that Legion is an object-based system; objects communicate with method invocations, which are implemented at the low level over message passing. Table 1 describes several event kinds used in configuring the protocol stack.

<table>
<thead>
<tr>
<th>Category</th>
<th>Event Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message-related events</td>
<td>MessageReceive</td>
<td>Object has received a message</td>
</tr>
<tr>
<td></td>
<td>MessageSend</td>
<td>Object is sending a message</td>
</tr>
<tr>
<td></td>
<td>MessageComplete</td>
<td>Message has been successfully sent</td>
</tr>
<tr>
<td></td>
<td>MessageError</td>
<td>Error in sending message</td>
</tr>
<tr>
<td>Method-related events</td>
<td>MethodReceive</td>
<td>Object has received a complete method invocation; all parameters have been received</td>
</tr>
<tr>
<td></td>
<td>MethodReady</td>
<td>A method has passed the security method access control check and is ready to be serviced</td>
</tr>
<tr>
<td></td>
<td>MethodSend</td>
<td>Object is invoking a method on a remote object</td>
</tr>
<tr>
<td></td>
<td>MethodDone</td>
<td>Object is done servicing a method</td>
</tr>
<tr>
<td>Object-related events</td>
<td>ObjectCreated</td>
<td>An object has been created</td>
</tr>
<tr>
<td></td>
<td>ObjectDeleted</td>
<td>An object has been deleted</td>
</tr>
</tbody>
</table>

Table 1. Some of the events used to configure the protocol stack of Legion objects.

Figure 5 illustrates the major components of the Legion protocol stack. To invoke a method on a remote object, the GraphComponent announces a MethodSend event for
each node in the graph that has the sender as a source of an input token. In turn, the MessageLayerComponent bundles parameters into a message and announces a MessageSend event. Finally, the NetworkComponent sends the message over the network.

When an object receives a message from the network, it announces a MessageReceive event. The MethodAssemblyComponent determines whether the received message is sufficient to form a complete method invocation (recall that in data flow multiple messages may be required to trigger a method execution). If the message results only in a partial method invocation, the object stores the message in an internal database. When the required messages arrive to complete the method invocation, a MethodReceive event is raised. At this point, the MethodInvocationComponent, stores the complete method in a database of ready methods. Then, a server loop may extract ready methods from the database and execute them.

![Diagram of Protocol Stack of Object using Components](image)

**Figure 5.** Major components of the Legion protocol stack.
This sequence of events implements our basic requirements for our protocol stack. To extend the basic implementation, e.g., to add security, is simple. For example, adding a security component to encrypt and decrypt the message stream is a matter of registering handlers with the *MessageReceive* and *MessageSend* event kinds. Typically, the encryption handler is the last one registered with *MessageSend* while the decryption handler is the first one invoked as a result of *MessageReceive*. In this example, we assume that the two objects have agreed *a priori* to use compatible encryption/decryption routines. Mechanisms for negotiating protocols are outside the scope of this paper.

### 4.2 Propagating exceptions

A natural way of exploiting the RGE model is for exception propagation between objects. Note that in RGE, exceptions and events are no longer separate concepts—exceptions are simply a special case of events.

Now we demonstrate how a single server can support multiple exception propagation policies simultaneously (Figure 6). Consider a server object *S* used by two applications, *AppA* and *AppB*. *AppA* specifies that objects propagate exceptions back to *AppA*. *AppB* specifies that exceptions propagate to a third-party object, *ExceptionMonitorB*. Finally, the creator of *S*, *CreatorS*, specifies that all exceptions raised by *S* shall propagate to *CreatorS*. *AppA* and *AppB* use implicit parameters to specify an exoevent interest set (per method association), whereas *CreatorS* registers its interest with *S* directly (per object association).
Figure 6. Object $S$ supports multiple exception propagation policies. Exceptions raised servicing method invocations (straight arrow) on behalf of AppA result in a callback method on AppA (curved arrow). Exceptions raised on behalf of AppB (straight arrow) result in a method invocation on ExceptionMonitorB (curved arrow). Any exceptions raised by $S$ (regardless of the invoker) result in a method invocation on CreatorS (curved arrow).

AppA and AppB insert the following exoevent interest set in their implicit parameters to implement the above policies (note that the difference in policies is expressed by the notification graph):

<table>
<thead>
<tr>
<th>Exoevent interest</th>
<th>For AppA</th>
<th>For AppB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExoeventType</td>
<td>Exception</td>
<td>Exception</td>
</tr>
<tr>
<td>NotificationGraph</td>
<td><img src="AppA_notifyException" alt="Graph" /></td>
<td><img src="ExceptionMonitorB_notifyException" alt="Graph" /></td>
</tr>
</tbody>
</table>

CreatorS registers with $S$ to be notified of all exceptions raised by $S$ using:

```java
S.registerNotification(CreatorSLoid, ExoEventInterest);
```

Where ExoEventInterest is given by:
Consider the case when AppA or one of its objects invokes a method on S that results in an exception by S. Since AppA inserted an exoevent interest set in its implicit parameters, the interest propagates to S automatically. Upon raising the exception, S finds the notification graph inserted by AppA and executes it. AppA is thus notified of the exception via the callback method, AppA.notifyException(). Furthermore, since CreatorS registered an interest, it too will be notified of the exception via the method CreatorS.notifyException().

Now consider the case when AppB or one of its objects invokes a method on S. S finds the notification graph inserted by AppB and executes it. This time, the graph specifies the callback method, ExceptionMonitorB.notifyException(). ExceptionMonitorB is thus notified of the exception. As in the previous example, CreatorS.notifyException() will be invoked as well.

These examples demonstrate the flexibility of binding policy to objects at run-time. Designers of object S need not worry about where to propagate exceptions; they need only raise them. Furthermore, S is able to support multiple policies simultaneously by virtue of not supporting any—the policies themselves are specified dynamically by objects external to S.

4.3 Bag-of-tasks scheduling

We next illustrate how we have used the RGE model to implement a self-scheduling policy for stateless objects—objects that embody purely functional method invocations.
We exploit the functional nature of stateless objects to instantiate multiple worker instances and distribute method calls among them (Figure 7).

**Figure 7.** Proxy object selects a worker for servicing a method call. When the worker is done, it raises a `MethodDone` event. The graph associated with the `MethodDone` event results in a callback method invocation `WorkerIsDone()` on the Proxy.

Calls to stateless objects are routed to a `Proxy` object, which then assigns method call requests to one of the workers. The default scheduling policy for `Proxy` was random placement—a worker was selected at random to service method calls.

The `Proxy` is a natural place for experimenting with different scheduling policies. We illustrate how we modified the scheduling policy using the RGE model. Note that neither clients nor workers were aware of the policy change.

Before dispatching a method to the workers, the `Proxy` object builds a notification graph with node `Proxy.WorkerIsDone()` and associates it with the `MethodDone` event exported by the workers. The data field of the event carries a computation ID used by `Proxy` to keep track of ongoing computations. When a worker finishes its assigned computation, it raises the `MethodDone` event. This results in the execution of the notification graph inserted by the `Proxy`. `Proxy` is notified of the completion of a
method call and dispatches a new method invocation to the same worker, assuming that there is work available, thereby achieving self-scheduling.

The presence of events and graphs supporting reflective objects greatly simplifies the task of developing this scheduler. We replaced the default policy with a more sophisticated one, without having to change or modify the workers themselves. The only additional code required is the WorkerIsDone() callback method and minor bookkeeping code in the Proxy object, and the graph for the worker to perform the callback into the Proxy object. This was possible because of the dynamic binding of graphs with the MethodDone event. Further refinements are possible. For example, we could add fault-tolerance capabilities by having the Proxy re-issue lost computations after a timeout interval as we did in an earlier version of Legion [22].

4.4 Shutting down distributed applications

Our last example illustrates an algorithm for shutting down a distributed application, perhaps in response to a keyboard interrupt from the user. At the heart of this algorithm is the use of RGE to keep track of the current set of objects in an application. An object is considered to belong to an application if it was created by a Main object—the first object started in an application—or by a child of the Main object. As the application progresses, the set of objects may grow/shrink as objects create/delete other objects.

To simplify the discussion, we assume that there are no failures. The Main object keeps track of its objects as they are created and deleted by inserting the following exoevent interests in its exoevent interest set:
<table>
<thead>
<tr>
<th>ExoeventType</th>
<th>ObjectCreated</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotificationGraph</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ExoeventType</th>
<th>ObjectDeleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotificationGraph</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 2. Keeping track of objects creation/deletion.

The exoevent interest set propagates to all of the `Main` object's children. Whenever an object is created or deleted, the appropriate graph is executed and `Main` is notified via either an `ObjectCreated()` or `ObjectDeleted()` method call (Table 2). Thus, the `Main` object has a current list of all objects. To shutdown the distributed application, the `Main` object invokes the method `DeleteSelf` on each object in its list.

The `Main` object must also deal with two potential race conditions:

- In-transit deletion notification. The `Main` object is not yet notified of the deletion of an object and will attempt to invoke a `DeleteSelf` method.
- In-transit creation notification. The `Main` object is not yet notified of the creation of an object and hence does not invoke the `DeleteSelf` method.

In-transit deletion notifications are not a problem. When the `Main` object attempts to invoke the `DeleteSelf` method on an already dead object, the invocation will simply fail. In-transit creation notifications can be handled by requiring that before objects respond to the `DeleteSelf` invocation, they must ensure that they have no pending object creations.

---

4 `DeleteSelf()` is an object-mandatory method that all Legion objects must support.
In this example, we used the RGE model to monitor the set of objects in an application. Except for the Main object, objects did not need to be modified to support the shutdown algorithm.

5 Discussion

In the examples described above, the RGE model was used as the basis for supporting and extending object functionality. The default set of events for objects reflects the Legion model of computation—an object-based system implemented over a message-passing layer. We believe this set to be sufficient for most purposes as many algorithms may be expressed in terms of manipulating messages and methods.

Although we target toolkit and middleware developers, we often find it useful and simpler to wrap commonly used policy in higher-level functions. For example, a common way to use the exception propagation model is to propagate exceptions back to the caller. Thus, we provide developers with functions to implement this policy, without requiring them to interact with either the event or graph interface.

While we have discussed the RGE model and its use by developers, we have not discussed the interface provided to application writers. For the most part, application writers never interact with graphs or events. In general, events are raised unbeknownst to application writers. For example, in PVM, MPI, MPL, Fortran, and NetSolve, the distributed shutdown algorithm is triggered when a user hits Control-C. In MPL, the MethodDone events used in the bag-of-tasks scheduling example and the ObjectCreated/ObjectDeleted events are raised automatically by compiler-inserted code.
The ability to compose policies has been invaluable in meeting our users' requirements and in deploying Legion itself. Based on our experiences, we believe that, provided policies are composable at a high-level, we can map them onto the RGE model.

6 Related Work

The RGE model provides a blueprint for structuring distributed applications based on reflective principles. The concept of reflection is not novel; its use has been advocated in several contexts, including operating systems [32], programming languages [18][19][20][24], soft real-time systems [17], real-time global databases [26], agent-based systems [8], dependable systems [1], and in general, to incorporate non-functional requirements into user applications [27].

To our knowledge, RGE is the only reflective model that uses graphs and events as data structures for representing computations. RGE uses the Macro-Data Flow model to express and specify method invocations between objects [15]. Other data-flow systems include Paralex [2], CDF [3], HeNCE [4] and Code/Rope [9]. Unlike most graph systems, RGE graphs are exposed to toolkit and middleware developers; they can be assembled dynamically and executed remotely. RGE graphs are reflective: graphs are the self-representation of a computation and transforming graphs has a direct impact on the future of a computation. Furthermore, we use graphs in a novel way and allow them to be associated with events. This enables us to encapsulate functionality using graphs and dynamically bind such functionality to objects.

Our model shares many characteristics with projects such as SPIN [5], Coyote [6] and Ensemble [16] that use an event architecture as the basis for flexibility, extensibility, and component interaction. One may view Legion as a "configurable operating system"
specifically designed for metacomputing. However, Legion does not replace the operating system on host machines but provides a middleware layer between the native operating system and applications.

CORBA’s Event Notification Service (ENS) [23] and Java’s Distributed Event Specification (DES) [29] provide an event-based notification service. In both, objects must export a well-defined interface to be notified of an event. The RGE model is more flexible and enables an arbitrary set of methods. Furthermore, in RGE, the concepts of exceptions and events are unified—exceptions are simply special kinds of events—in contrast to both CORBA and Java.

Java defines two event models: Java DES to specify the propagation of events between objects on different virtual machines, and Java Beans [30], to specify component interaction inside a single virtual machine. Java DES outlines an approach for transforming Java Beans events into Java Distributed events. We take a similar approach in RGE to export internal events and make them visible to remote objects.

Globus is another metasystem project [11]. The primary difference between Globus and Legion is a philosophical one: Globus employs a “sum-of-service” approach for supporting users and specifies standard interfaces for such functions as security and resource management. Legion employs an “architecture” approach—system developers target a unified model that enables component reuse and interoperability. The two approaches are not mutually exclusive. For example, we have already mapped the two standard message-passing APIs, PVM [28] and MPI [21], onto the RGE model and Legion.
7 Conclusion

We have presented a reflective computational model, the Reflective Graph and Event (RGE) model, and demonstrated its use in Legion, an object-based metacomputing environment. While we used Legion as a proof-of-concept, the model is applicable for metacomputing in general. We have chosen reflection as the design philosophy behind our model because it has been shown to support extensibility, flexibility, composability and reusability, in other contexts such as extensible operating systems and programming languages. Now, we have applied reflection to metacomputing.

Novel features of our models are the uses of graphs and events to specify and represent computations, to allow executable program graphs as event handlers using exoevents, to enable the late binding of policies to objects, and to present an event propagation model that unifies the concept of exceptions and events.

To show the versatility of the RGE model, we presented four applications of the model that have been deployed in Legion: building a configurable protocol stack for objects, defining a novel event notification model that unifies the concept of exceptions with events, and implementing a bag-of-tasks scheduler and a distributed application shutdown algorithm.

RGE encourages the encapsulation of functionality inside reusable components—components developed by one set of system developers may be reused by another. The RGE model provides a structural framework in which components may be composed together in a unified and consistent manner. Thus, not only are the examples shown in this paper deployed and available to Legion's PVM, MPI, NetSolve, C++, and Fortran users, they may also be used within a single application simultaneously.

28
Future work consists of further developing components along several dimensions: fault tolerance, security, and resource management, to name only a few. Over time, we hope to present system developers with an extensive component library; each component having its own costs and benefits tradeoffs. As components are developed by one set of system developers, they can be made available to others as well. Our model encourages this practice as RGE components are by definition generic in nature—they manipulate the underlying computation at an abstract level. Another research direction is to map additional environments and languages onto our model to increase the range of choices afforded end users.

It will be interesting to observe whether the metacomputing community will converge on a standard set of components or whether many components with varying cost and benefit characteristics will emerge. While it is too early to tell, the RGE model provides an experimental platform for the quick prototyping and deployment of components.

8 References


Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications

Anh Nguyen-Tuong and Andrew S. Grimshaw

University of Virginia Department of Computer Science
{nguyen, grimshaw}@virginia.edu
http://legion.virginia.edu

Abstract

As part of the Legion metacomputing project, we have developed a reflective model, the Reflective Graph & Event (RGE) model, for incorporating functionality into applications. In this paper we apply the RGE model to the problem of making applications more robust to failures. RGE encourages system developers to express fault-tolerance algorithms in terms of transformations on the data structures that represent computations—messages and methods—hence enabling the construction of generic and reusable fault-tolerance components. We illustrate the expressive power of RGE by encapsulating the following fault-tolerance techniques into RGE components: two-phase commit distributed checkpointing, passive replication, pessimistic method logging, and forward recovery.

1 Introduction

The advent of fast networks and the wide availability of computing resources make possible the realization of powerful virtual computers, or metasystems, that harness resources on a national or global scale. One of the technological challenges that must be solved before such virtual machines can be used in production mode is the adoption of fault-tolerance techniques for system-level services and user applications. Unfortunately, fault-tolerance protocols are widely regarded as complex. Implementing them correctly is likely to overwhelm all but the best programmers.

Our approach to remedying this problem is to view fault-tolerant applications as the sum of three parts: the application, the fault-tolerance technique, and the infrastructure required to enable their composition. Application programmers should focus on writing applications while fault-tolerance experts should encapsulate algorithms inside components. Within the context of the Legion metacomputing project, we have developed a reflective computational model, the Reflective Graph and Event (RGE) model, for enabling the composition of fault-tolerance techniques with user applications [13][15].

The basic design philosophy behind a reflective architecture is to expose—instead of hide—the elements that make up the structure of the system to developers. A reflective system is introspective; the system has a representation of itself that it can observe—its self-representation. Often, the self-representation of a reflective architecture is expressed in terms of abstract entities that may be manipulated to modify the behavior of the system. Thus, a reflective system promotes the writing of generic and reusable components that manipulate the self-representation. Such components may be written by domain experts and incorporated transparently into user applications. For example, Fabre et al. use a reflective programming language to incorporate fault-tolerance techniques into non-fault-tolerant applications [9], thereby freeing application programmers from the complex and error-prone task of implementing fault-tolerance algorithms.

In this paper, we demonstrate the applicability of the RGE model in encapsulating the following fault-tolerance techniques: distributed checkpointing, passive replication, pessimistic method logging, and forward recovery. The RGE model enables the manipulation of user computations at an abstract level by representing them as events, event handlers and program graphs [26]. These data structures are the self-representation of our reflective architecture and manipulating them is the basis for expressing fault-

*This work is partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C and DARPA (GA) SC H607305A.
tolerance algorithms. The advantages of using an event-based architecture are well-known: components are decoupled from one another spatially and temporally, and they may be added/removed dynamically. Developers may extend object functionality by registering handlers with the appropriate events and by defining new events. A novel feature of the RGE event mechanism is that handlers may be executable program graphs that specify method invocations on remote objects. Graphs may be bound to their associated events at run-time, enabling the dynamic composition of functionality to objects.

The paper is organized as follows. We present related work in Section 2 and introduce the Legion system model in Section 3. We provide an overview of the RGE model in Section 4 and apply the model to encapsulate fault-tolerance techniques in Section 5. We conclude in Section 6.

2 Related Work

The RGE model provides a blueprint for structuring distributed applications based on reflective principles. The concept of reflection is not novel; its use has been advocated in several contexts, including programming languages [21][24], soft real-time systems [18], real-time global databases [29], agent-based systems [6], and in general, to incorporate non-functional requirements into user applications [30].

Reflection has also been used to incorporate fault-tolerance techniques into applications. Lee extends the Common Lisp Object System [21] to support persistence using reflection in [22]. Fabre and Perenou exploit reflective features of the language open-C++ to incorporate fault tolerance, security, and group communication protocols into applications transparently in the FRIENDS project [10]. MAUD is a meta-level architecture for building adaptively dependable systems that has been implemented on an actor-based system [1]. To our knowledge, RGE is the only reflective model that uses graphs and events as data structures for representing computations.

The event paradigm is well established and many systems use it as the basis for extensibility, e.g., Coyote [5], the Java Bean Component Model [31], SPIN [28], and Ensemble [17]. We use the event abstraction within the RGE model to capture and reflect the “internals” of objects to programmers. Events allow programmers to intercept and reroute both messages and method invocations. More importantly, associating events with the acts of receiving/sending messages/methods allows protocol writers to express many algorithms in a natural way by treating messages as abstract entities. Furthermore, RGE events may be associated with graph handlers dynamically—enabling the run-time binding of functionality to objects. Graphs used in RGE are the embodiment of the Macro-Data Flow model [16]. Other data-flow systems include Paralex [2], COSMOS [7], CDF [3], HeNCE [4], Mentat [14][25] and Code/Rope [8]. Of these, Paralex and Mentat support replication while COSMOS uses checkpoint/rollback for fault tolerance. Unlike most graph systems, RGE graphs are exposed to system developers; they can be assembled dynamically and executed remotely. Graphs are reflective: graphs are the self-representation of a computation and transforming them has a direct impact on the future of a computation.

Globus is another metasystem project [11]. The primary difference between Globus and Legion is a philosophical one: Globus employs a “sum-of-service” approach for supporting users and specifies standard interfaces for such functions as security and resource management. Legion employs an “architecture” approach—system developers target a unified model that enables component reuse and interoperability. To our knowledge Globus does not provide integrated support for incorporating fault-tolerance techniques into user applications. Instead, application writers may use a heartbeat monitoring service as a base for implementing fault-tolerance techniques. Note that the two approaches are not mutually exclusive—RGE fault-tolerance components can make use of an external failure detecting service.

3 System Model

Legion is based on an object model of computing. Legion objects encapsulate both hardware and software resources. Objects are logically independent collections of data and associated methods with disjoint address spaces. Objects can contain one or more associated threads of control, and communicate via asynchronous method invocations. Objects are named entities identified by a location-independent Legion Object IDentifier (LOID) and are mapped to Legion Object Addresses (LOA) for actual communication. The LOA of an object includes the necessary information to communicate with it for remote method invocation, e.g., the IP address and port number.

Objects are persistent and can be in one of two states: active or inert. Active objects contain one or more threads of control and are ready to service method calls. Inert objects exist on persistent storage as
passive object state representations (OPR) organized in a directory structure. Legion moves objects between active and inert states to use resources efficiently, to support object mobility, and to enable failure resilience.

Every Legion object is defined and managed by its class object. Class objects in Legion are themselves active objects, and are given system-level responsibility. They create new instances; schedule, activate, and deactivate their instances; and assist client objects in locating instances of the class.

For a detailed description of the Legion object model, please see Grimshaw [15].

4 Reflective Graph and Event Model

As the name indicates, RGE uses graphs and events to specify and represent user computations. We provide an overview of graphs in Section 4.1, events in Section 4.2, followed by a discussion of exoevents—events whose handlers are graphs—in Section 4.3. For a more detailed presentation of the RGE model, please see Nguyen-Tuong et al. [26].

4.1 Graphs

Our use of graphs originated in the Mentat project, a high-performance object-oriented parallel processing system [14]. Graphs are the embodiment of the Macro-Data Flow model, an extension of pure data flow designed for coarse-grained parallel processing. For more details on Macro-Data Flow and how it is used to exploit opportunities for parallelism please see Grimshaw et al. [16].

Graphs specify method invocations and data dependencies between objects. Graph nodes are called actors and represent method invocation on objects, arcs denote data-dependencies between actors, and tokens flowing across arcs represent data or control information. When an actor has a token on each of its input arcs, it may execute its corresponding method, and deposit a result token on each output arc. Figure 1 illustrates a fragment of code and the corresponding graph representation.

(1) main() {
(2)   int a = 10, b = 15, x, y, z;
(3)   MyObject A, B;
(4)   x = A.op1(a);
(5)   y = B.op1(b);
(6)   z = A.op2(x,y);
(7)   printf("z=\d\n", z);
(8) }

Figure 1. Sample code fragment and corresponding RGE program graph

Unlike a traditional client/server model, the results from the method invocations on lines 4 and 5 do not return to the Main object. The graph directly specifies where they should send their return values, namely, A.op2. Thus, graphs are reflective data structures that represent the future flow of the current computation. Graphs are first-class entities and may be assembled at run-time, transformed, passed as arguments to other objects, and executed remotely. Graphs enable system developers to build objects that adapt to their environment by assembling the proper method invocations dynamically and modifying the future flow of the computation.

Graphs may be annotated with <name, type, value> triples. The name field is simply a generic string, the type field indicates the type, and the value field consists of arbitrary data. The name and type fields dictate the interpretation of the value field. Annotations are properties tied to individual arcs and nodes, e.g., “Architecture=C90”, “Memory Usage=20MB”, “Semantic Property=Stateless”, and denote meta-level information. Annotations may propagate through the object method invocation chain, in which case we call

---

1 A client/server call is a special case of a 2-node graph: one for the server and the other for the return value.
them implicit parameters. If object A annotates its graph with an implicit parameter, invokes a method on object B, and B invokes a method on object C, A’s implicit parameter propagates to C. Implicit parameters provide a mechanism for adding meta-level information transitively and are similar to CORBA’s contexts [27]. The primary difference with CORBA’s contexts is that implicit parameters propagate automatically through the method invocation call chain.

4.2 Events

The event paradigm provides a well-understood mechanism for adding new functionality to objects. The versatility of the event paradigm resides in its ability to decouple communication between various components of a system both temporally and spatially—essential features of a component-based systems. Events provide a uniform infrastructure to bind components together. When component X wishes to announce to the system that something of interest has happened, it announces an event E. Components that have registered their event handlers with the event manager previously are notified of the event E. The handlers are then called immediately upon the announcement of E (synchronous), or alternatively, the execution of the handlers may be deferred (asynchronous). In addition, events may carry arbitrary data.

One of the primary applications of the RGE model is to implement a configurable protocol stack for Legion objects [32]. A striking feature of the protocol stack is that only a few events are employed. These events may be classified into three broad categories: message-related, method-related and object management-related events. These categories reflect the fact that Legion is an object-based system implemented at the low level over message passing. Table 1 describes several event kinds used in configuring the protocol stack.

<table>
<thead>
<tr>
<th>Category</th>
<th>Event Data</th>
<th>Event Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message-related events</td>
<td>Message and message</td>
<td>MessageReceive</td>
<td>Object has received a message</td>
</tr>
<tr>
<td></td>
<td>headers</td>
<td>MessageSend</td>
<td>Object is sending a message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MessageComplete</td>
<td>Message has been successfully sent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MessageError</td>
<td>Error in sending message</td>
</tr>
<tr>
<td>Method-related events</td>
<td>Method signature,</td>
<td>MethodReceive</td>
<td>Object has received a complete method invocation; all parameters have been</td>
</tr>
<tr>
<td></td>
<td>arguments, annotations</td>
<td>MethodReady</td>
<td>A method has passed the security method access control check and is ready</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MethodSend</td>
<td>Object is invoking a method on a remote object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MethodDone</td>
<td>Object is done servicing a method</td>
</tr>
<tr>
<td>Object-management</td>
<td>LOID of the object</td>
<td>ObjectCreated</td>
<td>An object has been created</td>
</tr>
<tr>
<td>related events</td>
<td></td>
<td>ObjectDeleted</td>
<td>An object has been deleted</td>
</tr>
<tr>
<td></td>
<td>State of the</td>
<td>SaveState</td>
<td>Saves the state of the object in its OPR (persistent storage)</td>
</tr>
<tr>
<td></td>
<td>object. OPR organized in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>directory structure</td>
<td>RestoreState</td>
<td>Restores the state of the object from its OPR This event is raised upon</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>object startup.</td>
</tr>
</tbody>
</table>

Table 1: Events used to configure the protocol stack of Legion objects

Figure 2 illustrates the major components of the Legion protocol stack. When an object receives a message from the network, it announces a MessageReceive event. The MethodAssemblyComponent determines whether the received message is sufficient to form a complete method invocation (recall that in data flow
multiple tokens/messages may be required to trigger a method execution). If the message results only in a partial method invocation, the object stores the message in an internal database. When the required messages arrive to complete the method invocation, a MethodReceive event is raised. At this point, the MethodInvocationComponent stores the complete method in a database of ready methods. Then, a server loop may extract ready methods from the database and execute them. Once the method finishes executing, a MethodDone event is raised.

On the sending side, the GraphComponent announces a MethodSend event for each node in the graph that has the sender as a source of an input token. In turn, the MessageLayerComponent transforms each parameter into messages and announces a MessageSend event. Finally, the NetworkComponent sends messages over the network.

4.3 Exoevent Notification Model

The RGE model provides several ways of associating graphs and events. One way is for protocol writers to inspect and transform program graphs, or create and execute new graphs, within an event handler. Another more flexible approach is to associate graphs and events dynamically and execute a program graph when an event is raised, thereby enabling the run-time composition of functionality to objects. We call events associated with graphs exoevents to highlight the fact that raising such events may result in a set of remote method invocations. The benefit for object designers is that they need not anticipate all possible policies when building their objects.

Before showing an application of the exoevent notification model, we define the following terms: exoevent, exoevent interest, and exoevent interest set (EIS).

- Exoevent. An exoevent is a set of 3-tuple items <item-name, data-type, data-value>. The item-name field is a string to identify an item; the data-type specifies how to interpret the data-value field of an item. Items may be added or removed from an exoevent. Users may search for a specific item by using the name field as a key. By convention, all exoevents contain an item with item-name="ExoEventType". The data-type field is a string describing the type of exoevents. By convention, we classify exoevents types within broad categories and further divide them using a “:” to delineate subcategories, e.g., “Exception”, “Warning”, “Exception:Security”, “Exception:Security:Access Control”.

- Exoevent Interest. An exoevent interest is a 2-tuple <exoeventType, notificationGraph> that associates an exoevent type with a computation graph. The exoevent type specifies the kind of exoevent of interest. The notificationGraph is a first-class program graph and specifies a computation to be executed if a match is made between an exoevent and an exoevent interest.

- Exoevent Interest Set (EIS). An exoevent interest set is a set of exoevent interests. The EIS propagates to remote objects using implicit parameters.

Consider a server S used by multiple clients (Figure 3). By inserting the proper exoevent interest in its exoevent interest set, each client may specify its own exoevent propagation policy. Client \( C_1 \) specifies that exceptions propagate back to itself whereas \( C_2 \) specifies that warnings propagate to a third-party monitor object.

5 Incorporating Fault-tolerance Techniques into Applications using RGE

We present designs for encapsulating several well-known fault-tolerance techniques using the RGE model: two-phase commit distributed checkpointing, pessimistic method logging, passive replication and forward recovery. To encapsulate fault-tolerance techniques inside components, developers express their algorithms using graphs and events. Typically, this involves inserting handlers with the appropriate events or associating graphs with events.

Note that for these examples, we make the following assumptions:

- Objects are fail-stop, i.e., objects fail by halting and other objects may detect the failure.
- Objects are deterministic. Given a given sequence of input methods, objects will make the same state transitions.
Objects always have access to stable storage via their OPR (Object Persistence Representation). Note that the OPR does not include the program counter or stack of the object as this information is not portable across heterogeneous architectures. This assumption can be relaxed using tools such as April that allow heterogeneous checkpoints [12].

Furthermore, we present only salient features of each technique due to space restrictions.

Figure 3. Clients C₁ and C₂ specify two different exoevent propagation policies. C₁ specifies that exceptions propagate back to it via the notifyException() method. C₂ specifies that warnings propagate to a third-party monitor object via the notifyWarning() method.

5.1 Two-Phase Commit Distributed Checkpointing (2PCDC)

A common method for ensuring the progress of long-running application is to checkpoint its state periodically on stable storage. Checkpoints may be viewed as "insurance policies" against failures—in the event of a failure, the application can be rolled back and restarted from its last checkpoint—thereby bounding the amount of lost work that must be recomputed. As with all insurance policies, there are choices and costs involved. Users may select from a variety of checkpointing algorithms, each providing a specified level of service for a given cost. Costs include the cost of the algorithm itself—memory, CPU, stable storage requirements, and run-time overhead—as well as the cost of implementing a given algorithm correctly. The RGE model directly addresses the latter cost: domain experts encapsulate fault-tolerance techniques into components that may be composed with user applications.

The basic idea behind a two-phase commit distributed checkpointing protocol is to ensure that either all objects in an application checkpoint or none do [23]. The set of local checkpoints taken must form a consistent global state—all methods received by an object must be recorded as having been sent. Two problems must be addressed to ensure a consistent global checkpoint: lost methods and orphan methods. Lost methods are methods that are marked as sent but not received, while orphan methods are methods marked as received but not sent (Figure 4). The algorithm presented here only seeks to prevent orphan methods; lost methods are assumed to be handled by the underlying communication channels.
Figure 4. Two objects, O1 and O2, with local checkpoints (black boxes). Orphan methods result when a method is marked as received in one checkpoint (O2) but not marked as sent in any other. Lost methods result when a method is marked as sent in one checkpoint (O1) but not marked as received in any other.

The algorithm consists of two phases. In phase I, a coordinator requests participants to take a tentative checkpoint. If a participant rejects the request for any reasons, it replies No. Otherwise, the participant takes a tentative checkpoint, replies Yes, suspends communication with other objects, and awaits the coordinator’s decision. If all participants reply Yes, the coordinator’s decision is to commit the checkpoints, otherwise its decision is to abort the protocol. The coordinator’s authoritative decision marks the end of the first phase. In phase II, the coordinator sends its decision to all participants. If the decision is Yes, participants commit the tentative checkpoint taken in the first phase to stable storage. Otherwise, participants may discard the tentative checkpoint previously taken.

<table>
<thead>
<tr>
<th>Coordinator</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase I</td>
<td>Phase I</td>
</tr>
<tr>
<td>requests participants to take</td>
<td>if accept request</td>
</tr>
<tr>
<td>tentative checkpoints</td>
<td>take a tentative checkpoint</td>
</tr>
<tr>
<td>await all replies</td>
<td>reply Yes</td>
</tr>
<tr>
<td>if all replies = “Yes”</td>
<td>suspend communication</td>
</tr>
<tr>
<td>decide Yes</td>
<td>else</td>
</tr>
<tr>
<td>else</td>
<td>reply No</td>
</tr>
<tr>
<td>decide No</td>
<td></td>
</tr>
<tr>
<td>Phase II</td>
<td>Phase II</td>
</tr>
<tr>
<td>inform participants of</td>
<td>if decision = “Yes”</td>
</tr>
<tr>
<td>decision</td>
<td>commit tentative checkpoint</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>discard tentative checkpoint</td>
</tr>
<tr>
<td></td>
<td>resume communication</td>
</tr>
</tbody>
</table>

Table 1. Overview of the Two-Phase Commit Distributed Checkpointing Algorithm

This basic algorithm may be extended in several ways. The coordinator can bound the amount of time that it waits for participants to reply. To handle a coordinator crash, the coordinator can save its decision onto stable storage at the end of phase I. The number of participants may be reduced by exploiting semantic information [23]. For the sake of brevity, we do not include these extensions in our mapping of the algorithm onto the RGE model, nor do we discuss the associated recovery algorithm.

Mapping onto the RGE model

The algorithm is encapsulated using a 2PCDC component. 2PCDC adds the following methods to the participant’s public interface, void TakeTentativeCheckpoint() and void Decision() so that the coordinator may invoke these methods.

In phase I, the 2PCDC component takes a tentative checkpoint by raising a SaveState event. The default handlers for the event write the state of the object into the object’s OPR. To suspend communication at the end of phase I and prevent orphan methods, we ensure that the next method serviced after TakeTentativeCheckpoint() is Decision(). 2PCDC adds a handler, AwaitDecision, to the MethodReceive event. AwaitDecision intercepts all methods until the receipt of Decision(), at which point AwaitDecision announces a MethodReady event.

---

2 Similarly to the checkpointing algorithm, the recovery algorithm uses a two phase-commit protocol
To commit the checkpoint in phase II, 2PCDC creates a new directory in the OPR, "2PC-Commit", in which it writes the state of the object.

We ensure that there are no lost messages by preventing lost messages (recall that multiple messages may be needed to form a method invocation). Upon receipt of a message, an object raises a MessageReceive event. The sender of the message registers its interest in MessageReceive using the Exoevent Notification Model (Section 4.3). If the invoker is not notified of MessageReceive in a timely manner, it retransmits the message. To handle duplicate messages, the invoking object appends a message identification number. The invoked object may then discard duplicates based on this number.

5.2 Pessimistic Method Logging (PML)

The two-phase commit distributed checkpointing algorithm requires objects to coordinate their local checkpoints to establish a consistent application global state. Further, during recovery, events that did not fail are potentially required to rollback their state. We now describe an adaptation of pessimistic message logging [33] for an object-based environment—pessimistic method logging (PML)—in which objects establish checkpoints and recover independently from one another.

In PML, objects checkpoint their state periodically and log received methods onto stable storage upon receipt before delivering the methods to the application layer. In the event of a failure, objects restart from their saved checkpoint and replay their log. Since objects are deterministic, replaying methods in the same order will produce the same execution (Figure 5).

PML is attractive due to its simple recovery characteristic—objects restart independently without the need for a costly coordination protocol. The disadvantage of PML is the high cost of saving methods onto stable storage. We do not discuss here techniques to reduce the overhead of pessimistic logging [19][20].

\[ O_1 \rightarrow (a) \rightarrow M_1(m_1,m_2) \rightarrow (b) \rightarrow M_2(m_3) \rightarrow (c) \rightarrow M_3(m_4) \]

Figure 5. Multiple messages (lowercase m) may be needed to form a method invocation (uppercase M). Object O₁ crashes at (a), (b) or (c). If O₁ crashes at (a), O₁ replays messages m₁ and m₂ from the log. If O₁ crashes at (b), O₁ replays the method M₁ from the log. During recovery, message m₄ is retransmitted by O₂. If O₁ crashes at (c), O₁ replays method M₁ and message m₄ from the log.

Mapping onto the RGE model

The algorithm is encapsulated using a PML component. PML creates the following directories in the OPR of the object, "/MessageLog" and "/MethodLog". PML inserts a LogMessage handler with the MessageReceive event as well as a LogMethod handler with the MethodReady event. When an object receives a message, LogMessage writes it into the "/MessageLog" directory. If the received message results in a full method invocation (recall that in our model, multiple messages may be needed to form a method), a MethodReady event is generated. LogMethod catches the event and writes the method into the "/MethodLog" directory. To reclaim storage space, LogMethod also deletes the messages associated with the received method from "/MessageLog". PML also inserts a Restart handler with the RestoreState event to ensure that PML is notified when an object restarts.

When communication is attempted on a crashed object, its class will restart it on an available host and restore its saved state. In the process, a RestoreState event is raised and caught by the Restart handler. Restart first replays the partial methods, i.e., messages, contained in "/MessageLog". Then, it replays the methods contained in "/MethodLog" before allowing normal processing to resume for the object. Replaying the method log may result in duplicate method invocations on remote objects. To prevent methods from executing multiple times, objects append to each message a unique message identifier so that receiving objects may discard duplicate entries.
To prevent lost methods and messages, PML uses the `MessageReceive` exoevent as described previously in Section 5.1. There are no orphan messages since all received messages are stored in the "MessageLog" directory in the OPR of objects.

### 5.3 Passive Replication

In passive replication, a primary object services method invocations. When the primary object finishes servicing a state-updating method it sends its new state to a backup object before replying to the caller. Upon failure of the primary, the backup takes over and services subsequent method invocations (Figure 6).

![Figure 6](image)

Figure 6. To communicate with O, a client first obtains a binding from Class O [1][2]. The client invokes a state-updating method, `write()`, on Primary(O) [3]. Before the result of `write()` is returned to the client [5], Primary(O) first forwards its state to the Backup [4]. If the primary fails, Class O invokes `BecomePrimary()` on the backup. Subsequent binding requests from clients will result in a binding to the new primary. Note that the methods `ReceiveState()`, `ReceiveReply()` and `BecomePrimary()`, are added transparently to the user code.

#### Mapping onto the RGE model

The passive replication algorithm is encapsulated inside of a PassiveReplication component. Inside the primary, PassiveReplication inserts a `SendStateToBackup` handler with the `MethodDone` event. If PassiveReplication is contained within the backup, it adds and exports the methods `BecomePrimary(LOID NewBackup)`, `ReceivePrimaryState(State S)`, and `ReceiveReply(Reply R)`. At the primary, if the method serviced is state-updating, `SendStateToBackup` extracts the state of the object from its OPR and forwards it by invoking the method `SendStateToBackup` on the backup. `SendStateToBackup` determines whether a method is state-updating by inspecting the function signature. If the signature is not of the form "read-only return-type func(args...)", then the method is state-updating. If the method is non-state-updating, the primary sends only the reply value to the backup by invoking `ReceiveReply()`. At the backup, PassiveReplication waits for the invocation of either the `BecomePrimary()` or `ReceivePrimaryState()` methods. If `BecomePrimary()` is invoked, the backup becomes primary and forwards its state to the new backup. Since the old primary can crash before sending the return value to the client, the new primary resends the last return value. Thus, clients may receive duplicate return values. We assume that clients can handle duplicate values.

When a binding request is issued for a crashed object, the default behavior is for the class of the object to restart the object on an available host, and return the new binding. Instead, the class now selects a replica as the new primary and invokes the `BecomePrimary()` method on the new primary, before returning the binding of the new primary. Passive replication results in faster recovery of crashed objects than the default algorithm as the backups are already active and ready to service methods.

### 5.4 Forward recovery

In forward recovery, applications do not rollback to a previously consistent state. Instead, they attempt to repair themselves so as to continue processing from a consistent state. By its nature, forward recovery is application-dependent and not as general as the backward-recovery methods discussed in the previous sections.
Our approach for supporting forward recovery is to use the exoevent notification model described in Section 4.3 in which the concepts of raising and propagating exceptions are decoupled. Thus, object writers need not specify exception propagation policies at design time.

Mapping onto the RGE model

If an object wishes to be notified of an exoevent raised by objects in its future call chain, it inserts an exoevent interest in its exoevent interest set. Consider a remote method invocation in which a client C invokes a method service on an object. To be notified of all exceptions raised by S.service, C annotates its program graph with the exoevent interest shown in Figure 7. Since the exoeventType field is set to "Exception", all exceptions propagate back via the notifyException method on C.

![Figure 7. Client C specifies interest in exceptions raised by S.](image)

6 Conclusion

To achieve our goal of alleviating the difficulty of writing robust metacomputing applications, we have presented a reflective model of computation, the Reflective Graph and Event model, for expressing fault-tolerance techniques inside reusable components and enabling the composition of such components with user applications. We have presented designs for mapping several well-known fault-tolerance techniques to the RGE model: two-phase commit distributed checkpointing, passive replication, pessimistic logging, and forward recovery.

The RGE model is implemented and deployed within the Legion metacomputing system. The forward recovery example has also been implemented and is in use. Future work consists of implementing and deploying the other examples described in this paper, mapping other fault-tolerance techniques onto the RGE model, and evaluating the performance of the resultant system.

7 References


Resource Management in Legion

Steve J. Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw

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

Abstract

The recent development of gigabit networking technology, combined with the proliferation of low-cost, high-performance microprocessors, has given rise to metacomputing environments. These environments can combine many thousands of hosts, from hundreds of administrative domains, connected by transnational and worldwide networks. Managing the resources in such a system is a complex task, but is necessary to efficiently and economically execute user programs.

In this paper, we describe the resource management portions of the Legion metacomputing system, including the basic model and its implementation. These mechanisms are flexible both in their support for system-level resource management but also in their adaptability for user-level scheduling policies. We show this by implementing a simple scheduling policy and demonstrating how it can be adapted to more complex algorithms.

Keywords: parallel and distributed systems, task scheduling, resource management, autonomy

1 Introduction

The recent development of gigabit networking technology, combined with the proliferation of low-cost, high-performance microprocessors, has given rise to metacomputing environments. These environments can combine many thousands of hosts, from hundreds of administrative domains, connected by local,

* 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.

Preprint submitted to Elsevier Science 5 October 1998
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 project is developing metacomputing software, and in this paper, we will describe the resource management subsystem of Legion. In particular, we will describe the Legion scheduling model, our implementation of the model, and the use of these mechanisms to support user-level scheduling.

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. 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 ensure that their programs receive the best treatment, while administrators want to ensure that their systems are safe, secure, and available for their priority users. Legion provides a methodology allowing each group to express their desires, and the system acts as a mediator to find a resource allocation that is acceptable to both parties. With such a system in place, users may neither know nor care whether their jobs are running across the hall or across the country. Administrators can offer excess cycles to the Legion system, or even set up workstation farms selling cycles as a commodity, secure in the knowledge that their local access and use policies will be respected.

Legion achieves this vision through a flexible, modular approach to scheduling support. Throughout the paper, we will refer to the current implementation of Legion, or the default behavior. This is because Legion is fundamentally a set of interface definitions for an object system, and our prototype is only one implementation that manifests those interfaces. 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.

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 and heterogeneity, although other issues certainly play a role.

Supporting heterogeneity requires Legion to accommodate vastly differing computing capabilities among constituent machines, including differences in architectures, operating systems, and installed software. Such support is important to run complex distributed computations, such as a weather forecasting and visualization program—portions of the computation may be best suited for vector supercomputers, message-passing architectures, or graphics workstations. Autonomy means that each site has the freedom to have heterogeneous resources, define local policies, and refuse to run jobs from remote sites. Users have the freedom to choose where they would like their jobs to run, and to decline an unsatisfactory choice made by the system.

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.
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 defines the create_instance() method, which is responsible for placing an instance on a viable host. 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.\(^1\)

The two remaining core objects represent the basic resource types in Legion: Hosts and Vaults.\(^2\) 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 working with the Globus project and the NSF PACI centers to implement generic functionality that will allow Host Objects to 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 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

\(^1\)The current default is to place the object “here,” i.e. using the class’s Host and Vault, if possible.

\(^2\)We are developing Network Objects to encapsulate host connectivity and interconnection.
other object in the system to be able to decode the reservation token. 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 Object Persistent Representation (OPR). The OPR holds the persistent state of the object, and is used for migration and for shutdown/restart purposes.

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 [14,15]; 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 behalves. 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 third 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 special knowledge of the application. Application writers 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. Before we examine each component in detail, we will examine their interactions at a higher level. Note that figure 3 and the following discussion are intended to detail 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:

(i) The Collection is populated with information describing the resources.
(ii) The Scheduler queries the Collection, and
(iii) based on the result and knowledge 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.
(iv) This mapping is passed to the Enactor, which
(v) invokes methods on hosts and vaults to
(vi) obtain reservations from the resources named in the mapping.
(vii) After obtaining reservations, the Enactor consults with the Scheduler to confirm the schedule, and
(viii) after receiving approval from the Scheduler,
(ix) attempts to instantiate the objects through member function calls on the
appropriate class objects.
(x) The class objects report success/failure codes, and
(xi) the Enactor returns the result to the Scheduler.
(xii) If, during execution, a resource decides that the object needs to be migrated, it performs an outcall to a Monitor,
(xiii) Which notifies the Scheduler and Enactor that rescheduling should be performed.

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.

<table>
<thead>
<tr>
<th>Reservation Management</th>
<th>Process Management</th>
<th>Info. Reporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_reservation()</td>
<td>startObject()</td>
<td>get_compatible_vaults()</td>
</tr>
<tr>
<td>check_reservation()</td>
<td>killObject()</td>
<td>vault_OK()</td>
</tr>
<tr>
<td>cancel_reservation()</td>
<td>deactivateObject()</td>
<td></td>
</tr>
</tbody>
</table>

Table 1
Host Object Resource Management Interface

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.

In addition to the information reporting methods listed above, the Host also supports the attribute database included in all Legion objects. These information reporting methods for Host Objects allow us to build Collections using a pull model—the Collection can query the host to determine its current state. 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. Future versions of host objects will export scheduling policy information so that user-level Schedulers can better determine whether particular hosts are good candidates for object placement.

The Host Object reassesses its local state periodically, and repopulates its attributes. If a push model 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 Host Objects do. 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

---

8 Our current default is a push model, although we are implementing intermediate agents while will pull data from hosts and push it into collections.
int JoinCollection(LOID joiner);
int JoinCollection(LOID joiner, LinkedList <Uval_ObjAttribute>);
int LeaveCollection(LegionLOID leaver);
int QueryCollection(String Query, &CollectionData result);
int UpdateCollectionEntry(LOID constituent, LinkedList <Uval_ObjAttribute>);

**Fig. 4. Collection Interface**

```
int-binop  =>  +  |  -  |  /  |  *  |  mod  |  &  |  |
             max  |  min
int-expr   =>  int-expr  int-binop  int-expr  |
             (int-expr)  |  integer  |
             int(float-expr)  |  id
string-expr =>  string-expr  +  string-expr  |
             (string-expr)  |  string  |  id
float-binop =>  +  |  -  |  /  |  *  |  max  |  min
float-expr  =>  float-expr  float-binop  float-expr  |
             (float-expr)  |  float  |
             float(int-expr)  |  id
comp        =>  <  |  >  |  =  |  >=  |  <=  |  <>
bool-binop  =>  and  |  or  |  xor
bool-expr   =>  bool-expr  bool-binop  bool-expr  |
             not  bool-expr  |
             int-expr  comp  int-expr  |
             float-expr  comp  float-expr  |
             string-expr  comp  string-expr  |
             match(string-expr, string-expr)  |
             (bool-expr)  |  true  |  false  |  id
```

**Fig. 5. Grammar for Collection Query Language**

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. A Collection query is string conforming to the grammar in figure 5, which is largely the same as that used 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 $\$Attribute$.name.

For example, to find all hosts that run the IRIX operating system version 5.x, one could use the regular expression matching feature for strings and query
as follows:

\[
\text{match(\$host.os.name, "IRIX") and match(\$host.os.name, "5\..*")}
\]

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.

An important use of Collections is to structure resources within the Legion system. Having a few, global, Collections will prohibit the scalability we wish to achieve. Therefore, Collections may receive data from, and send data to, other Collections. This allows us to have a Collection for each administrative domain, and to combine Collections in other Collections. This is analogous to the hierarchical structuring of scheduling modules in [4], and we expect to see the same scalability benefits realized there.

### 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 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.

Schedules must be passed between Schedulers and Enactors. A graphical representation for a Schedule appears in figure 6. 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 \(\text{Class LOID} \rightarrow (\text{Host LOID} \times \text{vault LOID})\). Each mapping indicates that an instance of the class should be started on the indicated \((\text{host}, \text{vault})\) pair. In the future, this mapping process may also select from among the available implementations of an object as well.
Fig. 6. The Schedule data structure

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 6. 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 7. 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.

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

Fig. 7. Enactor Interface

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 de-
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).

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 [14]. Using this mechanism, the Enactor can register an outcall
to 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. If desired, the Enactor or Scheduler
can 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 mecha-
nisms. We start with a simple random policy, and demonstrate how to build a
"smarter" Scheduler based on the simple random policy. This improved Sched-
uler provides a template for building Schedulers with more complex placement
algorithms. We then discuss our plans for building more sophisticated Sched-
ulers with application and domain-specific knowledge.
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. 8. Pseudocode for random placement

For the sake of brevity and presentation, we have omitted the full source code in favor of pseudocode. The source code is contained in release 1.4 of the Legion system, first made available in September 1998. The current release of the Legion software is available from [9].

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 8. 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 attribute list of the class of the 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.

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 1.4.

After generating the mapping, the Scheduler must interact with the Enactor to determine if the placement was successful. Although not shown in figure 8, 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 9 and 10.

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 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. Again, this is a simple-minded approach to solving the problem, but serves to demonstrate how one could construct a richer Scheduler.

4 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.
IRS.Generate_Placement(ObjectClass list, int n) {
  for each ObjectClass 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 \( H \) at random
        extract list of compatible vaults from \( H \)
        randomly pick a compatible vault \( V \)
        append the target \((H, V)\) to the list for this instance
      }
  }
  master schedule = first item from each object instance 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. 9. Pseudocode for the IRS Placement Generator

IRSWRAPPER(ObjectClass list) {
  for \( i \) in \( 1 \) to \( \text{SchedTryLimit} \), do {
    sched = IRS.Generate_Placement(ObjectClass List, NSched);
    for \( j \) in \( 1 \) to \( \text{EnactTryLimit} \), do {
      if (make_reservations(sched) succeeded) {
        if (enact_placement(sched) succeeded) {
          return success;
        }
      }
    }
  }
  return failure;
}

Fig. 10. Pseudocode for the IRS Wrapper

4.3 Specialized Policies

We are in the process of defining and implementing specialized placement policies for structured multi-object applications. Examples of these applica-
tions 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.

There are many software systems for managing a locally-distributed multi-computer, including Condor [11] and LoadLeveler [13]. 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. We have examined the components of the RM 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 expect to incorporate Network Objects as a core Legion resource in late 1998 or early 1999. 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


A New Model of Security for Metasystems

Steve J. Chapin, Chenxi Wang, William A. Wulf, Fritz Knabe, and Andrew Grimshaw

Department of Computer Science, University of Virginia, Charlottesville, VA 22903-2442, {chapin, cw2e, wulf, knabe, grimshaw}@cs.virginia.edu

Abstract

With the rapid growth of high-speed networking and microprocessing power, metasystems have become increasingly popular. The need for protection and security in such environments has never been greater. However, the conventional approach to security, that of enforcing a single system-wide policy, will not work for the large-scale distributed systems we envision. Our new model shifts the emphasis from "system as enforcer" to user-definable policies, making users responsible for the security of their objects.

This security model has been implemented as part of the Legion project. Legion is an object-oriented metacomputing system, with strong support for autonomy. This includes support for per-object, user-defined policies in many areas, including resource management and security. This paper briefly describes the Legion system, presents our security model, and discusses the realization of that model in Legion.

Keywords: security, metasystems

1 Introduction

High-speed networking has significantly changed the nature of computing, and specifically gives rise to a new set of security concerns and issues. The conventional security approach has been for a single authority ("the system") to mediate all interactions between users and resources, and to enforce a single system-wide policy. This approach has served us well in the environment of a centralized system because the operating system implements all the key components and knows who is responsible for each process.

*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.

Preprint submitted to Elsevier Science 5 October 1998
However, in a metasystem several things have changed:

- Distributed Kernel: There is no clear notion of a single protected kernel. The path between any two objects may involve several machines that are not equally trusted.
- System Scope and Size: The system is usually much larger than a centralized one. We expect it to be a federation of distinct administrative domains with separate authorities.
- Heterogeneity: The system may involve many subdomains with distinct security policies, channels that are secured in several ways, and platforms with different operating systems.

The intricate nature of metasystems has fundamentally changed the requirements for system security. Within the Legion project, we are investigating a new model of computer security appropriate to large distributed systems.

Users of Legion-like systems must feel confident that the privacy and integrity of their data will not be compromised—either by granting others access to their system, or by running their own programs on an unknown remote computer. Creating that confidence is an especially challenging problem for a number of reasons; for example:

- We envision Legion as a very large distributed system; at least for purposes of design, it is useful to think of it as running on millions of processors distributed throughout the galaxy.
- Legion will run on top of a variety of host operating systems; it will not have control of the hardware or operating system on which it runs.
- There won't be a single organization or person that “owns” all of the systems involved. Thus no one can be trusted to enforce security standards on them; indeed, some individual owners might be malicious.

No single security policy will satisfy all users of a huge system—the CIA, NationsBank, and the University of Virginia Hospital will have different views of what is necessary and appropriate. We cannot even presume a single “login” mechanism—some situations will demand a far more rigorous one than others. And, for both logical and performance reasons, the potential size and scope of Legion suggests that we should not have distinguished “trusted” components that could become points of failure, penetration, or bottlenecks.

Running “on top of” host operating systems has many implications, but in particular it means that we must assume additional security weaknesses in addition to the usual assumption of insecure communication. We assume that copies of Legion system objects will be corrupted (rogue Legionnaires), that some other agent may try to impersonate Legion, and that a person with “root” privileges to a component system can modify the bits arbitrarily.
The assumption of "no owner" and wide distribution exacerbates these issues. Because Legion cannot replace existing host operating systems, the idea of securing them all is not a feasible option. We presume that at least some of the hosts in the system will be compromised, and may even be malicious.

These problems pose new challenges for computer security. They are sufficiently different from the prior problems faced by single-host systems that some of the assumptions that have pervaded work on computer security must be re-examined. Consider one such assumption: that security is absolute; a system is either secure or it is not. A second assumption is that "the system" is the enforcer of security.

In the physical world, security is never absolute. Some safes are better than others, but none is expected to withstand an arbitrary attack. In fact, safes are rated by the time they resist particular attacks. If a particular safe isn't good enough, its owner has the responsibility to get a better one, hire a guard, string an electric fence, or whatever. It isn't "the system," whatever that may be, that provides added security—that burden rests on the owner of the object.

Note that we said that users must feel confident that the privacy and integrity of their data will not be compromised; we did not say that they had to be guaranteed of anything. Security needs to be "good enough" for a particular circumstance, at a cost commensurate with the protection provided. Of course, what is good enough in one case may not be in another—so we need a mechanism that first lets the user know how much confidence they are justified in having, and second provides an avenue for gaining more when required.

The phrase "trusted computing base" (TCB) is common when referring to systems that enforce a security policy. The mental image is that "the system" mediates all interactions between users and resources, and for each interaction decides to permit or prohibit it based on consulting a "trusted data base"; the Lampson access matrix [4] is the archetype of such models.

As with the previous assumption, this one just doesn't work in a Legion-like context. In the first place there isn't a single system-wide policy. New policies may emerge all the time, and the complexities of overlapping or intersecting security domains blur the very notion of a perimeter to be protected. In the second place, since we have to presume that the code might be reverse-engineered and modified, we cannot rely on the system enforcing security—at most, we can view it as a set of interfaces, protocols, and agents, some of whom we trust.

Moreover, security has a cost in time, convenience, or both. The intuitive determination of how much confidence is "good enough" is moderated by cost considerations. As has been observed many times, one reason that extant computer systems have not paid more attention to security is that the cost,
especially in convenience, is too high. These prior systems took the approach that security is absolute, and everyone either paid the cost of full security or had none, regardless of their individual needs. To succeed, our model must scale along cost—it must have essentially zero cost if no security is needed, and the cost must increase in proportion to the extra confidence one gains. Further, these costs must scale on the basis of individual objects, not only for the system as a whole.

These observations call for a change in our way of thinking and a shift in security paradigm. In the rest of the paper, we suggest a new security model that differs from the traditional approach, and describe the current implementation of the model within the Legion metasystem. We also illustrate ideas to deal with the issues raised above, as well as others. Before proceeding to describe our plan of attack, the following describes the Legion system to provide context.

2 Background: The Legion Project

The Legion [3] project at the University of Virginia is an attempt to provide metasystem services that create the illusion of a single virtual machine. This machine provides secure shared object and name spaces, high performance via both task and data parallelism, application adjustable fault tolerance, improved response time, and greater throughput. In all cases, we allow and encourage per-object user-definable resource policies. Legion is targeted towards wide-area assemblies of workstations, supercomputers, and parallel supercomputers. Such a system will unleash the integrated potential of many diverse, powerful resources which may very well revolutionize how we work, how we play, and in general, how we interact with one another.

The potential benefits of a metasystem such as Legion are enormous. We envision (1) more effective collaboration by putting coworkers in the same virtual workplace; (2) higher application performance due to parallel execution and exploitation of off-site resources; (3) improved access to data and computational resources; (4) improved researcher and user productivity resulting from more effective collaboration and better application performance; (5) increased resource utilization; and (6) a considerably simpler programming environment for applications programmers.

Legion is an object-oriented metasystem. The principles of the object-oriented paradigm are the foundation for the construction of Legion; All components of interest in Legion are objects, and all objects, including classes, are instances of classes. Use of the object-oriented paradigm enables us to exploit encapsulation and inheritance, as well as providing benefits such as software reuse,
fault containment, and reduction in complexity.

Hand-in-hand with the object-oriented paradigm is one of our driving philosophical themes: we cannot design a system that will satisfy every user’s needs, therefore we must design an extensible system. This philosophy manifests itself throughout, particularly in our use of delayed binding and what we call “service sliders.” For example, there is a trade-off between security and performance (due to the cost of authentication, encryption, etc.). Rather than providing a fixed level of security, we allow users to choose their own trade-offs by implementing their own policies or using existing policies via inheritance. Similarly, users can select the level of fault-tolerance that they want—and pay for only what they use. By allowing users to implement their own services, or inherit from library classes, we provide the user with flexibility while at the same time providing a menu of existing choices.

3 The Security Model

In this section we describe the security model and its current implementation in Legion. We first present the design guidelines and principles. We discuss the trade-offs and our design decisions. We then explain how the model works, with particular focus on how it can be used to enforce discretionary policies.

3.1 Design Principles

The Legion Security model is based on three principles:

(i) as in the Hippocratic Oath, do no harm!
(ii) caveat emptor—let the buyer beware.
(iii) small is beautiful.

Legion’s first responsibility is to minimize the possibility that it will provide an avenue via which an intruder can do mischief to a remote system. The remote system is, by the second principle, responsible for ensuring that it is running a valid copy of Legion—but subject to that, Legion should not permit its corruption.

The second principle means that in the final analysis users are responsible for their own security. Legion provides a model and mechanism that make it feasible, conceptually simple, and inexpensive in the default case, but in the end the user has the ultimate responsibility to determine what policy is to be enforced and how vigorous that enforcement will be. This, we think, also
models the real world; the strongest door with the strongest lock is useless if
the user leaves it open.

The third principle simply means, given that one cannot absolutely, uncondi-
tionally depend on Legion to enforce security, there is no reason to invest it
with elaborate mechanisms. On the contrary, at least intuitively, the simpler
the model and the less it does, the lower the probability that a corrupted
version can do harm. The remainder of the paper describes such a simple
model.

As noted above, Legion is an object-oriented system. Thus, the unit of pro-
tection is the object, and the “rights” to the object allow invocation of its
member functions (each member function is associated with a distinct right).
This is not a new idea; it dates to at least the Hydra system in the mid 1970’s
[12] and is also in some proposed CORBA models [2]. Note, however, that
it subsumes more common notions such as protection at the level of file sys-
tems. In Legion, files are merely user-defined objects, which happen to have
methods read/write/seek/etc. Directories are just another type of object with
methods such as lookup/enter/delete/etc. There is no reason why there must
be only one type of file or one type of directory and, indeed, these need not
be distinguished concepts defined by, or even known to Legion.

The basic concepts of the Legion Security Model are minimal; there are just
four:

(i) every object provides certain known member functions (that may be de-
aulted to NIL); we will describe MayI, CanI, Iam, and Delegate.
(ii) there are two objects associated with each operation: a responsible agent
(RA) and a calling agent (CA). The RA is someone who can be held
accountable for the particular operation. The CA is the object that ini-
tiated the current method call. The RA is a generalization of the “user
id” in conventional systems; for the moment it is adequate to think of it
as identifying the user or agent who was responsible for the sequence of
method invocations that lead to the current one. There are a certain set
of member functions associated with an RA object. User-defined objects
can act as RA by supplying these member functions.
(iii) every invocation of a member function is performed in the context of
a certificate which contains the Legion Object ID (LOID) of the RA
which generated the certificate, a list of allowed method invocations, and
a timeout. The certificate is digitally signed by the maker.
(iv) there are a small set of rules for actions that Legion will take, primarily
at member function invocation. These rules are defined informally here.

The general approach is that Legion will invoke the known member functions
(MayI, etc.) at the appropriate time, thus giving objects the responsibility
Fig. 1. Object A calls B.foo(), automatically invoking B.MayI()

of defining and ensuring the policy by providing their own implementations of those well-known functions. Precisely how this happens is detailed in the following sections.

3.2 Protecting Oneself—Privacy

In Legion users are responsible for their own security. They are the ones who decide how secure their applications ought to be, and from there, which policy is to be enforced and how rigorous the enforcement should be. For example, a truly paranoid user’s object can include code in every method to authenticate the caller and to determine whether that caller has the right to make this call. For many users, however, this degree of caution is unnecessary and some delegation to the Legion mechanism is appropriate—for example, rather than engaging in an authentication dialog with the caller, an object might trust that the CA is correct.

Our first objective is to have policies defined by the objects themselves. At the same time, we don’t want to have to include policy-enforcement code in every member function unless the object is particularly sensitive. So, instead, we require that every class define a special member function, MayI (this can be defaulted, but we’ll ignore that for now). MayI defines the security policy for objects of that class. Legion automatically calls the MayI function before every member function invocation, and permits that invocation only if MayI sanctions it (see figure 1).

In figure 1, Object A invokes method B.foo. This call is passed to B, and the Legion run-time system automatically invokes B.MayI rather than invoking foo.¹ If B.MayI returns true, then foo is invoked with the arguments passed from A. If not, then an exception is raised and passed back to A (Legion exception handling is beyond the scope of this paper). All the information

¹Ignore the call to A.Iam for the moment.
necessary to make such a decision (the calling Agent (A), the method being invoked (foo), and the parameters of the call) are available as input to MayI.

Note how this simple idea begins to meet our objectives. First, it permits the creator of an object class to define the privacy policy for objects of that class; there is no system-wide policy. Second, it is fully extensible—when a user defines a new class its member functions become the “rights” for that class and its MayI function/policy determines who may exercise those rights. Third, it is fully distributed. Fourth, it is not particularly burdensome; users can default MayI to “always OK,” inherit a MayI policy from a class they trust, or write a new policy if the situation warrants. Fifth, the code for implementing the security policy is localized to the MayI function rather than distributed among the member functions. Finally, the default “always OK” policy can be optimized so that there is no overhead at all associated with the mechanism (the “no play, no pay” option).

3.3 Authentication

The previous discussion finessed one point: who or what is the “I” that the MayI function grants access? Indeed, the request must first be authenticated to identify the principal that uttered it, and then authorized only if the principal has the right to perform the operation on the object. The principal behind the request could be human users, software programs, or compound identities such as delegations, roles and groups.

Authentication in Legion is aided by the use of Legion certificates. Recall that the certificate contains the object identifier of the responsible agent, and that the calling agent is identified in the method call.

In the general spirit of our approach, the authentication of the caller and caller’s context can be anything that the MayI function demands—and in sensitive cases, that is just as it should be. In most cases, however, “I” will be simply the CA, or the RA, or any subset of the two. Indeed, by analogy with familiar systems where “I” is the user, that subset may be just the RA.

Legion makes a specified level of effort to assure the authenticity of the certificate IDs; this effort should be adequate for most purposes. However, in the spirit of the second principle, we expect that MayI functions with extraordinary security concerns will code their own authentication protocols by, for example, calling back to the caller, and/or responsible agent. To make this possible, we require every Legion object to supply a special public member function, Iam, for authentication purposes. In the same principle as MayI, Iam could be optimized to NIL. Figure 1 shows a call from B.MayI back to A.Iam to verify A’s identity. The specific protocol used between MayI and Iam to
authenticate A’s identity is immaterial; if Iam satisfies MayI, then the call will proceed, else MayI will fail (and an exception will be raised).

Legion bases authentication on public-key cryptography in the default case. Knowledge of the private key is the proof of authenticity. In addition, a set of general authentication protocols will be provided as the system standard. Iam can choose to support all or none of them. Other more elaborate protocols could be negotiated between objects and made known to the Iam function. Objects unprepared to adequately authenticate themselves are ipso facto not to be trusted.

3.3.1 Login

The avenue via which Legion users authenticate themselves to Legion is the Login procedure. Login establishes the user’s identity as well as creating a responsible agent object for the user. The login procedure is therefore the building block for future authentication and delegation.

By the same design principle, Legion does not mandate a single “Login” mechanism. Currently, there is a login object that is invoked when a user first logs in. This login object engages in a login dialog with the user and, if satisfied, declares itself to be the responsible agent. Actually, any Legion object may declare itself to be the current responsible agent should it choose. It simply generates an additional certificate designating itself as the RA.

There are many advantages to why we shouldn’t make this login mechanism universal. For example, logging on to Legion at the University of Virginia may require only a simple password while Legion in the CIA might demand that users submit fingerprints or retinal scan information. Users can define their own login class with varying degrees of rigor in the login dialog, specific to their needs. The login mechanism can also be easily inherited or defaulted to some simple scheme.

How do we know that a particular login class (or RA) is to be trusted? We don’t, in general. The MayI function of another class need not believe the login! After interrogating the class of the responsible agent the MayI function may reject the call if the login is either insufficiently rigorous, or simply unknown to this MayI. As in the infamous “real world,” trust can only be earned.

3.4 Delegation

In all security models one must consider the question of rights propagation; can a principal hand all or some of its authority to another, and how can a
principal restrict its authority? For example, a user on a workstation may wish to delegate the “read” right on her files to the C compiler. The compiler can then access files on her behalf as long as the delegation still stands.

In Legion, an object can generate a new certificate to delegate rights, e.g. the user above could generate a certificate granting the bearer the “read” right and pass it to the compiler. If an intermediate object in the call chain wished to delegate rights contained in its current certificate, it could invoke the Delegate function on the RA to generate a more limited certificate.

Our philosophy is that delegation policy is a part of the discretionary policy that should be defined by the object itself. Indeed, delegation policies can be arbitrarily complex or lightweight. Classes that want to take extreme precautions against delegation may choose not to support delegation at all. Alternatively, users can write their own delegation functions or inherit appropriate ones from existing classes.

So far we have discussed three security-related functions: MayI, Iam and Delegate (we defer discussion of CanI to the next section). They are user-defined functions, which together, quite elegantly, form a guard or reference monitor upon which any discretionary policy can be defined. In addition, MayI, Iam and Delegate can be defaulted to NIL and hence will impose no overhead. And indeed, many classes will favor the default case for performance reasons. When these functions are non-NIL, they enforce user-definable policies rather than some global Legion-defined one. These functions can be as simple or as elaborate as the user feels necessary to achieve their comfort level—the “service slider” approach again.

4 Mandatory Policies

Mandatory policies, such as multi-level security, presume that the parties involved may be conspirators and impose some sort of check by a third party—usually “the system”—between caller and called objects. Generally this imposition is completely dynamic; every call is checked.

In the Legion context, of course, we eschew the idea of a system-wide policy. Thus we need a safe mechanism that interposes an arbitrary enforcer of an arbitrary policy between caller and called object. Interestingly, when combined with inheritance, the MayI function already discussed provides half the answer, albeit in a somewhat different way.

Imagine that a new mandatory security regime is to be created. An obvious consideration is that the enforcer, which we’ll call the “security agent” must
Fig. 2. Automatic invocation of CanI on outgoing calls.

know about all of the kinds of objects in its domain—it cannot enforce “no write down” if it doesn’t know what a “write” to a specific object is, for example. Thus we’ll begin with the presumption that a good security agent simply won’t allow calls on objects of unknown pedigree.

Given that, it is reasonable to presume that the security agent can derive subclasses for the objects that it does know about; in these subclasses the security agent can inherit a MayI function of its choosing—and specifically one that performs an outcall to the security agent to verify the validity of each inward call. In this case, we include both compile-time and run-time activities in the actions of the security agent. These may, in fact, be separate but cooperating entities. All and only the objects that are instances of these derived classes will be permitted in this security agent’s regime.

As noted above, this solves half the problem—the security agent is invoked whenever an object under its control is called. We need to add the symmetric capability for outward calls; thus we add a method CanI that, if non-null, is invoked by Legion whenever an object attempts to make a call on another object. Now, by deriving a class that defines both the MayI and CanI methods, the security agent can be ensured that it gets invoked on every call involving one of the objects under its control.

Figure 2 depicts the use of CanI in a method call. Again, Object A invokes B.foo, but the compiler has interposed code so that A.CanI is automatically invoked before the call leaves A. If CanI returns true, then the method call proceeds as in our earlier example.² If CanI returns false, an exception is raised.

Note that while the usual mechanism for enforcing mandatory policies is done completely at run time, the one we have described is partially a compile time (or link time) mechanism—that is, the time at which the MayI and CanI methods are bound into the subclass. Although this seems almost required by

² We omit the potential call to Iam to keep the picture legible.
the rejection of a single system-wide policy, it might raise concerns over the possibility of intentional corruption of the mechanism. This is a subtler topic than can be handled in detail here, but the reader may gain some comfort from the observation that we have inverted the usual (temporal) relation between defender and attacker. In the traditional scenario the defender of security puts out a system which the attackers then may analyze and attack at leisure. In our case, if the attack is to be mounted from within an object that the security agent has “wrapped” with its own Mayl and Canl functions, the attacker must put their code out first without knowledge of how it will be wrapped. In this case, the security agent has the advantage of examining the purported attacker’s code before deciding whether to allow it into its security regime.

5 Is There An Imposter In The House?

In a large distributed system such as we envision, it is impossible to prevent corruption of some computers. We must presume that someone will try to pose as a valid Legion system or object in order to gain access to, or tamper with other objects in an unauthorized way. That is why, in the final analysis, the most sensitive data should not be stored on a computer connected to any network, whether running Legion or not.

On the other hand, perhaps we can make the probability of such mischief sufficiently low and its cost sufficiently high to be acceptable for all but the most sensitive applications. We have formulated a number of principles that form a basis for our ongoing research. They are:

(i) Defense in depth: There won’t be a single silver bullet that “solves” the problem of rogue Legionnaires, so each of the following is intended as an independent mechanism. The chance that a rogue can defeat them all is at least lower than defeating any one separately.

(ii) Least Privilege: Legion will run with the least privilege possible on each host operating system. There are two points to this: first, it will reduce the probability that a remote user can damage the host, and second it is the manifestation of a more pervasive minimalist design philosophy.

(iii) No privilege hierarchy (compartmentalize): There must not be a general notion of something being “more privileged than” something else. Specifically Legion is not more privileged than the objects it supports, and it is completely natural to set up non-overlapping domains/policies. This precludes the notion of a “Legion root,” guaranteeing that no single entity can gain system-wide ultimate privileges.

(iv) Minimize functionality to minimize threats: The less one expects Legion to do, the harder it is to corrupt it into doing the wrong thing! Thus, for example we have moved a great deal of functionality into user-definable
objects-responsible agents and security agents were discussed here, but similar moves have been made for binding, scheduling, etc. This increases the control that an individual or organization has over their destiny.

(v) If it quacks like a Legion...: Legion is defined by its behavior, not its code. There are a number of security-related implications of this. First, it's possible for several entities to implement compatible Legion systems; this reduces the possibility of a primordial trojan horse; it also permits competing, guaranteed implementations. Second, it opens the possibility of dynamic behavioral checks—imagine a benign worm that periodically checks the behavior of a system that purports to be a Legion, for example.

(vi) Firewalls: It must be possible to restrict the machines on which an object is stored or is executed, and conversely restrict the objects that are stored or executed on a machine. Moreover, the mechanism that achieves this must not be part of Legion. It must be definable on a per class basis just like MayI and lam. (Of course, like the other security aspects of Legion objects, we expect that the majority of folks will simply inherit this mechanism from a class that they trust). Our prototype implementation uses user-level, per-class and per-host scheduling support to achieve this.

(vii) Punishment vs. Prevention: It will never be possible to prevent all misdeeds, but it may be possible to detect some of them and make public visible examples of them as a deterrent.

It should be noted that there is an informal, but important link between physical and computer security that is especially relevant to this discussion. Any individual or organization concerned with security must control the physical security of their own equipment; doing this increases the probability that the Legion code at their own site is valid. That, coupled with the security agent's ability to monitor every invocation, can be used to further increase an installation's confidence.

6 Recapping Some Options

The Legion security model shifts the emphasis from "system as enforcer" to user-definable policies-to give users responsibility for their own security-and to provide a model that makes both the conceptual cost and performance cost scale with the security needed. At one extreme, the blithely trusting need do nothing and the implementation can optimize away all the checking cost. At the other extreme, ultimate security suggests staying off the net altogether. Between these extremes lie several options, including:

- High security systems might be willing to accept the base Legion communication mechanism, but not even trust it to MayI or check certificates properly. For these we suggest embedding checks in each member function
and use physical security in conjunction with Legion.
- Somewhat less sensitive systems might trust the local “imposter checking” mechanisms to adequately ensure that MayI and certificate checking is done. However, they may still want to invoke MayI on each member function invocation to obtain a high degree of assurance. Such systems may execute authentication protocols with the responsible or calling agent to ensure that the remote Legion is not an imposter.
- In situations where security is not a primary concern, careful systems may feel that a lighter weight check, and not call back to the responsible or calling agent for authentication checks.

Our point is that there is a rich spectrum of options and costs; the user must choose the level at which they are sufficiently confident. Caveat emptor!

7 Related Work and CORBA Security

There is a rich body of research on security that spans a spectrum from the deeply theoretical to the eminently practical, most of which is relevant to this work. In particular, all of the work on cryptographic protocols [10] and on firewalls [1] is directly applicable to the development of Legion itself. Other work, such as that on the definition of access control models [4], on information flow policies [9] and on verification [7] will be more applicable to the development of MayI functions—which we will lean on as we develop a number of base classes from which users may inherit policies. In the same vein we will lean on existing technologies such as Kerberos [5], RSAREF [8], Sesame [6], etc.

We are not aware, however, of other work that has turned the problem inside out and placed the responsibility for security enforcement on the user/class-designer. The closest related work is in connection with CORBA; indeed many of the concerns we raised in the introduction are also cited in the OMG White Paper on Security [2]. A credo of that work, however, was “no research,” and so they retain the model of system as enforcer. Indeed an exemplar of our concern with this approach is where they talk about the trusted computing base (TCB):

“The TCB should be kept to a minimum, but is likely to contain operating system(s), communications software (though note that integrity and confidentiality of data in transit is often above this layer), ORB, object adapters, security services and other object services called by any of the above during a security relevant operation.”

It’s precisely this sort of very large “minimum” security perimeter that caused
us to wonder whether there was another way.

8 Technical Challenges and Future Work

There are many technical issues that we are unable to discuss in depth due to limited space. These issues pose challenging research questions and greatly affect the design of Legion security. For example,

- Encryption: Legion does not specify the use of any particular encryption algorithm, although our prototype implementation uses the RSAREF public-key encryption library. Applications concerned about the privacy of their communication should choose any encryption scheme they deem necessary. But that does raise one question, namely, how much protection of messages should be done by default? Should we send messages in the clear but digitally signed? Should we encrypt every message? What is the right performance-cost trade-off?

- Key Management: Public-key cryptography is the basis of authentication in Legion. However, Legion eschews any distinguished trusted objects. Name and key management thus need to be handled without any centralized component—no single key certification or distribution server. To make the key management simple, we define that every object's unique identifier be the public key of that object. A new key generation scheme is developed to do completely distributed, unique key generation. See [11] for more details on Legion key generation and management.

- Rogue Legionnaires: Will our users be comfortable enough to use Legion despite the fact that Legion itself could be corrupted? Do the principles we stated in fact help enough to make users confident? Can we describe the limits of the approach well enough for users to make well-informed decisions?

- Composition of security policies: In a multi-policy environment like Legion, what can we say when objects that enforce different policies are used together? In particular what happens when conflicting, even contradictory, security policies operate in conjunction? What can we do to effectively resolve conflict should it arises and help users evaluate combinatorial policies? How can we express policies to expedite evaluation and composition?

- There are a host of implementation issues related to other functional aspects of a real system—e.g., scheduling—that have security implications (how better to effect denial of service than to simply not schedule the task!).

We are testing out our ideas and starting to address these questions on a Legion prototype which is currently operational both within the University of Virginia and at our research partners (including NSF Supercomputer Centers, DoD MSRC, and DoE National Labs). As the overall Legion project proceeds, we will be able to develop the model in a more realistic context and scale.
We have built several base classes with security policies based on access control lists. We are in the process of incorporating Kerberos authentication into Legion. In cases where our simple login mechanism is deemed insufficient, we are working with our research partners to integrate more stringent mechanisms into Legion.

9 Conclusion

Building metasystems across the Internet will inevitably involve the interaction and cooperation of diverse agents with differing security and integrity requirements. There will be “bad actors” in this environment, just as in other facets of life. The problems faced by Legion-like systems will have to be solved in this context.

The model we have developed and implemented, we believe, is both a conceptually elegant and a robust solution to these problems. We believe it is fully distributed; it is extensible to new, initially unanticipated types of objects; it supports an indefinite number and range of policies and login mechanisms; it permits rational, user-defined trade-offs between security and performance. At the same time, we believe that it has an efficient implementation.

In the coming months, as we deploy Legion in a nationwide metasystem, we will test the “we believe” part of the last paragraph.

References


Wide-Area Computing: Resource Sharing on a Large Scale

Computing over wide-area networks has been largely ad hoc, but as needs increase, piecemeal solutions no longer make sense. Legion, a network-level operating system, was designed from scratch to target wide-area computing demands.

Consider almost any computing resource today—whether hardware, software, or data—and it will invariably be networked. Networking, especially wide-area networking, has created dramatic new possibilities for resource sharing. Cooperating contractors want selected access to each other’s enterprise systems. Researchers in geographically distant universities need to pool and analyze data from multisite experiments. Legacy codes on different computing platforms must exchange information to support data mining and other integrated applications.

These new possibilities depend on the ability to manage shared resources. But the sheer complexity of networked environments can turn this management problem into a nightmare. How do you share and manage resources yet maintain the autonomy of multiple administrative domains, hide the differences between incompatible computer architectures, communicate consistently as machines and network connections are lost, and respect overlapping security policies? The usual approach to these problems has been to deal with each situation individually. Piecemeal solutions are cobbled together from scripts, sockets, and various networking tools. If all goes well, a sophisticated programmer can build and maintain the application, but even then the implementation tends to be brittle and limited.

Resource management is traditionally an operating system problem, but large-scale collections of resources transcend classic operating system boundaries. What is needed is a wide-area operating system that can abstract over a complex set of resources and provide a high-level way to share and manage them over the network. To be effective, such a system must address the challenges posed by real end-user applications (see the sidebar “Challenges for a Wide-Area Operating System”). Scalability, security, and fault tolerance are just a few of the characteristics a viable solution must have.

Five years ago, we set out to design and build a wide-area operating system that would encompass all these challenges, allowing multiple organizations with diverse platforms to share and combine their resources. Our system, Legion (http://legion.virginia.edu), is now operational on hundreds of hosts across nine US sites, including the two NSF supercomputer centers (San Diego Supercomputer Center and National Center for Supercomputing Applications), two DoD supercomputer centers (Naval Oceanographic Office and Army...
Research Laboratory, NASA’s Aeronautical Research Center, and several universities. Users have ported a range of scientific applications to Legion in areas such as molecular biology, materials science, ocean and atmospheric science, electrical engineering, and computer science.

Legion is essentially a conduit between the end user and widely distributed collections of resources. Like a traditional operating system, it supports services such as resource management and a distributed file system. This operating system-style interface leverages application programmer experience and simplifies porting legacy applications to the Legion platform. However, unlike typical operating systems, Legion is layered on top of existing software services. It uses the existing operating systems, resource management tools, and security mechanisms at host sites to implement higher level system-wide services. Because of this middleware approach, Legion is able to reuse local services, and sites do not have to change familiar local software interfaces.

Legion is a component-based system: Distributed application components are represented as independent, active objects. This approach greatly simplifies the development of distributed applications and tools. Instead of facing the complexity of a wide range of distributed resources and service interfaces, the programmer works with the simple, uniform abstraction of distributed objects. Legion also supports a high level of site autonomy. Local sites can select and configure the components that represent their resources and services in any way they see fit, retaining complete control over local access control policies, resource quota mechanisms, and so on. Legion’s inherent flexibility is its greatest strength, and its most important defining characteristic.
HOW LEGION WORKS

With components that must interoperate in wide-area heterogeneous environments, Legion's fundamental object model resembles the Common Object Request Broker Architecture (CORBA). Programmers describe object interfaces in an interface description language (IDL) and then compile and link them to implementations in programming languages such as C++, Java, or Fortran. All system elements are objects and can communicate with one another regardless of location, heterogeneity, or implementation details. Within this object-based framework, Legion provides the services of a distributed operating system. Figure 1 outlines the structure of a sample Legion system.

The easiest way to understand how Legion works is to consider how it handles classic operating system tasks, which we consider in turn.

Representing and managing resources

As Figure 1 shows, local sites use Host and Vault objects to represent processors and storage, respectively. Using objects to represent resources has two primary benefits:

- Objects define a simple, consistent interface to Legion's resources. Hosts provide the uniform interface for creating objects (tasks); Vaults provide the uniform interface for allocating persistent storage. These interfaces provide a consistent view of system resources, even though local resource interfaces differ significantly in practice.
- The resource object model provides a tremendous degree of site autonomy. Applications (acting as resource clients) use the generic object interfaces for the resources they require. Resource providers are free to employ any desired implementation of the resource objects.

The second benefit is particularly significant. For example, if system administrators at a site want to enforce a specialized access control policy for their local hosts, they can extend or replace the basic Host implementation to enforce that policy. Similarly, some of the hosts in Legion systems may require access through a local queue management system such as Genias Software's Codine (http://www.genias.de) or IBM's LoadLeveler (http://www.rs6000.ibm.com/software/sp_products/loadlev.html). In these cases, resource providers simply use extended, queue-aware Host objects. Likewise, if a resource provider makes storage in a local file system available to Legion, yet wants to continue using local Unix-based accounting and quota tools, he can use a Vault object implementation that allocates storage under the appropriate

Challenges for a Wide-Area Operating System

At Boeing Company, designers use simulation to make ever more complex airframes at a manageable cost. Pratt & Whitney, which designs and supplies jet engines to Boeing, also relies heavily on simulation. When Boeing's engineers simulate an airframe's behavior, they need to know how the engine coupled to that airframe will perform under various conditions. However, Pratt & Whitney cannot release its proprietary engine simulations because of the significant intellectual property they encode. In an unwieldy information exchange process, Boeing engineers must ask Pratt & Whitney engineers to run their simulation at specified data points and then send them results by tape. Boeing engineers then combine the information with their own simulation data and modify it accordingly. The process iterates.

In a completely different domain, Harvard Medical School researches the causes and symptoms of multiple sclerosis. They need to get MRI scans from multiple partner institutions and to make a database of image-processed results available to the partners. As a first step, they want a tool that can automatically identify pertinent MRI scans at partner hospitals, securely move those scans over the Internet to Harvard, and then process them. The partners will provide very little administrative support for the tool.

In another medical setting, seven competing Dayton, Ohio, hospitals are working together to reduce costs. By sharing patient records and making them electronically available to emergency room physicians, they avoid expensive and time-consuming tests and can provide better care more quickly. Each hospital has its own legacy medical records system, IS personnel, and procedures that must somehow be merged. However, each also has databases and programs that cannot be shared.

Finally, climate modeling groups at San Diego Supercomputer Center, UCLA, and Lawrence Berkeley Laboratory want to couple a global atmospheric circulation model with a regional, mesoscale weather model. The coupled models would feed data to each other, creating more accurate and detailed combined results. The existing regional model runs only on a Cray T90, while the global model runs on a Cray T3E and is being migrated to the IBM SP. The applications need a way to coordinate and exchange data with one another at runtime, be scheduled to run simultaneously on separate supercomputers, and be easily controlled by a researcher at a single workstation.

These applications characterize the spirit of wide-area computing. Some of the requirements are unique, while others overlap. The applications also illustrate the following significant challenges, from managing complexity to implementing flexible, robust security.

Provide a high-level programming model

Complexity is the programmer's nemesis: A large-scale system can comprise several different architectures, tens of sites, hundreds of applications, and potentially thousands
local Unix user-id for each Legion client.

Legion provides configurable default implementations of the basic resource objects, so resource providers generally need not write any code to make their resources available. However, through object extension and replacement, Legion is flexible enough to support new local resource interfaces and policies as they arise.

**Managing tasks and objects**

Traditional operating systems must provide interfaces for starting new tasks and controlling their execution (suspend, resume, terminate, and so on). In Legion, the notion of a task or process corresponds closely to the Legion objects. Objects are the active computational entities within the system. Legion encapsulates object management functions in the Class Manager object type. Class Managers have three main functions:

- **They support a consistent interface for object management.** The Class Manager includes a natural set of object (or task) management operations, such as methods to create and destroy objects. Each Class Manager is responsible for a set of instances, which clients control through the Class Manager interface. Class Managers act as policymakers for their instances. For example, an object’s Class Manager determines which resources the object may use, and might enforce a policy that lets instances run only on a known set of trusted hosts.

- **They actively monitor their instances.** Class Managers query the status of their instances to detect failures and coordinate failure response (see Figure 1). In this role, Class Managers act as a distributed, agent-based fault-detection and response mechanism within Legion.

- **They support persistence.** All Legion objects can be persistent, existing arbitrarily beyond the life of their creating program. When an object is not in use, it can be deactivated: its state is saved to stable storage, and its containing process is deallocated (to conserve resources). This notion of object activation/deactivation is similar to traditional operating systems temporarily swapping out a job. To make object deactivation transparent to clients, the Class Manager acts as an automatic reactivation agent. If a client attempts to invoke a method on an inactive object, the Class Manager automatically reactivates it. Reactivation is thus as transparent as resuming swapped-out processes in traditional systems.

Decomposing object management responsibilities into an arbitrary number of Class Managers provides of hosts. Reducing and managing complexity is therefore critical. The object-oriented paradigm and object-based programming provide programmers and application designers with encapsulation features and tools for abstraction that reduce and compartmentalize complexity. We firmly believe that object-based techniques are key to constructing robust, wide-area systems.

These techniques are not enough, however. Composable, high-level services must replace low-level interfaces such as ssh and sockets in the programmer’s toolbox. Without such services, the complexity of distributed programming goes up dramatically, increasing both the skill set required to build applications and the fragility of the resulting software.

**Offer a single system image**

To combat the daunting number of distinct hosts and file systems, programmers need a single system image—the abstraction of a single machine and associated storage. For some, a “single system image” means a single shared address space; for others, the ability to run ps and get a list of all processes throughout the system. We define a single system image as a universal name space and management infrastructure for all objects of interest to the system and its users: files, processes, processors (hosts), storage, users, services, and so on. The names should be location independent (not contain any location information) and should be usable from anywhere in the system. Further, as programmers use resources to create their own objects, they should not be forced to explicitly place objects on a particular host or disk—the system should handle this. Thus, the programmer or user can specify or know an object’s location when necessary, but if this information is not relevant to his task, he can ignore it.

**Accommodate diverse administrative policies**

Most wide-area computing requires joining multiple organizations and administrative domains. To make this bridging easy, the system must accommodate a diverse set of local-use policies, access control policies, and computational cultures. For example, a site might insist that users authenticate via Kerberos before using its resources, or that users sign an “acceptable use policy” statement, or that each day from 1:00 p.m. to 6:00 p.m. no applications can be run that consume more than five CPU minutes. Extensibility and flexibility thus become essential—users must be able to readily extend and configure the system to satisfy local requirements.

**Manage heterogeneous resources**

Resource heterogeneity is a natural part of the distributed environment. Types of heterogeneity include processor, data format, configuration (how much memory and disk? which libraries are available on a host?), and operating system. If heterogeneity is not managed, individual users and programmers must deal with the complexity induced by all the possible permutations of hardware, operating system, and resources, a task that can rapidly overwhelm even the best programmers.
a natural distribution of the system's object management activities. Also, because Class Managers are extensible, replaceable objects, it is easy to customize the system's object management mechanisms. For example, to enable certain forms of failure resilience, some Legion classes use replication. The specialized Class Managers used for these object classes create and manage replicas transparently to clients.

Naming

Naming is a basic interface issue in operating system design. For example, operating systems typically define a name space for identifying processes (such as Unix PIDs), as well as a file system name space for identifying files and directories. Legion represents all entities—files, processors, storage devices, networks, users, and so on—as objects. These objects are identified by a three-level naming scheme. At the lowest level, each object is assigned an object address—a list of network addresses for the object. An object address might contain an IP address and port number, for example. Because Legion objects can migrate, object addresses change over time. Legion thus defines an intermediate layer of location-independent names called Legion object identifiers. LOIDs are globally unique identifiers that are assigned to objects when they are created. Because they are binary, system-assigned names, they are not convenient for users. To address this deficiency, Legion supports a hierarchical directory service, context space, which lets users assign arbitrary Unix-like string paths to objects.

The Legion naming mechanism reduces the complexity of designing distributed applications because it provides a single global name space for all system entities. A typical distributed environment supports separate name spaces for files, hosts, and processes; Legion, in contrast, supports the same global name space for all these as well as additional entities. The interface to this global name space is very easy to use; at the highest level (context space) the user manipulates names in the familiar form of Unix-style paths. Furthermore, Legion's scalable, replicated binding services make name translation automatic and efficient.

Providing an extensible file system

Traditional operating systems typically rely on a file system to manage and represent persistent storage. However, Legion's global name space and persistent object model make a separate file system unnecessary—in practice, the generalized persistent object space defined by Legion serves all the purposes of conventional file systems. In Legion's "file system," users see familiar elements such as paths, directories, and universally accessible files, but they also see arbitrary object types such as Hosts, Class Managers, and application tasks.

Grow without limits

The system must be able to add new hosts and resources over time. If the past has shown us anything, it is that the number of interconnected computational resources will only increase. Users and organizations do not want arbitrary limits on system size and capacity. System architectures must therefore be scalable and conform to the distributed systems principle that "the amount of service required of any single component of the system must not grow as the system grows." If an architecture does not conform, a component whose load (requests per second, for example) increases as the system expands will at some point become saturated, and performance will suffer.

Tolerate faults

Several years ago Leslie Lamport quipped, "A distributed system is one in which I cannot get something done because a machine I've never heard of is down." This indictment is driven by a simple fact: Without mechanisms to deal with failure, application availability is the product of component availability. In today's business climate, an unavailable application can easily cost thousands of dollars per minute. A wide-area system must therefore be resilient to failure and provide a failure and recovery model and associated services to applications developers, so they can write robust applications. The model must include notions of fault detection, fault propagation, and a set of useful failure mode assumptions.

Handle multilanguage and legacy applications

"I don't know what computer language they'll be using in a hundred years, but it will be called Fortran" was a popular refrain in the 1980s. Hundreds of millions of lines of legacy code today are written in languages as varied as Lisp, RPG, Cobol, assemblers, C/C++, Java, and (of course) Fortran. One thing is certain: Those codes will not be replaced overnight, and we will still want to be able to run them in distributed environments. The implication is that there must be a mechanism for supporting legacy code without modification, and it must be able to support a variety of programming languages. A wide-area computing environment must be language-neutral.

Implement flexible, robust security

Security includes a range of topics, including authentication (how do I know who you are?), access control (who can do what to each resource?), and data integrity (how can I make sure no one can read or modify my data in memory, on disk, or on the network?). Each of these issues is in the Boeing/Pratt & Whitney example. Clearly we must be able to provide high levels of security, but there is more to the problem. Security can be costly in performance, capability restriction, and other dimensions. Moreover, different users and organizations want to enforce very different policies. The challenge is to provide each user and organization with just the right mechanism and policy rules but still to allow different users and organizations to interact.
Because of this generality, Legion's object space is more flexible than conventional file systems. For example, users can customize individual files to better suit application-specific behaviors such as specialized file access patterns. Consider a file that contains a two-dimensional grid of data items. In a traditional file interface, accessing a single grid row or column might require multiple file operations. In Legion, users can define an extended file type to represent the 2D file object, with additional methods to permit row and column access.

**Enabling Interprocess Communication**

At the lowest level, Legion objects communicate via message passing to transmit method parameters and results. However, applications for wide-area systems need tools to reduce communication and to tolerate high latencies. To address these requirements, Legion supports macrodataflow, a variation of the traditional remote method invocation model.

Like other asynchronous remote method mechanisms, macrodataflow permits multiple concurrent invocations and lets users overlap remote methods and local computation. However, unlike other remote method protocols, macrodataflow forwards method results directly to data-dependent receivers. For example, if the caller does not directly use the result of a remote method, but needs it only as a parameter for future invocations, the caller will never receive the result. The macrodataflow protocol avoids the unnecessary act of communicating the result back to the caller, and instead forwards the message directly to the objects where it is needed.

Legion fully automates the macrodataflow protocol. Clients can specify and execute program graphs of interdependent remote method invocations using macrodataflow library interfaces, or via Legion-aware compilers such as the Mentat Programming Language Compiler. Similarly, object developers need not be aware of macrodataflow; Legion automatically matches incoming method parameters from multiple sources into complete method invocations, and forwards outgoing results directly to data-dependent recipients.

**Protecting resources and applications**

Wide-area operating systems must protect the security of both local resource providers and application users. Resource providers require that the wide-area operating system manage local resources in accordance with local policies. Application programmers

---

**How Legion Differs from ...**

**Common Object Request Broker Architecture**

CORBA 3.0 defines communication protocols, naming and binding mechanisms, invocation methods, persistence, and many other features and services essential for an object-based architecture. As such, its feature set and Legion's overlap in many areas.

The two architectures differ in their underlying emphasis, however. CORBA was initially a reaction to the software integration problem. Differences between software components in location, vendor, implementation language, or execution platform made building integrated applications difficult if not impossible. CORBA developers focused on enabling interoperability, and the architecture provides a common, object-based playing field where components can communicate and interact.

In contrast, Legion began with fundamental computing resources on a wide-area network—CPU, disk, data, and so on—and built an overarching framework for them. It emphasizes the ability to manage and reason about resources. The goal was to reconstruct a coherent computing environment with core operating system capabilities over a complex, heterogeneous environment. Thus, Legion can be used simply for its high-level operating system services to run, schedule, and manage legacy applications in a network, but it can mimic the CORBA standard for integrating applications. These two aspects combined give Legion its real power.

As CORBA evolves, some operating system-type services are starting to be defined for it. Scalability and other wide-area concerns are becoming more important. It remains to be seen how well its architecture will accommodate these changes.

**Globe**

The Globe project at Vrije University also shares many goals and attributes with Legion. Both occupy middleware roles, both support implementation flexibility, both have a single uniform object model and architecture, and both use objects to abstract implementation details. However, the object models of the two systems differ in many respects. Globe objects are passive and are physically distributed over potentially many resources, whereas Legion objects are active, independent entities. Because of this difference, Legion provides a more unified view of system components. Whereas in Globe there is a dichotomy between objects and processes, in Legion, objects are themselves the units of computation, providing the basis for distribution, scheduling, and resource management.

Globe and Legion both provide a platform for constructing applications based on interoperable components. But Legion differs significantly in also providing an integrated infrastructure for resource management. This hallmark of a wide-area operating system is essential for large-scale resource sharing.

**Globus**

The Globus project at Argonne National Laboratory and the University of Southern California has the same base of target environments, technical objectives, and target end users as Legion, and shares some of its design features. However, Globus and
must satisfy the security requirements of their applications.

Legion's security mechanisms are an integral part of its object architecture. The basic Legion security service is user-selectable data privacy and integrity within the Legion message-passing layer. Legion lets messages be fully encrypted for privacy, digested and signed for integrity checking, or sent in the clear if low performance overhead is an application priority. Cryptographic services in Legion are based on the RSA public key system (http://www.rsa.com). To protect against certain kinds of public key tampering, objects encode their RSA public keys directly into their LOIDs. Simply by knowing an object's LOID, a client can communicate securely with that object.

In any operating system, access control and resource protection are central issues. In Legion, all resources are represented by objects, so access control and resource protection are specified entirely at the object level. Invoked objects autonomously enforce access control invocation by invocation, using a mandatory internal method called MayI. When a method invocation arrives at an object, it is first processed by the object's MayI method, which can enforce an arbitrary access control policy. Typically, MayI makes access control decisions on the basis of credentials passed along with method parameters. Credentials consist of a free-form set of rights signed by a responsible client. The default MayI implementation is based on user-configurable access control lists, including the notion of groups.

In addition to access control mechanisms, operating systems must define mechanisms for user identity and authentication. Users (like all other Legion entities) are represented by objects, which are assigned unique LOIDs. The user's LOID contains his public key, but the user keeps his private key safe through arbitrary local means, such as a smart card. Trusted Legion programs executed by the user (the Legion login shell, for example) rely on the user's private key to sign appropriate credentials for outgoing methods. These credentials form the basis for authenticating the user and are typically used in conjunction with per-object access control lists to enforce user access control.

APPLICATIONS OF LEGION

Legion's services can accommodate a variety of domains and platforms. Two current applications illustrate its flexibility in supporting distributed enterprise computing.

Legion have fundamentally different high-level objectives. Globus provides a basic set of services that lets users write applications for a wide-area environment. Working components become part of a composite distributed computing toolkit. Legion, in contrast, strives to reduce complexity and to provide the programmer with a single view of the underlying resources, so it builds higher level system functionality on top of a single unified object model.

The Globus approach has several strong points. One is that it takes great advantage of code reuse and builds on user knowledge of familiar tools and work environments. This approach also has several drawbacks. As the number of services grows, the lack of a common programming interface and model becomes a significant burden. By providing a common object programming model for all services, Legion permits users and tool builders to combine the many services available in the wide-area operating system: schedulers, VOs services, application components, and so on. For example, users can run the same access control tools to configure security for files and for hosts. We believe the long-term advantages of basing a system on a cohesive, comprehensive, and extensible design outweigh the short-term advantages of evolutionary composition of existing services.

The Web

The Web is not a single entity whose characteristics can be isolated and analyzed. Rather, it is a broad collection of applications, protocols, and libraries focused on content delivery to end users running browsers. Advances in Web browser interfaces and functionality have driven the Web revolution, transforming it from an elitist tool to an omnipresent phenomenon. Given that the Web is most users' primary experience with distributed computing, it is important to define its role in wide-area computing.

The Web in its current form clearly does not constitute a wide-area operating system. Basic operating system issues, such as resource management and task scheduling, are simply not part of the Web's structure. This is not an indictment of the Web, but a recognition of its true strength as a remote access medium for distributed content and a ubiquitous interface technology for accessing distributed applications. As such, the Web is the perfect front end, or interface, to applications running in wide-area operating systems such as Legion. Application interfaces can be written in Java or they may use HTML and the Common Gateway Interface (CGI). They can communicate with back-end applications using either native socket protocols, HTTP, or higher level interfaces provided by the wide-area operating system. Viewed this way, the Web and wide-area operating systems such as Legion are complementary. For many users, the Web provides the most natural window into the Legion universe.

References


May 1999
Figure 2. The MRI data collection system in development for Harvard Medical School. The components of the MRI data collection application run on central servers at Harvard and on front-end computers at the MRI centers.

MRI data collection

The MRI data collection system in development for Harvard Medical School (see first example in the sidebar “Challenges for a Wide-Area Operating System") is a good illustration of an application structure that fits well with Legion's services. The components of the MRI data collection application run on central servers at Harvard and on front-end computers at the MRI centers. Figure 2 shows the architecture. Each leaf node has an MRI collection object (blue) that scans the local disk for specially tagged MRI images that the scanner has dumped. The MRI collection object copies these images into its persistent data space so that they will not be deleted when the scanner’s "dumping directory" is automatically cleaned up. Periodically, the MRI collection object calls the image processing object at Harvard (red) to upload the data in encrypted form, authenticating itself by including appropriately signed certificates in the method invocations. When it receives a complete batch of scans, the image processing object starts an image-processing pipeline, which consists of objects automatically scheduled onto local compute servers. The final stage of the processing pipeline inserts the results in the project's image database.

When a leaf node is rebooted, the node's Host object (yellow) starts automatically and registers with its manager (green) in the larger Legion net. The Class Manager object (pink) for the MRI collection component detects, via polling of the green Host object Class Manager, that the node is up and requests a restart of the blue MRI collection object for that node. The yellow Host object on the node handles the request, detecting simultaneously if the MRI collection object has been upgraded and, if so, downloading the new executable automatically. As it comes up, the MRI collection object recovers its state, which may include as-yet-untransmitted MRI scans.

Both the Host object and MRI collection object Class Managers have replicated persistent state. If the Class Manager goes down, its own higher-order Class Manager will detect the loss and restart it using the replica. This detection and restart behavior recurses up a tree of metmanagers (typically only one or two levels) to the root Legion manager object, which has a hot spare.

The Class Manager, Host, and other objects in the system are all configured with strict access control. Calls to various objects must present credentials to gain authorization. The MRI collection application and its Legion infrastructure are owned and accessible only by a small set of Legion users at Harvard. These users can centrally monitor and configure the system using Legion tools that provide views of all the hosts, objects, and so forth that are running or down.

Climate modeling

Climate modeling has progressed beyond basic atmospheric simulations to include multiple aspects of the earth system, such as full-depth ocean models, high-resolution land-surface models, sea ice models, and chemistry models. Typically, these models come from different research groups at a variety of institutions, are written in different languages, and require different resources. As described in the fourth example in the sidebar “Challenges for a Wide-Area Operating System,” coupled applications composed from existing models require the ability to coordinate existing components and to manage combined resources.

Legion's ability to combine and add value to existing components to create more complex applications fits nicely with this application. To construct the coupled climate model system, developers use the existing simulations as implementations for two new Legion object types: Global Model and Mesoscale Model. In doing so, they modify the simulations to enable linkage to a Legion object interface (described in IDL), and modify the I/O calls in the models to use Legion...
file objects in place of the local file system. Each new model object supports a method to request the execution of a simulation time interval. Coordination and coupling of the model objects is accomplished through the use of a Legion Coupler object. This object also transforms data from each model into the format required by the other (for example, the models employ geographic grids that differ by an order of magnitude in resolution).

Legion also satisfies this application's requirements for managing resources. For example, application developers can configure the Class Manager for the Global Model object to know that a Cray T3E is required for this object type. When the model becomes available on the IBM SP, they can reconfigure the Class Manager with a single command to account for this new resource selection possibility. When a user wants to run the complete coupled simulation, a standard Legion component—the Scheduler object—coordinates the acquisition of all needed resources (such as a T3E or SP to run the global model, a T90 to run the mesoscale model, and a workstation to host the Coupler object). Regardless of the resources selected, Legion automatically takes care of installing the needed application components at the target sites, and it uses the appropriate interfaces for the local site's task and storage allocation.

We are continuing to develop higher level services in Legion as we acquire more information from applications. For example, broad classes of applications can profit from similar fault-response techniques. To address this need, we are designing drop-in fault tolerance modules based on the existing detection and reporting infrastructure.

We also plan to develop new application tools, such as an integrated Legion debugger, and to port application toolkits such as Netsolve (http://www.cs.utk.edu/netsolve). These efforts are guided by our close collaborations with an expanding set of applications groups, such as the Harvard Medical School and the climate modeling groups mentioned earlier. Finally, we are actively engaged in commercializing the Legion platform for use in Internet and enterprise settings. For more information, visit the Legion site (http://legion.virginia.edu).

Acknowledgments

We thank Charles Guttman of the Department of Radiology, Harvard Medical School, for the MRI example, and Greg Follen of NASA's Lewis Research Center for briefing us on Boeing and Pratt & Whitney. We also thank Sarah Wells for her assistance.

This work was partially supported by DARPA (Navy) contract #N66001-96-C-8327, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), and DOE D459000-16-SC.

References


Andrew Grimshaw is an associate professor of computer science at the University of Virginia, where his research interests include metasystems, high-performance parallel computing, heterogeneous parallel computing, compilers for parallel systems, and operating systems. He is the chief designer and architect of Mentat and Legion. He received a Ph.D in computer science from the University of Illinois at Urbana-Champaign.

Adam Ferrari is a research scientist with the Legion project in the Department of Computer Science at the University of Virginia. His research interests include high-performance distributed computing, operating systems, metacomputing, and computer security. He received an MS from Emory University and a PhD from the University of Virginia, both in computer science. He is a member of the IEEE Computer Society and the ACM.

Frederick Knabe is a senior research scientist in the Department of Computer Science at the University of Virginia. His research interests include wide-area computing, computer security, and software risks. He received a PhD in computer science from Carnegie Mellon University.

Marty Humphrey is a research assistant professor of computer science at the University of Virginia. His research interests include real-time operating systems, real-time scheduling, distributed computing, and metacomputing. He received a PhD in computer science from the University of Massachusetts and is a member of the IEEE.

Contact the authors at {grimshaw, ferrari, knabe, humphrey}@virginia.edu.

May 1999
Metasystems

How they create the illusion of a giant desktop computational environment—transparent, distributed, shared, secure, fault-tolerant.

Andrew Grimshaw, Adam Ferrari, Greg Lindahl, and Katherine Holcomb

Metasystems give users the illusion that the files, databases, computers, and external devices they can reach over a network constitute one giant transparent computational environment. With a metasystem, users can share files, computational objects, databases, and instruments. They need not decide where to execute their programs nor copy the necessary binaries and data files; the metasystem does these jobs. And new classes of applications—meta-applications—are available through the emerging infrastructure, further increasing users’ efficiency and productivity.

Here we explore the potential of metasystems and the technical problems that have to be solved to realize them, highlighting five metasystem uses: shared persistent object spaces; transparent remote execution; wide-area queueing systems; wide-area parallel processing; and meta-applications. We also cover an example of a meta-application being developed by NPACI—a coupled ocean-atmosphere-weather model.

A metasystem’s building blocks are geographically separated resources (people, hosts, instruments, databases) connected by one or more high-speed interconnections. An essential software layer, often called middleware, transforms a collection of independent hosts into a single, coherent virtual machine, giving the user the power of a unified environment. To the user, this virtual machine is a single entity that executes applications, schedules application components, detects and recovers from faults, provides protection and security to users and resource owners, and brings users together through enhanced support for collaboration and sharing.

Why don’t we use metasystems? The fundamental difficulty is lack of software—specifically, an inadequate conceptual model for metasystem software design. Faced with accelerating changes in hardware and networking, the computing community has sought to stretch the existing paradigm for sharing resources over a network—interacting autonomous hosts—to a level that exceeds its abilities. The result is a collection of partial solutions, some good in isolation but lacking coherence and scalability, making development of even a single wide-area application at best demanding and at worst nearly impossible.

The challenge to the computer science community is to provide a solid, integrated middleware foundation on which to build wide-area applications. NPACI, in its role as agent for high-end computing, has an additional challenge—to design and deploy metasystems technology providing the high performance needed for the most demanding scientific applications.

As envisioned, the NPACI metasystem contains thousands of hosts and petabytes of data. Users will have the illusion of a very powerful computer on
Land cover of Africa in Plate Carrée projection.
(Courtesy, Joseph J. J. and John Townshend, University of Maryland.)
their desks that can manipulate objects representing data resources, such as digital libraries or video streams; applications, such as teleconferencing or physical simulations; and physical devices, such as cameras, telescopes, and linear accelerators. The objects being manipulated may be shared with other users, allowing construction of shared virtual workspaces.

This metasystem will support the illusion of a single machine by transparently scheduling application components on processors; managing data migration, caching, transfer, and coercion, or the masking of data-format differences between systems; detecting and managing faults; and ensuring that users' data and physical resources are protected. Moreover, the technology used by the metasystem will have to scale appropriately.

The potential benefits of such a metasystem to the scientific community are enormous, including:

- More effective collaboration, achieved by putting coworkers in the same virtual workplace
- Higher application performance, owing to parallel execution and exploitation of off-site resources
- Improved access to data and computational resources
- Improved researcher and user productivity, resulting from more effective collaboration and better application performance
- A considerably simpler programming environment for applications programmers

There are many ways to use a metasystem, ranging from relatively simple ones to intricate implementations exploiting its abilities to solve currently impossible problems. We sketch five wide-ranging uses to illustrate some of the possibilities.

Shared persistent object (file) space. The simplest service a metasystem provides is location transparency, whereby users gain authorized access to entities without knowing where they reside. For file access, this well-understood type of service is often called a "distributed file system." Well-known distributed file systems include NFS [9] and Andrew [10].

In shared object spaces, instead of just sharing files, all entries (files as well as executing tasks) can be named and shared among authorized users. This merging of "files" and "objects" is driven by the observation that files are merely a special kind of object that happens to live on a disk, so files are slower to access but persist when the computer is turned off. In a shared object space, a file object is a typed object with an interface that exports standard file operations, such as read, write, and seek. The interface can also define an object's properties, including its persistence, fault, synchronization, and performance characteristics. Files do not have to be of the same type.

Beyond basic sequential files, class-based persistent objects offer a range of opportunities, including:

- Application-specific "file" interfaces. For example, instead of just read, write, and seek, a 2D-array-file may have additional functions, such as read_column, read_row, and read_sub_array. The advantage is the ability to interact with the file system in

Above: Where damage starts; opposite: Stress in a composite.
(Courtesy, Kumar Vemaganti, University of Texas, Austin.)
The challenge is to provide a solid, integrated middleware foundation on which to build wide-area applications.

application terms rather than as arbitrarily linearized streams of data.

- **User specification of caching and prefetch strategies.** Users can exploit application domain knowledge about access patterns and locality to tailor the caching and prefetch strategies to the application.

- **Asynchronous I/O.** Combined with a parallel object implementation, I/O can be performed concurrently with application computation. The combination of user specification of prefetching and asynchronous I/O can drastically reduce I/O delays.

- **Multiple outstanding I/O requests.** Again, if combined with a parallel object implementation, the application can request remote data long before it is actually needed. The application can then compute while I/O is being processed. By the time data is needed, it may already be available, resulting in almost zero latency.

- **Data format heterogeneity.** Persistent classes may be constructed to hide data format heterogeneity, automatically translating data as it is read or written.

- **Active simulations.** In addition to passive files, persistent objects can also be active entities. For example, a factory simulation can proceed at a specified pace (such as wall clock time) and be accessed (read) by other objects. The factory simulation may itself use and manipulate other objects.

*Transparent remote execution.* A slightly more complex service is transparent remote execution. Consider the following hypothetical situation: A user is working on a computationally complex sequential code, perhaps written in C or Fortran. After compiling and linking the application, the user has to decide where to execute the code, choosing to run it on the local workstation (assuming adequate space and speed are available), on a local high-performance machine, or at a remote supercomputer center. The choices involve many trade-offs. First, there is the scheduling issue: Which choice will result in the fastest turnaround? This decision is especially crucial when the user has accounts at multiple centers. How is the user to know which choice is likely to result in the best performance without manually checking each? Next, there are the inconveniences of using remote resources. Data and executable binaries may first have to be copied physically to a remote center and the results copied back for analysis and display. Finally, there are the administrative difficulties of
Acquiring and maintaining multiple accounts at multiple sites.

In a metasystem, the user can simply initiate the application at the command line. The underlying system selects an appropriate host from among those the user is authorized to use, transfers binaries if necessary, and begins execution. Data is transparently accessed from the shared persistent object space, and the results are stored in a shared persistent object space for later use.

**Wide-area queueing system.** Queueing systems are part of everyday life at production supercomputer centers. The idea is simple: Rather than interactively starting a job, the user submits a description of the job to a program (the queueing system), and that program schedules the job to execute at some point in the future. The purpose of the queueing system is to optimize an objective function, such as system utilization, minimum average delay, or a priority scheme. There are more than 25 different queueing systems in use today, including NQS, PBS, Condor, and Codine [8].

Just as queueing systems can be used to manage homogeneous resources at a particular site, metasystem queueing systems manage heterogeneous resources at multiple sites. This feature enables submission of a job from any site and, subject to access control, automatic scheduling of the job somewhere in the metasystem. Further, if binaries for the application are available for multiple architectures, the queueing system has a choice of platforms on which to schedule the job.

The advantage to the user over the current state of practice in distributed systems is significant. Users with multiple accounts at different sites often have to shop around, looking for the shortest queue in which to run a job, and may need to copy data between remote sites. This manual metascheduling process consumes the most precious resource—user time—which is better spent doing science.

**Wide-area parallel processing.** Another opportunity offered by metasystems is to connect multiple resources for the purpose of executing a single application, enabling scientists to tackle mission-critical problems on a much larger scale than would otherwise be possible. However, not all problems can exploit this capability, since the application must be latency-tolerant (it is, after all, at least 30 msec from California to Virginia). Examples of applications in this category include "bag-of-tasks" problems and many regular finite-difference methods.

Consider typical bag-of-tasks problems, such as parameter-space studies, Monte Carlo computations, and other simple data-parallel problems. In a parameter-space study, the same program is repeatedly executed with slightly different parameters. (The program may be sequential or parallel.) In a Monte Carlo simulation, many random trials are performed; the trials may be distributed easily among a large number of workers and the results gathered.

![Figure 1. Two geographically separated distributed-memory MPPs connected via high-speed link and a user at a visualization station at a third site. The hosts can be the same or have different processors and interconnection networks.](image)

In a simple data-parallel problem, the same operation is performed on each of many different data points, and the computation at each data point is independent of those for all other data points.

These bag-of-tasks problems are well suited to metasystems because the related applications are highly latency tolerant. While one computation is being performed, the results of the previous computation can be sent back to the caller and the parameters for the next computation sent to the caller. Further, the order of execution is unimportant, completely eliminating the need for synchronization among workers, thus simplifying load-balancing significantly. The computations can also be spread easily to a large number of sites because the compu-
rations do not interact in any way.

A more complex class of problems might comprise 2D finite-difference applications, such as time-explicit hydrodynamics. Suppose we wish to use two distributed-memory massively parallel processors (MPPs) and a visualization system at different sites, all connected by a fast communication pipe running at 100Mb–1Gb (see Figure 1). Further suppose that the first host has twice as many processors as the second one has. Balancing the load requires decomposition of the problem in such a way that the first host has twice as much of the data as the second.

Given the decomposition in Figure 2 and information on the size of the mesh points, we can easily compute the bandwidth requirements of the communications channel to determine whether the total latency is acceptable.

**Meta-applications**

The most challenging class of applications is the meta-application—a multicomponent application whose components may have been previously executed as standalone computations.

The generic example in Figure 3 shows three previously standalone applications connected to form a larger, single application. Each of the components has hardware affinities. Component 1 is a fine-grain data-parallel code requiring a tightly coupled MPP machine, such as an IBM SP2; Component 2 is a coarse-grain data-parallel code that runs efficiently on an inexpensive "pile of PCs" machine; and Component 3 is a vectorizable code that runs best on a vector machine, such as a Cray T90. Component 1 also has a very large database physically located at Site 1. Component 1 is a shared-memory Fortran code, Component 2 is written in C using the Message Passing Interface (MPI) [7], and Component 3 is written in Cray Fortran.

The underlying software layer has to take into account component characteristics, distributed databases, scheduling, visualization, fault tolerance, and security.

**Component characteristics.** Meta-application components are often written and maintained by geographically separate research groups. The components may be written in different languages or may use different parallel dialects of the same language (such as Fortran components using MPI, Parallel Virtual Machine, or PVM, [4] and High Performance Fortran, or HPF). Since the components often represent valuable intellectual property for their owners, these owners may want to protect their code by keeping it in house. Similarly, software licenses may be available only on a subset of the available hosts. The metasystem has to facilitate intercomponent communication when the components are in different languages, and it has to migrate binaries transparently from site to site.

**Geographically distributed databases.** Data is usually physically close to the research group that collects and maintains it. So before execution can begin, the data often has to be copied to a single location. The challenge to the metasystem is to determine when it makes sense to move the computation to the data or vice versa.

**Scheduling.** Scheduling meta-applications is another significant challenge. The general scheduling problem of mapping an arbitrary task graph onto more than three processors is known to be an extremely difficult problem. Application-class-specific heuristics that exploit knowledge about the application (the "shape" of the graph) have to be used [1]. Even so, the scheduling problem is quite difficult.

Consider scheduling a simple meta-application with three data-parallel components onto a single distributed-memory MPP (see Figure 4). First, the number of processors allocated to each component must ensure that each component progresses at the same rate; efficiency may require that each component use a different number of processors. Second, the component tasks have to be mapped to the
Metacomputing is indispensable for the success of large, complex simulations.

Processors in such a way as to reduce the communication load among them; random placement can lead to communication bottlenecks. Finally, the computational requirements of the components may vary over time, requiring dynamic repartitioning of the available resources.

Now suppose that instead of a fixed number of processors on an MPP, we have to choose among a large number of diverse systems, each connected to the others by networks of widely varying capability. The scheduling problem is certainly a tough one.

Visualization. During computation, users may want to see what is happening, at the global scale and at smaller scales. Further, they may wish to look inside individual models at particular points and perhaps adjust the computation by modifying some parameters.

Fault tolerance. As the number of hosts and processors participating in the computation increases, the probability of a failure increases and the mean time to failure decreases. When the mean time to failure is less than the completion time for a portion of the application, the probability that the application will finish successfully is low. Fault tolerance is a necessity under these circumstances.

Security. Security encompasses a range of issues, including authentication, access control, and encryption. Different users have different requirements. Suppose, though, that a meta-application is using databases that are distributed geographically. These databases must be protected against unauthorized access and update, and the distributed components' results must be protected from tampering or destruction. Finally, data transferred among components may be proprietary and need protection against tampering and snooping.

A meta-application, in sum, may be composed of multiple models running at different scales, written by different research groups using different languages or parallel processing tools, using proprietary, geographically distributed databases, on faulty hardware. Such applications require a system that cleanly and easily supports interoperability among components, as well as the plug-and-play incorporation of components into a running program and the scheduling of components onto processing resources (within a large MPP and between hosts). Further, the system must support transparent, secure, and efficient access to remote databases while providing robust integrated visualization.

Metacomputing at NPACI
Metacomputing is one of NPACI's four enabling technology thrust areas. (The other three are also covered in this issue.) The NPACI metasystems plan consists of three broad components: rapid hardening and deployment of leading-edge metasystems software, integration of software and tools developed in the other technology thrust areas, and close collaboration among metasystem thrust teams and pioneering applications teams.

NPACI does not support basic research per se in metasystems or in the other thrust areas. Rather, it supports technology transfer from existing funded research projects to the production computing environment. However, research software is rarely immediately ready for production use. Before such software can be used, it must undergo aggressive testing and elimination of bugs. And the development of documentation, sample codes, tutorials, and other training materials is needed to make the software accessible to mainstream scientists.

The NPACI hardening effort consists of four metasystems projects: AppLeS [1], Globus [2, 3], Legion [5, 6], and the Network Weather Service (NWS) [11].

AppLeS (Application-Level Scheduler). Directed by Fran Berman of the University of California, San Diego, this project focuses on developing application schedulers for individual metasystem applications. Each AppLeS agent couples with its application to develop and deploy a customized application schedule that can respond to the dynamic and heterogeneous performance characteristics of the underlying metasystem. AppLeS uses...
application-specific information, dynamic system information (provided by NWS), and predictive models to develop a performance-efficient, time-dependent, and load-dependent schedule.

AppLeS applications can either target networked environments with no additional infrastructure or use the infrastructure provided by other NPACI metasystem projects. AppLeS agents are being developed and deployed for such NPACI applications as protein-docking codes, a synthetic-aperture radar atlas (SARA), and tomography codes.

Globus. Globus is an infrastructure toolkit developed by Carl Kesselman of the University of Southern California and Ian Foster of Argonne National Laboratory with a broad group of collaborators. It provides services in the areas of communication, resource location/allocation, security, information, and data access. Globus has withstood many tests, including a recent one involving battlefield simulations distributed across more than 30 machines and representing the independent activity of more than 100,000 tanks, trucks, and other units.

For each of Globus's five service areas, there is a defined application programming interface (API) and a set of implementation notes explaining the semantics and the implementation. For example, Globus communication services (the Nexus communication library) provide unicast and multicast message delivery services. Globus information services (through its Metacomputing Directory Service) provide a uniform mechanism for obtaining real-time information about system structure and status. Globus resource location and allocation services provide mechanisms for expressing application resource requirements, for identifying resources meeting these requirements, for scheduling resources once they are located, and for initiating and managing computation on these resources.

Legion. Legion is a reflective object-based metasystem being developed by the author Andrew Grimshaw and his group at the University of Virginia. Legion aims to provide a single, coherent, virtual machine addressing scalability, programming ease, heterogeneity, fault tolerance, security for users and resource providers, site autonomy, multilanguage support, and interoperability.

Legion supports a range of services and programming tools, including a shared, secure persistent object space supporting authentication and arbitrary access control lists for objects; parallel programming tools, such as MPI, PVM, the Mentat programming language, and Fortran dataflow; complete site autonomy (a site can decide which users can run which binaries on the site); a complete set of authenticat-

ion, encryption, and access control services; and user-defined scheduling and resource management.

Network Weather Service. NWS, being developed by Richard Wolski of the University of California, San Diego, dynamically forecasts the performance that various network and computational resources can deliver over a given time interval. It is being used to track and monitor the performance of end-to-end very high-speed Backbone Network Service (sponsored by the NSF). It operates a set of sensors (network and CPU monitors) from which it gathers readings of current conditions. It then uses sophisticated numerical models to generate forecasts of what the conditions will be for a given time frame. The supported forecasting methods treat successive measurements from each monitor as a time series. These methods fall into three categories: mean-based, median-based, and autoregressive methods.

NWS tracks the accuracy of all predictors to generate a forecast, using prediction error as an accuracy measure and selecting the predictor exhibiting the lowest cumulative error measure at any given moment. In this way, NWS automatically identifies the best forecasting technique for any given resource.

Metasystems represent the glue that unites all of the other NPACI projects. Over time, the other enabling technologies and the applications prototypes will exist in a metacomputing environment. For example, KeLP coupling and interpolation and the NetSolve numerical object environment (described in J. Saltz's article in this section) are already being integrated with Legion and Globus. And application integration is moving ahead. An example is the collaboration between the Legion team and an earth science application originating at the University of California at Los Angeles.

Coupled Ocean-Atmosphere Modeling

Global climate modeling is an example of a field that can benefit from metacomputing. Climate modeling has progressed beyond atmospheric simulations to include multiple aspects of the Earth system, such as full-depth world ocean models, high-resolution land-surface models, sea-ice models, and chemistry models. Each component model generally requires a different resolution in space and time. Some components, such as a full model of atmospheric chemistry, can add as much as an order of magnitude to the total processing time.

The models usually originate with different research groups around the world and are written in different languages. An additional complication is that many models have parallel implementations, often using different parallel toolkits. With existing
ols, coupling these models would be tedious and
error-prone at best; not only do the data formats and
couplings have to be implemented by hand, but (for
the sake of performance) models have to be sched-
ulated appropriately on the available resources. In
tion, databases required for the component
models are usually geographically distributed, and if
there is a huge quantity of data, the data needed for
resolving the interactions models may not fit on
the machine on which the simulation is to be
performed.

The metacomputing environment will enable the
models to run on different, perhaps physically
remote, machines, as long as fault tolerance and
security are addressed. Security problems are exacer-
ated by the operation of a network of far-flung
resources, because the data and the results of the cou-
pled models are the intellectual property of the
researchers who compiled the input data and pro-
duced the results. A final issue is visualization. The
larger and more complex the simulation, the more
critical is the need for visualization—for humans to
be able to digest the enormous amount of data gen-
erated by high-resolution scientific models.

The Earth System Model (ESM) being developed
by NPARC partners R. Carlos Mechoso and Richard
Turco at UCLA is a coupled atmosphere and ocean
model that can include atmospheric and ocean
chemistry to study such problems as the effect on
climate of chlorofluorocarbons and aerosols. Work is
in progress to couple a regional mesoscale model
developed at the Lawrence Berkeley Laboratory to the
ESM to enable researchers to focus on the effect
of global climate phenomena, especially El Niño
and La Niña, on California weather. Future plans call for
even smaller scales, down to individual bays and
estuaries along the California coast.

The global atmospheric model is currently being
run at a resolution of approximately 1.25 degrees of
latitude × 1 degree of longitude, with 50 vertical
atmospheric levels; such a model can resolve large
typhoons and weather systems. Although this is state-of-
the-art climate modeling, it is still too coarse a grid
to resolve details of smaller individual weather
systems. The mesoscale model runs on a resolution
20km over a region encompassing mainly Califor-
nia and Nevada.

A full system of this nature would be nearly
impossible to run on existing individual supercom-
puters, even those that are massively parallel. ESM
alone can consume most of the cycles of an MPP
Cray T3E, while the current implementation of the
mesoscale model runs most efficiently on machines
like a Cray C90 or T90. Therefore, the Legion
research team is developing a Legion-based cou-
lping module to manage the data exchange and syn-
chronization among various models running on
different hardware. The researchers in charge of the
component models would need to add only a few
communication calls to their codes; no major
changes or rewriting would be required—a major
advantage when dealing with complex scientific
models.

ESM communicates internally via a native paral-
lel toolkit, such as MPI or PVM. There is no impedi-
ment to using Legion in this situation, but we have
decided not to alter the current implementation.
The ESM master process acts as the intermediary
between ESM and the Legion coupler. The Legion
coupler receives data from ESM, performs the
appropriate transformations from the ESM grid to
the regional grid, and sends the transformed data to
the regional model. From the perspective of the
models, this is little different from writing to or
reading from a file. Once full coupling is achieved
between ESM and the regional model, a dynamic
interface is essential, but much of the communi-
cation effort has already been completed by way of a
unidirectional interface. Moreover, maintaining a
separate "manager" means that resolutions can be
changed and models interchanged or added with lit-
tle difficulty.

This coupling project illustrates clearly the
advantage of a metacomputing approach for a
demanding, important scientific simulation. The
models run optimally on different architectures but
must communicate at regular intervals to calibrate
one another. Each model has to use a large store of
surface data that might be in a format incompatible
with the architectures on which the other models are
being run. The lengthy runs required for climate
simulation make fault tolerance imperative, espe-
cially in a coupled environment in which one model
could hang hopelessly if another fails. Metacom-
puting is thus indispensable for the success of large,
complex simulations.

Computational Productivity
Metasystems technology is maturing. Three years
going to the Supercomputing'95 conference, the T-Way
was a one-time demonstration of a large number of
applications that had been constructed in a rather ad
hoc manner. Today, metasystems testbeds are opera-
tional on an almost full-time basis. As the technol-
ogy matures further and becomes hardened enough
for use in production systems, we can expect a sig-
nificant increase in the computational productivity
of the sciences.
REFERENCES

Andrew Grimshaw (grimshaw@cs.virginia.edu) is an associate professor of computer science at the University of Virginia.

Adam Ferrari (ferrari@cs.virginia.edu) is a research scientist on the Legion project at the University of Virginia.

Greg Lindahl (lindahl@cs.virginia.edu) is a research scientist at the University of Virginia.

Katherine Holcomb (holcomb@cs.virginia.edu) is a research scientist at the University of Virginia.

"Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."

© 1998 ACM 0002-0782/98/1100 $5.00
Campus-Wide Computing: 
Early Results Using Legion at the University of Virginia

Andrew S. Grimshaw
Anh Nguyen-Tuong
William A. Wulf

Abstract
The Legion project at the University of Virginia is an attempt to provide system services that provide the illusion of a single virtual machine to users, a virtual machine that provides both improved response time via parallel execution and greater throughput. Legion is targeted towards both workstation clusters and towards larger, wide-area, assemblies of workstations, supercomputers, and parallel supercomputers. Rather than construct Legion from scratch we are extending an existing object-oriented parallel processing system by aggressively incorporating lessons learned over twenty years by the heterogeneous distributed systems community. The campus-wide virtual computer is an early Legion prototype. In this paper we present challenges that had to be overcome to realize a working CWVC, as well as performance on a production biochemistry application.

1. Introduction

Providing resources to computationally demanding applications at the lowest cost is a challenge facing many organizations. The traditional solution to providing the necessary cycles has been to use a supercomputer. An alternative, less costly, solution that has emerged recently is to use networks of existing high-performance workstations instead, managing the collection of resources as a single entity. These systems are called variously “workstation farms” or “workstation clusters”. The advantage of the cluster approach is that the resources are often already in place, and under-utilized. A second advantage is that the cost per MIP/FLOP is much less. A key problem that must be addressed in cluster computing is management. The collection of workstations is just that, a collection. Without system software to tie the machines together it is not easy for a user to exploit cycles on many different workstations.

There are two broad categories of solutions to the problem of managing the workstation resources, throughput oriented systems, and response-time oriented systems. Throughput oriented systems are interested in exploiting available resources in order to service the largest number of jobs, where a job is a single program that does not communicate with other jobs. There are several

1. This work is partially funded by NSF grants ASC-9201822 and CDA-8922545-01, National Laboratory of Medicine grant (LM04969), NRaD contract N00014-94-1-0882, and ARPA grant J-FBI-93-116.
such systems available today, DQS, Condor, LoadLeveler, LSF, CODINE, and NQS to name a few [16][20][28][29][31][33]. Response time oriented systems are concerned with minimizing the execution time of a single application, i.e., with harnessing the available workstations to act as a virtual parallel machine. The purpose is to more quickly solve larger problems than would otherwise be possible on a single workstation. Examples of such parallel processing tools available for workstations include Express, Linda, Piranha, Mentat, P4, and PVM [9][24][7][46]. The two objectives, throughput and response time, are not necessarily mutually exclusive.

The problem with existing systems is that they are incomplete. One system may provide load balancing and a shared file space, but not parallel processing or fault-tolerance. Another system may provide high application performance via parallel execution yet provide no fault tolerance and require all hosts to see the same file system image. No single system currently provides all of the features required.

The Legion project at the University of Virginia is an attempt to provide system services that provide the illusion of a single virtual machine to users, a virtual machine that provides both improved response time via parallel execution and greater throughput [21]. Legion is targeted towards both workstation clusters and towards larger, wide-area, assemblies of workstations, supercomputers, and parallel supercomputers. Legion tackles problems not solved by existing workstation based parallel processing tools such as fault-tolerance, wide area network support, heterogeneity, the lack of a single file name space, protection and security, as well as providing efficient scheduling and comprehensive resource management. At the same time Legion provides the parallel processing, object-interoperability, task scheduling, security, and file system facilities not usually found in job-based load balancing systems.

Rather than attempt to construct Legion from scratch, we have chosen an evolutionary approach. We began by first constructing a campus-wide virtual computer (CWVC) testbed based on Mentat [24][25]. Mentat is an existing, robust, object-oriented, parallel processing system that supports execution across heterogeneous platforms. Starting with an existing system eliminates the long lead times and uncertainty associated with starting from scratch. Additionally, existing applications can be run in the new environment permitting ideas and system implementations to be tested using real rather than synthetic applications. For example, because Mentat is implemented
using replaceable modules (objects) we can easily experiment with new scheduling algorithms developed for the Legion environment without changing the applications.

The campus-wide virtual computer is a direct extension of Mentat onto a larger scale, and is a prototype for the nationwide system and reflects the fact that the university is a microcosm of the world. The computational resources at the University are operated by many different departments, there is no shared name space, and sharing of resources is currently rare. Resources owned by the departments consist of a variety of workstations (SUN, DEC, HP, IBM, SGI) and small parallel machines. This equipment is used for "production" applications during the day.

Even though the CWVC is much smaller, and the components much closer together, than in the envisioned nationwide Legion, it still presents many of the same challenges. The processors are heterogeneous, the interconnection network is irregular, with orders of magnitude differences in bandwidth and latency, and the machines are currently in use for on-site applications that must not be negatively impacted. Further, each department operates essentially as an island of service, with its own NFS mount structure, and trusting only machines in the island.

The CWVC is both a prototype and a demonstration project. The objectives are to:

- demonstrate the usefulness of network-based, heterogeneous, parallel processing to university computational science problems,
- provide a shared high-performance resource for university researchers,
- provide a given level of service (as measured by turn-around time) at reduced cost,
- act as a testbed for the nationwide Legion.

The prototype consists of over eighty workstations and an IBM SP-2 in six buildings and is now operational\(^2\). Before the system could become useful to users several challenges had to be overcome; simply scaling Mentat to run on the eighty machines was insufficient. The first challenge is to construct a federated file system and unified name space so that files, both executables and user data, can be accessed from any host in the system. The second challenge is to make the system able to deal with host and network failure in a graceful fashion. If one or more hosts fail the system must continue to operate without interruption. In a university environment, where some of the hosts are in public labs, periodic host failure is the rule and not the exception. The third challenge is protection and security. Mentat, like many other parallel processing systems,

\(^2\) To see the CWVC in action connect to our web page at http://uvacs.cs.virginia.edu/~mentat/legion/ and follow the links to Cyberia. Cyberia is an on-line look at the running CWVC.
does not address security. Each user either starts their own copy of Mentat and runs in their protection domain. Alternatively, the user uses a shared copy (one set of daemons, always running), in which case their application runs in the same protection domain as the shared copy. Thus all of their data files must be world readable, and their output files must be world writable. This is acceptable for a small group of colleagues, but is unacceptable for a large system.

The last two challenges have their roots in human nature rather than technical necessity. The fourth challenge is to provide autonomy to workstation owners and system administrators. To allow users and local system administrators to control resource utilization on their hosts in order to enforce local scheduling policies and to allow users to throttle system utilization of their hosts. We call this “pain management” because without the ability to effect local utilization users feel that they are in pain and are reluctant to add their resources to the pool. The fifth and final challenge discussed here is resource accounting. To avoid the tragedy of the commons in which everyone uses resources yet none contribute it is necessary to know how much resource is contributed (offered and used) by each user and how much is consumed. Users who contribute more than they use can be rewarded, and those who consume more than the contribute can be charged. How users are rewarded and charged is a policy issue not yet addressed. Keeping track of resource utilization requires mechanism. Finally there is performance. While implementing solutions to the above challenges we must not let performance suffer! For many users that is after all the whole point.

In this paper we present our early results using the CWVC and outline the shape of the solutions to the six challenges. To demonstrate performance we use a biochemistry application, complib, that compares DNA and protein sequences. The performance results are encouraging. Several other production applications have been developed by and for university researchers, including planetary atmosphere simulations, steric acid circulation tank simulations, solid state circuit simulations, parallel genetic algorithms, and 3D image segmentation.

We begin our presentation with background material on Legion and Mentat. We then look at the CWVC, discussing progress over the last year in each of the six challenge areas. The computational environment is then described to provide context for the performance results. We then briefly describe complib, presenting an English description of the problem and its computational structure. The results are next, followed by a discussion of our next steps.
2. Background

2.1. Legion

Legion will consist of workstations, vector supercomputers, and parallel supercomputers connected by local area networks, enterprise-wide networks, and the National Information Infrastructure. The total computation power of such an assembly of machines is enormous, approaching a petaflop; this massive potential is, as yet, unrealized. These machines are currently tied together in a loose confederation of shared communication resources used primarily to support electronic mail, file transfer, and remote login. However, these resources could be used to provide far more than just communication services; they have the potential to provide a single, seamless, computational environment in which processor cycles, communication, and data are all shared, and in which the workstation across the continent is no less a resource than the one down the hall.

A Legion user has the illusion of a single, very powerful computer on her desk. It is Legion's responsibility to transparently schedule application components on processors, manage data transfer and coercion, and provide communication and synchronization in such a manner as to minimize execution time via parallel execution of the application components. System boundaries will be invisible, as will the location of data and the existence of faults.

Before the Legion vision can be realized, several technical challenges must be overcome. These are software problems; the hardware challenges are being addressed and are the enabling technologies that provide the opportunity. The software challenges revolve around eight central themes: achieving high performance via parallelism, managing and exploiting component heterogeneity, resource management, file and data access, fault-tolerance, ease-of-use and user interfaces, protection and authentication, and exploitation of high-performance communications protocols. We realize that these are serious issues; we examine them in more detail in [21][22].

In addition to the purely technical issues, there are also political, sociological, and economic ones. These include encouraging the participation of resource-rich centers and the avoidance of the human tendency to free-ride. We intend to discourage such practices by developing and employing accounting policies that encourage good community behavior.
2.2. Related Work

The vision of a seamless metasystem or metacomputer such as Legion is not novel. Indeed, a number of systems have been designed to attack one or more of the problems mentioned above, e.g., Andrew, Locus and NSF for file systems [32][35][49], Locus for fault-tolerance, Sun XDR and the University of Washington HCS for heterogeneity[39][40][45]. None has been fully successful. Several changes have occurred that make the realization of a complete high performance metacomputer possible. First, high-speed optical communication has revolutionized long distance data communication. Realized bandwidths have gone from T1 (1.5 mBits/Sec.) to OC/12 over Sonet (152 mBits/sec.), with similar magnitudes of change expected in the next five years. The increase in bandwidth makes nation-wide metasystems practical and not just a science fiction fantasy. The second change is that achieving high performance via parallelism, previously available only for tightly coupled parallel processors, is now possible for loosely coupled distributed systems [7][8][9][12][27][34][37][46][51]. Related advances in resource management driven by parallel computing have attacked the problem of decomposing and scheduling application components to minimize elapsed time.

Legion differs from other work in heterogeneous distributed systems [3][4][5][6][19][35][36][38][39][40][47][48][49] and object-oriented distributed systems [2][13][38]. One major difference is our emphasis on performance. Often interoperability, fault-tolerance, or consistency is the main focus, and performance is sacrificed. Legion though is performance oriented. The underlying model is parallel, and the user, not the system designers, will choose the appropriate level of fault tolerance and consistency that best meets their needs.

There are other metasystems [17][30][43][50] and heterogeneous parallel computing [7][10][18][46] projects underway. Our work differs from other heterogeneous parallel processing systems in the scope. We’re addressing file systems, fault-tolerance, interoperability, etc. Our work differs from the other metasystems efforts in that we have a system already in place that can evolve - we are not starting from scratch.

As mentioned in the introduction several systems have been developed to manage workstation farms, DQS, Condor, LoadLeveler, LSF, and CODINE. Legion differs in that it is both a throughput and a response time system, we attempt to both better utilize system CPU resources
and decrease execution time via parallelism. Further, Legion provides a single name space to users with data accessible from anywhere in the system. In other words Legion is a complete system solution, rather than a load balancing tool.

Whether or not a metasystem is explicitly constructed by design, the nation (and perhaps the world) will eventually build a system that shares at least some of the attributes of Legion. The reason is simple: individual and organizational users will be required to deal with the increasingly obvious shortcomings of a computing infrastructure consisting of islands of computational power connected via the Internet. Internet tools such as gopher, worldwide web and Mosaic are examples of current attempts to bridge the gaps between local systems.

The issue is not whether metasystems will be developed; clearly they will. Rather, the question is whether they will come about by design and in a coherent, seamless system – or painfully and in an ad hoc manner by patching together congeries of independently developed systems, each with different objectives, design philosophies, and computation models.

2.3. Approach

The principles of the object-oriented paradigm are the foundation for the construction of Legion; our goal will be exploitation of the paradigm’s encapsulation and inheritance properties, as well as benefits such as software reuse, fault containment, and reduction in complexity. The need for the paradigm is particularly acute in a system as large and complex as Legion. Other investigators have proposed constructing application libraries and applications for wide-area parallel processing using only low-level message passing services. Use of such tools requires the programmer to address the full complexity of the environment; the difficult problems of managing faults, scheduling, load balancing, etc., are likely to overwhelm all but the best programmers.

3. The CWVC

In order to achieve campus wide computing it is insufficient to simply run Mentat on all machines at the University. Mentat, like other existing parallel processing systems, was not designed to deal with system faults, non-overlapping file systems, and other problems that plague distributed systems as opposed to MPP’s. Further, we do not believe that the Mentat programming language is the best language for all applications. Therefore, other models and languages will need
to be supported in the long run for Legion to be successful. Below we present the solutions we have implemented in the CWVC to the problems of files and I/O, fault tolerance, pain management, accounting, protection and security, and the need to support other programming models.

3.1. The Federated File System

A problem that arises when forming a system out of hosts in different organizations is that, unlike a single department, they usually do not share a single file system. This presents at least four difficulties; application binaries may not be present at all sites; application components may not be able to read and write files that they require for correct execution; sharing of data and results between collaborators at different sites requires either sending the files via email or copying them via ftp; and remote databases, e.g., genome databases, must be copied in their entirety into the local environment before they can be used, resulting in wasteful copies, out-of-date data, and the inconvenience of frequently manually copying the data.

An obvious solution to this problem is to extend some existing file system such as AFS or NFS to both the campus wide and nation wide system. There are problems with this approach. First, some file systems such as NFS simply will not scale to the level we require. They require far too much human intervention (setting up mount points, etc.). Further, the use of NFS requires that all users have the same user id on all hosts, a requirement we simply cannot meet. The Andrew file system, on the other hand, is scalable. The problem with using Andrew is the requirement that all hosts run Andrew. We feel that we cannot impose a file system standard on participating organizations. Further, we do not feel that the system should dictate a particular file semantics as Andrew does; file semantics should be based on the file type. Nevertheless we intend to borrow heavily from early distributed file system projects such as Andrew because issues such as naming, location transparency, fault transparency, replication transparency, and migration have been addressed both in the literature [32] and in one or more existing operational systems.

Our solution is to construct a federated file system using the local host file systems as the component elements. The basic idea is simple and consists of two parts, constructing a Legion name space on top of the existing file system, and a modified set of stdio libraries that interact with the Legion file space and Legion file objects. The Legion name space is implemented using a
collection of name servers that point to instances of file classes. A file is in the Legion name space if its name (complete path name) is found in one of the name servers. Otherwise it is a local file.

The name servers keep track of the name of the file, e.g., "/legion/grimshaw/mywork1" , and a set of file attributes\(^3\). The attributes include the class of the file, e.g., unix_file, restrictions on the placement of class instances, e.g. on hosts with a ".cs.virginia.edu" suffix, and an initialization string to be passed to an object instance on instantiation. This string is most often the Unix path name of the actual file data.

We also provide a set of library routines that support the stdio interface, e.g., open, close, read, write, etc. These library routines are linked by applications and intercept calls to the I/O system. All open calls are first examined to determine whether they are Legion files or local files. Local file operations are passed onto the host operating system. Legion file operations are trapped and passed onto the unix_file objects. This technique of trapping file operations is not unique to Legion and was first used in Unix United. Unfortunately the relinking technique does not work on all supported platforms. Therefore we have had to design and implement a separate I/O library for use by applications on those platforms.

Finally, there exist a set of operations to manipulate the Legion namespace that are analogous to operations that manipulate the Unix name space, e.g., lmkdir, lrm, lmv, etc. Files enter the Legion namespace via one of three mechanisms, they are created in the Legion name space,

\(^3\) The implementation details are changing—the philosophy is not.
they are moved into the Legion name space, or they are linked into the Legion names space (similar to a soft-link).

3.2. System Fault Tolerance

With a system as large as the CWVC it is a certainty that a machine will go down every few hours. Further, whole buildings will lose power, while other buildings will not. Mentat, and other network-capable parallel processing systems were not designed to address any form of fault tolerance. We knew that if we wanted to construct a facility that others would use it had to be highly available (there when they needed it) and robust to failures of all kinds, host failure, network failure, application failure, etc. The mechanism to provide full application fault-tolerance is a long term goal and is still in the research and design stage. In the short term though, the system itself must be fault tolerant or no one will use the it; contemporary computer users have very high expectations with regard to system availability. Before we can describe the current fault tolerance mechanism a brief bit of background is required.

Each host in a running Legion system has at least two daemons (exclusive of the fault tolerance daemons). The first is the instantiation manager (IM) which is responsible for scheduling, keeping track of all Legion objects on the host, monitoring the load on the system, and other management functions. The second is the token matching unit (TMU) which supports language features [26]. The system configuration file (config.db) contains the names of all hosts in Legion\(^4\). The set of IM’s running on those hosts define the system. If an IM goes down then Legion is down on that host. Further, user programs, user objects, and IM’s communicate with IM’s in order to carry out their functions. If an IM on a host goes down (either because of a program fault or host failure) objects on other hosts will not receive a response and will lock up, i.e., they will fail. To eliminate this form of failure we must ensure either that hosts and IM’s never fail, an impossible goal, or deal with failure when it occurs.

To deal with failure we have constructed a new class of daemons, phoenix, that monitors the system for failure. Phoenix is responsible for restarting system components if possible. If they cannot be restarted phoenix notifies the remaining IM’s of a configuration change. The IM’s in turn notify all objects on their host. If a host has been removed from the configuration due to failure

\(^4\) The configuration database is a non-scalable entity. Future releases will use a different mechanism.
phoenix will begin to monitor the host for recovery. When the host has recovered, phoenix will restart Legion on that host and notify the other IM’s of the configuration change.

The phoenix instances are arranged in a tree structure. The root phoenix starts cluster phoenix instances on a single host in each cluster. These cluster phoenix instances in turn start a leaf phoenix instance on each host in the cluster. The leaf phoenix instances monitor the health of the local daemons, restarting them as necessary. The root phoenix monitors the health of the leaves. When one fails (usually due to a host failure) the root restarts it. This design is non-scalable and has a single point of failure (the root). Because of the single point of failure we place the root on our most reliable host. A long term solution to both the stability and single point of failure problems is to use a distributed algorithm. So far though, neither has been a serious problem.

Transparent application fault-tolerance is being investigated but is not currently implemented. We believe that fault-tolerance services are a necessary condition for success of any large scale system such as Legion. We expect to have fault-tolerance for stateless objects and applications which use them by Supercomputing. This encompasses a large class of applications.

3.3. Pain Management

A third problem is pain. The pain that workstation “owners” feel when a Legion job is executing on their machine. Because of the pain, and the need not to antagonize resource owners, we have constructed a Legion “thermostat” (Figure 2) that permits resource owners to set maximum resource use limits (in percent CPU and physical memory) on Legion programs running on their hosts. The thermostat mechanism works in a fashion similar to many home thermostats. The owner can select resource availability during different time intervals. Further, just as with a home thermostat, if the owner is unhappy with the current setting, she may change it at any time. (Just like a home thermostat it may take a few minutes before the load is adjusted to the new level.)

The system guarantees that Legion resource consumption will stay below the limits imposed by the user. The available resources are divided between user objects running on the host. Thus, some user objects will be “throttled”, resulting in possibly longer execution times for applications. We feel that the trade-off between autonomy and application performance must be made in favor of the local user. If it is not, then resource owners may withdraw their resources.
Figure 2 Legion Thermostat. Authorized users can specify both a resource schedule and modify the current resource limits. The interface on the left controls current resource limits. By changing the sliders resource limits may be temporarily modified. The “scheduler” button brings up the interface on the right, which allows authorized users to change the daily schedule.

In order to encourage resource owners to participate and specify high resource limits the resource owner receives credits when resources are consumed and when resources are offered even if they are not used. The amount of credit received depends on the type of system, the time of day, and how much resource was offered and used. Different amounts of credit are received for offered but not used, and offered and used resource. The scale factors (for type of system and time of day) are in essence prices. Prices are a policy issue determined by system administrators.

3.4. Accounting

If Legion and the CWVC are successful then all a user will need in order to have access to the set of resources managed by the system is an interface such as an X-Windows interface. In such an environment a rational user will purchase just the interface, and leave the purchase of expensive resources to other users. It would not be long before no new equipment was purchased, resulting in an impoverished system. To avoid this classic “tragedy of the commons”, mechanism and policy are required to encourage good community behavior. In the case of Legion and the CWVC this takes the form of an accounting system which monitors resource contribution and consumption.
Accounting for resource consumption is accomplished by associating each instantiated user object with an owner, a CWVC user, and monitoring user object activity. The owner-id is the Unix UID of the shell which launched the application. Resource consumption is monitored on an object-by-object basis. The run-time libraries used by objects are instrumented to collect information such as the amount of CPU used, the number of messages sent and received, the total volume of data moved in and out of the object, and the number of method invocations performed. On object termination this information is forwarded to the instantiation manager. The instantiation manager in turn places the resource data into a host specific accounting database. Periodically the accounting data is collected, merged into a single database, and resource reports are generated.

Resource contribution is monitored similarly. Actual resource contribution is derived from the resource consumption database; if resources were used on a host by an object then they were contributed by that host. Offered resources are determined by maintaining thermostat logs. Recall the thermostat is used to throttle resource consumption on a host by restricting usage to a particular percentage. The thermostat log indicates when thermostat settings were changed on a host.

Both resource contribution and consumption are scaled using the resource scale file. The scale file consists of scale factors for each host type, the time of day, and whether the resource was consumed, offered, or contributed. This is used to account for the fact that, for example, a Sparc IPC CPU second at 1:00 AM is not worth nearly as much as an SGI CPU second at noon. The scale factors can be thought of as prices. Using the accounting files and the resource scale file we can calculate a resource balance for each user. The resource balance is the users amount of surplus or deficit. A system-wide surplus, available to system administrators, can be generated by maintaining a spread between the price for resources consumed and resources offered.

The mechanism lets us to know who is using and contributing how much resource. Without a policy on resource consumption collecting the information is an exercise in programming only. Policy is still being worked out with resource holders. We envision a situation in which users with chronic resource deficits can either buy (using dollars) more resource or receive "grants" from the system. The grants will come out of the system-wide surplus. The funds generated can either be used to pay users with chronic surpluses, or be re-invested in additional equipment.

5 Use of the Unix ID is temporary, they are not consistent across systems. A Legion user ID, LUID, will be used in the future, introducing authentication issues not yet addressed in the implementation.
3.5. Protection and Security

Protection and security are critical in wide-area distributed systems. Mentat does not address security; instead it relies on having read/write permission to all application databases. Our security plan has long-term and short-term components.

**Long term.** Legion cannot solve the general security problem, but a reasonable level of privacy and integrity of data must be provided or the system will not be used. Legion will run on top of whatever operating system is available on the participating host, thus at first blush it may seem that the issue of security would be determined by the lowest common denominator of those host systems. The first rule of Legion security, like the Hippocratic Oath, will be to “do no harm.” In the first instance, that means that there should be no possibility of host data being compromised by a Legion task. In the second instance it means that no Legion user’s data should be compromised by a rogue host. The mechanisms are reasonably well understood and include:

- execution of Legion tasks with “least privilege” on the host,
- storing no persistent objects on a “foreign host”,
- using cryptographic authentication protocols to verify that a server is valid and that a service request originated with a valid Legion task, and
- digital signatures to validate that data has not been corrupted.

Further, when needed for greater security, more costly techniques can be used, for example:

- messages can be encrypted,
- scheduling can be restricted to certain classes of nodes, and
- white noise can be injected into the communication stream.

Another goal of Legion security is to provide a better model and mechanism for Legion-to-Legion security than is provided by the underlying hosts. The object-oriented computational model suggests a capability-based access control model in which the “rights” are type (class) specific and map one-to-one onto the methods of the class. The distributed and heterogeneous nature of the system suggests the use of encrypted capabilities to ensure that they cannot be forged.

**Short term.** In the short-term we have implemented a security scheme which a determined attacker could compromise. The basic idea is simple. Users “transfer” files to the Legion name space. The file is placed in a hidden directory that only they have read privilege to, and a soft-link is created with the same name as the old file. This ensures that the user can still modify the file. At the same time a hard-link is created for Legion to the file. This permits Legion applications to read...
and write the file. The net result is that both the user and Legion programs may read and write the file, but other users cannot. The "hole" in this scheme is that other users Legion programs may read and write the file.

3.6. Support for other Models - PVM V 3.0

Clearly we cannot expect all Legion users to use MPL. Other parallel processing languages and models must be supported. This is particularly true for legacy codes. Our plan for legacy code and multiple model interoperability is detailed in [21]. One of the early tests for our plan is support for PVM [46]. PVM is a message Passing parallel processing system that has gained wide-spread acceptance in the user community, and is a de facto standard.

At first glance it seams counter-intuitive to implement PVM on top of a system such as the CWVC and Legion; they are usually layered on top of systems like PVM. We have implemented PVM on the CWVC so that existing PVM applications can be executed in the CWVC environment without the need to re-write the application. The ported PVM codes will not only execute, they will benefit from CWVC features such as the federated file system, pain management, and our load-sensitive scheduling algorithms. This ability to use existing codes, and to later integrate them with other parallel codes, support our larger Legion goals of legacy code support and interoperability.

The implementation is straightforward. Each PVM instance is represented by a typed Mentat object. Calls such as pvm_initiate(), pvm_send(), and pvm_recv() are implemented using calls to the CWVC run-time system that instantiate objects, send messages, receive messages, etc. PVM-specific services, e.g., barriers, are implemented using synchronization objects.

To both test the correctness of our implementation, and to compare the performance of our implementation against the "native" PVM implementation, we took to already implemented PVM applications and ran them in the same environment. The two applications are the latency/ bandwidth test code that comes with the PVM distribution, and the PVM implementation of the NAS benchmarks. TABLE I presents the performance comparison between the native PVM and Mentat-PVM implementations on the communication benchmarks. One can see that for short messages (<= 10,000 bytes) the Mentat-PVM implementation outperforms the native implementation. The Mentat-PVM time is the result of the optimized communication system used.
TABLE 1 Point to Point benchmarks on 40 mhz Sparc 2's.a.

<table>
<thead>
<tr>
<th>Message size (bytes)</th>
<th>Native PVM time (mSec)</th>
<th>Mentat-PVM time (mSec)</th>
<th>Native PVM bandwidth (bytes/Sec)</th>
<th>Mentat-PVM Bandwidth (bytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>8.81</td>
<td>8.47</td>
<td>11,800</td>
<td>13,310</td>
</tr>
<tr>
<td>1000</td>
<td>9.88</td>
<td>9.75</td>
<td>101,269</td>
<td>102,550</td>
</tr>
<tr>
<td>10,000</td>
<td>27.52</td>
<td>26.58</td>
<td>363,380</td>
<td>376,220</td>
</tr>
<tr>
<td>100,000</td>
<td>194,231</td>
<td>196,783</td>
<td>514,860</td>
<td>509,880</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,920,624</td>
<td>1,951,618</td>
<td>520,670</td>
<td>512,790</td>
</tr>
</tbody>
</table>

a. All measurements are on an 8 processor cluster using raw (no XDR) encoding, and direct routing. The time is the time to send a message of the specified size and to receive a one byte reply. It is the average of 3 runs (each run being the average of 20 sends/receives).

by Mentat. That short messages are faster is significant. Most messages in many applications are short messages.

To test application performance we selected the PVM NAS benchmark implementation because it is familiar to most readers. TABLE 2 presents the performance results. The measurements were taken using eight, 40 mhz Sparc 2's, using raw encoding.

TABLE 2 NAS Benchmark Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Native PVM</th>
<th>Mentat-PVM</th>
<th>Native PVM</th>
<th>Mentat-PVM</th>
<th>Native PVM</th>
<th>Mentat-PVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS Kernel^b</td>
<td>226</td>
<td>256</td>
<td>202</td>
<td>207</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>EP Kernel^b</td>
<td>346</td>
<td>350</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MG Kernel^c</td>
<td>123</td>
<td>110</td>
<td>62</td>
<td>59</td>
<td>49</td>
<td>49</td>
</tr>
</tbody>
</table>

a. 2^{21} keys in the range [0..2^{19}]
b. reduced problem size of 2^{26}
c. problem size of 128

4. Results

4.1. The Application - Protein and DNA Sequence Library Comparison

Our test application, complib, compares two protein or DNA sequence libraries. Each library contains of one or more sequences, each of which consists of a sequence name and a variable length string of characters (also known as residues) that represent the sequence. Each sequence in the first library, called the source library, is compared against each sequence in the second, target, library. For each sequence comparison a score is generated reflecting sequence
commonality using one of several algorithms. Three popular algorithms are Smith-Waterman [44], FASTA [39], and Blast [1]. The latter two algorithms are heuristics; the quality of the score is traded for speed. Smith-Waterman is the benchmark algorithm, generating the most reliable scores although at considerable time expense. We used the Smith-Waterman algorithm to compare our performance against previously published numbers in the biochemistry community [14][42].

Once all of the scores have been generated for one source sequence against all target sequences the scores are sorted and statistical information is generated. Thus, there are two phases that are executed sequentially for each source sequence, sequence comparison, and data reduction. An important attribute of the comparison algorithms is that all comparisons are independent of one another and, if many sequences are to be compared, they can be compared in any order. This natural data-parallelism is easy to exploit.

The main program manipulates three objects, the source genome library, the target genome library, and a recorder object that performs the statistical analysis and saves the results. The application is written in the Mentat programming language and is described in detail in [23]. The main program for loop is shown in Figure 3 below. The effect is that a pipe is formed, with sequence extraction from the source, sequence comparison in the target, and statistics generation are executed in a pipelined fashion. Each high-level sequence comparison is transparently expanded into a fan-out, fan-in program graph where the “leaves” are the workers, the source sequence is transmitted from the root of the tree to the leaves, and the results are collected and sorted by collators.

4.2. The Campus Computing Environment

All tests were conducted on the grounds-wide network at the University of Virginia. The available computing resources are shown in TABLE 3. The hosts were physically located in six different buildings. Each building has one or more Ethernet segments connected to the grounds-wide fiber backbone by routers. Some of the hosts are several “hops” from the fiber backbone. For these experiments all machines had at least one shared NFS mount point where all executables and data were stored. In general this is not a satisfactory solution as we cannot count on cross-mounted file systems.

6. At the University of Virginia, the campus is called the “grounds”.

17
for(i=0;i<num_source_seq;i++) {
    //for each sequence
    s_val = source.get_next();
    //Compare against target library
    result = target.compare(s_val);
    //Do statistics
    post_process.do_stats(result, s_val);
}

Figure 3 Mentat implementation of complib. The main loop of the program is shown in (a). Three objects are manipulated, the source, the target, and the postprocessor. The pipelined program graph is shown in (b). Target.compare() has been expanded showing sixteen workers in (c). The fan-out tree distributes the source sequence to the workers. The internal nodes of the reduction tree are collator objects. The reduction tree sorts and merges the results generated by the workers.

4.3. Complib results & discussion

All measurements were performed in early January 1995 during the Winter break. In general, workstations were operating under light loads except for two of the departmental Sparc 10 compute servers. Execution times for complib were generated using a 20 sequence source library containing 4478 residues and a 10,716 sequence target library consisting of 3,647,403 residues. We follow the tradition of the biochemistry community of reporting performance numbers in terms of millions of matrix entries per second. The number of matrix entries is obtained by multiplying the number of residues in the source library with the target library. In our experiment, there were 4478*3,647,403 residues, or approximately 16,333 million matrix entries.
To provide a benchmark for comparison we executed a sequential version of complib on all the platforms that comprise the CWVC. TABLE 3 lists the sequential execution times as well as the corresponding million matrix entries per second. In each case, the user CPU time was reported.

**TABLE 3 . Sequential complib (20 vs. 10716 sequences, 16,333 million matrix entries)**

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Number</th>
<th>Sequential Time (sec)</th>
<th>Matrix entries per second (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparc 10 (fast)</td>
<td>4</td>
<td>13460</td>
<td>1.21</td>
</tr>
<tr>
<td>Sparc 10 (slow)</td>
<td>8</td>
<td>13998</td>
<td>1.17</td>
</tr>
<tr>
<td>Sparc LX</td>
<td>6</td>
<td>23146</td>
<td>0.71</td>
</tr>
<tr>
<td>Sparc 2</td>
<td>5</td>
<td>33823</td>
<td>0.48</td>
</tr>
<tr>
<td>Sparc IPC</td>
<td>41</td>
<td>57259</td>
<td>0.29</td>
</tr>
<tr>
<td>SGI Indigo</td>
<td>17</td>
<td>11386</td>
<td>1.43</td>
</tr>
</tbody>
</table>

TABLE 4 gives the parallel execution times on the CWVC as well as the speedup relative to each of the platforms. The number of workers ranges from 8 to 64. Three runs of each were performed. The times shown represent the best wall clock time obtained excluding the initial overhead of distributing the target library\(^7\). All times are given in seconds.

**TABLE 4 . CWVC Times & Relative Speedups**

<table>
<thead>
<tr>
<th>Workers</th>
<th>CWVC</th>
<th>SGI</th>
<th>Sparc 10 (fast)</th>
<th>Sparc 10 (slow)</th>
<th>Sparc LX</th>
<th>Sparc 2</th>
<th>Sparc IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1692</td>
<td>6.7</td>
<td>7.9</td>
<td>8.3</td>
<td>13.7</td>
<td>20.0</td>
<td>33.8</td>
</tr>
<tr>
<td>16</td>
<td>926</td>
<td>12.3</td>
<td>14.5</td>
<td>15.1</td>
<td>25.0</td>
<td>36.5</td>
<td>61.8</td>
</tr>
<tr>
<td>24</td>
<td>951</td>
<td>12.0</td>
<td>15.2</td>
<td>14.7</td>
<td>24.3</td>
<td>35.6</td>
<td>60.2</td>
</tr>
<tr>
<td>32</td>
<td>724</td>
<td>15.7</td>
<td>18.6</td>
<td>19.3</td>
<td>32.0</td>
<td>46.7</td>
<td>79.1</td>
</tr>
<tr>
<td>64</td>
<td>592</td>
<td>19.2</td>
<td>22.7</td>
<td>23.6</td>
<td>39.1</td>
<td>57.1</td>
<td>96.7</td>
</tr>
</tbody>
</table>

\(^7\) In table 6, we show the performance of complib on the CWVC against other platforms\([14][42]\). All measurements reported were obtained by excluding the initial overhead time.
4.4. Is speedup meaningful in the CWVC?

A generalized definition for speedup in a heterogeneous environment is given in [15]. In our particular case, we characterize the performance of complib in terms of the number of matrix entries per second obtained. Furthermore, we define the efficiency of running complib on the CWVC as:

\[
\text{efficiency (CWVC,complib)} = \frac{\text{measured performance on the CWVC}}{\text{maximum theoretical performance}}
\]

The maximum theoretical performance is obtained by assuming that workers are placed on the more powerful processors first. For example, in the case of 24 workers, the theoretical maximum performance is 32.66 million matrix entries per second (17 workers placed on the sgis, 4 on the faster sparc10s, and 3 on the slower sparc10s). Note that efficiency is dependent on both the particular application, and the types and numbers of hosts that make up the CWVC.

**TABLE 5. Efficiency**

<table>
<thead>
<tr>
<th>Workers</th>
<th>Theoretical maximum matrix entries per second (millions)</th>
<th>Matrix entries per second (millions) on the CWVC</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>11.44</td>
<td>9.65</td>
<td>.84</td>
</tr>
<tr>
<td>16</td>
<td>22.88</td>
<td>17.64</td>
<td>.77</td>
</tr>
<tr>
<td>24</td>
<td>32.66</td>
<td>17.17</td>
<td>.54</td>
</tr>
<tr>
<td>32</td>
<td>40.64</td>
<td>22.56</td>
<td>.55</td>
</tr>
<tr>
<td>64</td>
<td>52.13</td>
<td>27.59</td>
<td>.53</td>
</tr>
</tbody>
</table>

The efficiency obtained for 24 to 64 workers is slightly above 50%. There are two causes for the reduced efficiency: other users competing for cycles, and a poor initial partition of the data. Our initial partitioner did not handle the heterogeneous processor case well. We have re-implemented the partitioner and expect better results in the future.

While efficiency gives us a measure of the overhead in running complib on the CWVC, the important performance number for biochemists is the number of matrix entries per second. We compare the number of matrix entries per second obtained against other platforms in TABLE 6. In terms of performance, the CWVC is roughly equivalent to the 32 node Paragon at the Jet Propulsion Lab.
TABLE 6. Comparison against various platforms\textsuperscript{a}

<table>
<thead>
<tr>
<th>Platform</th>
<th>Matrix entries per second (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWVC</td>
<td>27.59</td>
</tr>
<tr>
<td>CM-2 (32000 proc)</td>
<td>65</td>
</tr>
<tr>
<td>Paragon (32 nodes)</td>
<td>29</td>
</tr>
<tr>
<td>5 DEC Alpha AXP 300</td>
<td>18</td>
</tr>
</tbody>
</table>

\textsuperscript{a} The CM-2 performance numbers were obtained from [14], the Paragon and DEC Alphas' from [42].

4.5. Problems encountered

We encountered several problems transforming Mentat into the CWVC. Some of these, such as the need for a federated file system and single name space were anticipated. Others were not. At least three of these problems are not Legion specific, they will need to be overcome by most systems that have the same objectives as Legion/CWVC.

The first problem is the trend in workstation operating systems towards the exclusive use of dynamic linkers. For example, Irix, Solaris, and AIX do not support static linking. This is a problem whenever hosts do not share the same file system structure, in other words object libraries may be at a different location on different hosts, or not present at all. Thus, an executable that executes on one host, will not execute on another host of the same architecture type. This limits our ability to transport binaries from one location to another. Executable transport is not an issue in older operating systems such as SunOS which permit static linking.

Another facet of this problem occurred when we implemented the stdio library call traps. We could not statically link our routines in to replace the underlying C library routines. To solve this problem required that we explicitly manage the dynamic linker in our code. Unfortunately the mechanisms required vary from system to system.

A second class of problems that we encountered relate to Unix itself. It is not sufficient to simply scale the number of hosts when using a Unix based parallel processing tool if there is a single host that starts remote shells executing on other hosts. For example, if a daemon on host A starts daemons on hosts B..Z. This technique, which we used to practice, and which PVM uses begins to fail when there are many hosts for at least two reasons. First, the number of open files can rapidly exceed the limits of the operating system. (These limits can be changed by recompiling the
operating system.) The problem is that each rsh consumes at least two file descriptors, and possibly more if the pipe() command is used as well. Even if these file descriptors are closed by the “main” daemon they are not always closed immediately as called for in the manuals. Instead, there is usually a timeout of just under five minutes. Under normal operating conditions this is not a problem, but when a very large number of rsh’s are being generated in a short period of time, such as when starting the parallel system up, the system may run out of descriptors.

A related problem has to do with the use of NFS file servers. Simultaneous execution of a large number of copies (> 60) of the same executable, as in a data parallel program or system startup, may overload the file server, resulting in multiple lost requests. The host operating systems treat this as an error, and report that the executable does not exist, when clearly it does.

A final problem we encountered is application fault-tolerance. While the system itself is fault-tolerant and recovers from host failure, applications do not. If my application has an object on a host that has failed, then my application blocks, and never recovers. This requires the user to kill and restart the application. We consider this unacceptable in the long run. We have implemented fault-tolerance in two applications at the application level, and are exploring general application fault-tolerance.

5. Summary and On-going Work

Legion is an ambitious project to construct a nation-wide virtual computer. Rather than attempting to construct Legion from scratch we have chosen to begin with an existing system, Mentat, and transform it into Legion by incorporating research results from over twenty years of heterogeneous distributed computing. The first step in the transformation is the construction of the campus-wide virtual computer at the University of Virginia. The objectives of the campus-wide virtual computer are to:

- demonstrate the usefulness of network-based, heterogeneous parallel processing to university computational science problems,
- provide a shared high-performance resource for university researchers,
- provide a given level of service (as measured by turn-around time) at reduced cost,
- act as a testbed for the nationwide Legion.

The prototype implementation is well on its way to meeting these objectives. The performance results provide evidence that workstation based, heterogeneous parallel processing can be used to solve computationally challenging problems of interest to university researchers at
reduced cost. With respect to the testbed goal, the CWVC has been an invaluable tool which has enabled us to begin trying out designs and stressing implementations with real applications. Our experience over the last year putting the CWVC together highlighted several critical factors that must be addressed in both the short term and in the long term.

Finally, Legion and the CWVC are not static; both are works in progress. We will continue to enhance the system and extend the applications running on the CWVC. With respect to Legion we are forging ahead with enhanced security, protection, resource management, fault-tolerance, I/O, and naming services. In addition we are working on an OS/2 Warp port.

6. Acknowledgments

We would like to thank Bill Pearson of the biochemistry department for introducing us to sequence comparison, and for collaborating on the parallel version. We would also like to thank all of the members of the Legion team. The faculty are Bill Wulf, Jim French, Paul Reynolds Jr., and Alf Weaver. Staff members are Mark Hyett and Lindsey Faunt. The students who have worked on various components are Jon Weissman and Anh Nguyen-Tuong (run-time system), John Karpovich, Matt Judd, and Adam Ferrari (federated file system and I/O libraries), Emily West (complib) and with Mark Morgan (pain management and thermostat), Chenxi Wang (security), Roger Harper (PVM), and Mike Lewis. We would also like to thank the Jet Propulsion Lab for the use of their Intel Paragon.

7. References


[34] D. B. Loveman, "High Performance Fortran," IEEE Parallel & Distributed Technology: Systems & Applica-


A Flexible Security System for Metacomputing Environments*

Adam Ferrari, Frederick Knabe, Marty Humphrey, Steve Chapin, and Andrew Grimshaw

Department of Computer Science
University of Virginia, Charlottesville, VA 22903, USA

Technical Report CS-98-36

December 1, 1998

Abstract

A metacomputing environment is a collection of geographically distributed resources (people, computers, devices, databases) connected by one or more high-speed networks and potentially spanning multiple administrative domains. Security is an essential part of metasystem design—high-level resources and services defined by the metacomputer must be protected from one another and from possibly corrupted underlying resources, while those underlying resources must minimize their vulnerability to attacks from the metacomputer level. We present the Legion security architecture, a flexible, adaptable framework for solving the metacomputing security problem. We demonstrate that this framework is flexible enough to implement a wide range of security mechanisms and high-level policies.

1 Introduction

Legion [5, 6] is a distributed computing platform for combining very large collections of independently administered machines into single, coherent environments. Like a traditional operating system, Legion builds on a diverse set of lower-level resources to provide convenient user abstractions, services, and policy enforcement.

*This work was funded by DARPA contract N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, and DOE D459000-16-3C
mechanisms. The difference is that in Legion, the lower-level resources may con-
sist of thousands of heterogeneous processors, storage systems, databases, legacy
codes, and user objects, all distributed over wide-area networks spanning multiple
administrative domains. Legion provides the means to pull these scattered compo-
nents together into a single, object-based metacomputer that accommodates high
degrees of flexibility and site autonomy.

Security is an essential part of the Legion design. In a metacomputing environ-
ment, the security problem can be divided into two main concerns:

1. Protecting the metacomputer's high-level resources, services, and users from
each other and from possibly corrupted underlying resources, and

2. Preserving the security policies of the underlying resources that form the
foundation of the metacomputer and minimizing their vulnerability to attacks
from the metacomputer level.

For example, restricting who is able to configure a metacomputer-wide scheduling
service would fall in the first category, and its solution requires metacomputer-
specific definitions of identity, authorization, and access control. Meanwhile, en-
forcing a policy that permits only those metacomputer users who have local ac-
counts to run jobs on a given host falls in the second category, and it might require
a means to map between local identities and metacomputer identities.

To satisfy users and administrators, a full security solution must address and
reconcile both of these security concerns. Users must have confidence that the data
and computations they create within the metacomputer are adequately protected.
Administrators need assurances that by adding their resources to a metacomputer
(and thus making those resources more accessible and valuable to users), they are
not also introducing unreasonable security vulnerabilities into their systems.

Attempting to incorporate security as an add-on late in the implementation pro-
cess has been problematic in a number of first-generation metacomputing systems
such as PVM, MPI, and Mentat. To avoid this pitfall, the Legion group has ad-
dressed security issues since the earliest design phases [13]. Our metacomputing
security model has three interrelated design goals:

Flexibility. The framework must be adaptable to many different security policies
and allow multiple policies to coexist.

Autonomy. Organizations and users within a metacomputing environment should
be able to select and enforce their desired security policies independently.

Breadth. The metacomputer's architectural framework must enable a rich set of
security policy features.
These goals are strongly driven by our view that a fundamental capability of a metacomputer is its ability to scale over and across multiple trust domains. A Legion "system" is really a federation of meta- and lower-level resources from multiple domains, each with its own separately evaluated and enforced security policies. As such, there is no central kernel or trusted code base that can monitor and control all interactions between users and resources. Nor is there the concept of a superuser—no one person or entity controls all of the resources in a Legion system.

If it is to satisfy a broad range of security needs, our architecture must allow the implementation of a number of different security features. These include:

Isolation. Components in the metacomputer should be able to insulate themselves from security breaches in other parts of the system. This feature is particularly important in large-scale systems, where we must generally assume that at least some of the underlying hosts have been compromised or may even be malicious.

Access control. Resources typically require access control mechanisms that embody authentication and authorization policies.

Identity. The ability to assert and confirm identity is essential for access control, nonrepudiation, and other basic functions.

Detection and recovery. A metacomputing environment should support mechanisms for detecting intrusion and misuse of resources, and for recovering after a security breach.

Communication privacy and integrity. Communication over the networks that bind the metacomputer together may need to be encrypted or protected if the networks cannot themselves be trusted.

Standards. Existing security standards such as Kerberos, ssh, DCE, etc., may need to be integrated into the metacomputing environment to satisfy local administrative policy and to handle legacy software.

In this paper we elaborate a metacomputing architecture based on our design goals that addresses both parts of the metacomputing security problem. We also describe a wide set of mechanisms we have designed or implemented that enable a number of useful security policies, and provide examples of those policies. The key strength of our framework is its ability to support these policies and mechanisms along with many others.
The rest of the paper is organized as follows: Section 2 describes the Legion system architecture, concentrating on the elements most closely related to security in the system. Section 3 describes concrete security mechanism designs that we have implemented within the framework of the Legion architecture. Section 4 discusses examples of high-level policies, and how these policies can be implemented within the Legion security system. Section 5 describes related systems and approaches. Section 6 contains conclusions.

2 Architecture

Legion was designed with the explicit intent of supporting a powerful, flexible security architecture. Basic Legion design principles such as encapsulation, extensibility, flexibility, autonomy, and scalability have resulted in a system that can support the varied requirements of application programmers, tool builders, and resource providers. In this section, we examine the fundamental elements of the Legion architecture, introducing the concepts that will be the basis for the rest of the paper.

2.1 Object Model

Legion is composed of independent, active objects; all entities of interest within the system—processing resources, storage, users, etc.—are represented by objects [7]. Legion objects communicate via asynchronous method invocations supported by an underlying message passing system. Each method invocation contains a set of explicit (i.e., actual) parameters, and an optional set of arbitrary implicit parameters, attribute-value pairs that are available to called objects as invocation metadata. Method calls can produce an arbitrary set of results. Using data-flow information encoded in Legion method invocations, results are forwarded directly to where they are needed, rather than necessarily back to the caller. Objects are instances of classes that define their interface, which is required to be a superset of a minimal object-mandatory interface. Object mandatory methods include functions such as an interface query and methods to implement object persistence.

Legion objects are persistent and are defined to be in one of two states: active or inert. When an object is active, it is hosted within a running process and can service method invocations. When an object is inert, its state (called its Object Persistent Representation, or OPR) is stored on a persistent storage device managed within the system. Objects implement internal methods to store and recover their dynamic state.

For the purpose of communication, every object is identified by a unique,
location-independent Legion Object Identifier, or LOID. LOIDs consist of a variable number of variable length binary fields. Some LOID fields are reserved for system-level identification purposes, e.g., one field is used to identify the object's class, and another contains an instance number for the object (unique within the object's class). Additional fields can be used to contain other information about the object, for example, location hints or security information such as the object's public key.

2.2 Core Objects

Within this general object model, Legion defines the interfaces to a set of basic classes that are fundamental to the operation of the system, and that support the implementation of the object model itself. The most important core Legion object classes for the purposes of this discussion are Host Objects, Vault Objects, and Class Manager objects.¹

Host Objects in Legion represent processing resources. When a Legion object is activated, it is a Host Object that actually creates a process to contain the newly activated object. The Host Object thus controls access to its processing resource and can enforce local policies, e.g., ensuring that a user does not consume more processing time than allotted.

Vault Objects in Legion represent stable storage available within the system for containing OPRs. Just as Host Objects are the managers of active Legion objects, Vault Objects are the managers of inert Legion objects. For example, Vaults are the point of access control to storage resources, and can enforce policies such as file system allocations.

Hosts and Vaults provide the system with interfaces to processing and storage resources. The use of these interfaces is encapsulated by Class Manager Objects. Class Managers are responsible for managing the placement, activation, and deactivation of a set of objects of a given class. They provide a central mechanism for specifying policy for a set of like objects. Policies set by the Class Manager include defining which implementations are valid for instances, which hosts are suitable for execution of instances, which users may create new instances, and so on.

In addition to setting policy for instances, Class Managers serve as location authorities for instances. To accomplish message passing in Legion, LOIDs must be bound to low-level object addresses (typically an IP address plus port number). This binding process is supported by Class Managers, which maintain a record of each instance's object address. The binding process also serves as an automatic

¹In many of the cited Legion references, Class Manager Objects are referred to simply as “Class Objects.”
object activation mechanism in Legion: if a binding request for an inactive instance is received, the Class Manager automatically activates the referred-to object so it can service the pending method. The Legion message system defines a rebinding mechanism that is automatically invoked when bindings become stale, for example due to object migration.

Class Managers are first-class objects themselves, and are thus managed by higher-level Class Managers known as Meta-Class Managers. Meta-Class Managers are in turn managed by yet higher-level managers and so on. The recursion for a given Legion domain halts at a logically central Class Manager known as a Legion Class Manager. Within a domain, the location of the Legion Class Manager is well-known to ensure that the recursive binding process will terminate. A complete Legion system can be composed of any number of such hierarchies; inter-domain binding traffic is automatically forwarded among the cooperating Legion Class Managers, as depicted in Figure 1.

![Diagram of Legion Class Manager hierarchies](image)

Figure 1: Legion Class Manager hierarchies depicted for a system consisting of two Legion domains. Solid arrows represent the "instance of" relationship.

A critical aspect of the Legion core object classes is that they define interfaces, not implementations. The Legion software distribution provides a number of default reference implementations of each core object type, but the model explicitly enables and encourages the configuration, extension, and even replacement of local core object implementations to suit site- and user-specific functionality and performance requirements. For example, by replacing the implementation of the Host Object, a site can define arbitrary mechanisms and policies for the usage of their computational resources.
2.3 Legion Runtime Library

The implementation of Legion objects, including the core object types, is supported by a Legion Runtime Library (LRTL) interface. The LRTL provides services analogous to a traditional Object Request Broker (ORB), defining the interfaces to services such as message passing, object control (e.g., creation, location, deletion), dynamic invocation construction, distributed exception handling, and other basic required mechanisms.

A critical element of the LRTL is its flexible, configurable protocol stack [12]. All of the processing performed during the construction and execution of invocations on the object-caller side, and in the receipt, assembly, and service of invocations on the object-implementation side, is configured using a flexible event-based model. This feature of the Legion software allows tool builders to provide drop-in protocol layers for Legion object implementations in a convenient fashion. For example, adding message privacy through a cryptographic protocol is simply a matter of registering the appropriate message processing event handlers into the Legion protocol stack—the added service is transparent to the application developer.

2.4 Security Principles

Thus far, the presented architecture has specified neither security mechanisms nor policies. This loose specification is intentional—Legion is intended to provide a framework suitable for implementing the widest possible range of application-level and resource-level security mechanisms and policies, without dictating any globally required mechanisms or policies. It is our goal to enable sites to expose their resources to Legion in a manner compliant with their local policies. Similarly, it is our goal to enable application programmers to achieve desired, application-specific trade-offs among type, level, and cost of security policies and mechanisms.

This said, it is also our goal to provide practical metacomputing software that is extremely easy to use in the common case, both for system administrators and application developers. Towards this end, we have designed and implemented a number of fundamental security mechanisms within the Legion framework. We also embed a number of simple, conservative default policies in the system software that we expect will be useful in the common case. The default policies can always be overridden with ease, and thus do not detract from the flexibility of the system. However, in our experience, many users are only willing to execute software “out of the box” with minimal configuration effort. We want Legion to support these users safely and effectively as well.

We note that the description of the Legion architecture provided in this section is only a quick overview of the elements necessary for a discussion of Legion secu-
rity. Complete details of the Legion architecture and implementation are described in other sources, e.g., [4, 7].

3 Implementation and Default Policies

In Section 2 we described the basic Legion system architecture. Within this architecture we have designed and implemented a set of basic security mechanisms and policies. In this section we describe these basic, default Legion security mechanisms, concentrating on the types of security properties these mechanisms enable within the abstract Legion architecture.

3.1 Identity

Identity is fundamental to higher-level security services such as access control. In Legion, identity can be based most naturally on LOIDs, since all entities of interest (including users) are represented as Legion objects. As a default Legion security practice, we use one of the LOID fields to contain security information including an RSA public key.

By including the public key in an object’s LOID, we make it easy for other objects to encrypt communications to that object or to verify messages signed by it. Objects can just extract the key from the LOID, rather than looking it up in some separate database. By making the key an integral part of an object’s name, we eliminate some kinds of public key tampering. An attacker cannot substitute a new key in a known object’s id, because if any part of the LOID is altered, including the key, a new LOID is created that will not be recognized by Class Manager Objects during the binding process, and so on. One drawback, though, is that there is no mechanism for revoking an object’s key and issuing a new one, as this step implies a complete change of the object’s name.

An object normally gets its LOID from its Class Manager when it is created. The Class Manager assigns a new instance number to the object and creates its keys. The resulting LOID and keys are communicated over an encrypted channel to the Host Object on the machine where the object will run. Once the object is up, the Host Object passes its LOID and keys to it over a temporary socket connection. With a LOID in its possession, the object can now begin communicating with other objects using normal Legion mechanisms.

Certain objects are not created by Host Objects and get their LOIDs in different ways. Command-line Legion programs create their LOIDs and keys themselves. The instance number is chosen at random, and the new LOID is registered with a special command-line Class Manager for the domain. The private key is never
transmitted. Another special case is the core system objects that are necessary to bootstrap a Legion domain. These have their LOIDs and keys generated by a special domain initialization program.

Users also have LOIDs. A user creates his own LOID, which is then registered with a user Class Manager and entered in appropriate system groups and access control lists by the respective administrators. When an object such as a command-line program calls another object on behalf of the user, the user's LOID and associated credentials provide the basis for authentication and authorization. The ownership of a LOID resides in the user's unique knowledge of the private key that is paired with his LOID. The private key is kept encrypted on disk, on a smart card, or in some other safe place.

Although LOIDs serve as ids in Legion, they are not easily manipulated by people. The same service objects in different Legion systems will have different LOIDs, making it hard to write utility programs based on raw LOIDs. To solve these problems, Legion provides a directory service called context space that maps string names to LOIDs. A context contains string entries which may be linked to any kind of Legion object. Objects in a context may be files, hosts, users, etc., as well as other contexts. Context space is similar to a Unix file system; the contexts resemble directories where all the entries are soft links.

Contexts make it much easier to identify services and objects in a Legion system. However, this convenience also introduces some risks. Once objects rely on contexts to look up services, the focus for an attacker becomes the contexts themselves. If an attacker compromises a context, he can replace the LOIDs of valid objects with LOIDs of his own. There is nothing new about this type of vulnerability (an attacker who gains root access on Unix can easily replace system programs, for example), but it points out that LOIDs and their integral public keys do not protect against spoofing.

3.1.1 X.509 Certificates

The default security field of a LOID is more than just an RSA public key. It is actually an X.509 certificate that contains the key. In general, an X.509 certificate pairs a public key with a person's name, organization, identification of the public key algorithm, and other information. A certificate may be signed by a certification authority (CA) that vouches for the association of the key with the identifying information. To cover the case where a recipient doesn't recognize the CA, the CA's own certificate can be chained onto the certificate, allowing the CA's CA to be the basis of authority. The chain can have multiple links, each link generally leading to a higher authority CA. A recipient validates a certificate by traversing the chain until it reaches a CA it recognizes (for example, Verisign or the U.S.
Postal Service) and checks that all the intervening certificates are properly signed. Validators are free to put constraints on how deep a chain they will accept, who they will or will not trust, etc.

By default, each user in a Legion system has a signed X.509 certificate. If an organization installing Legion does not currently use X.509 certificates or endorse a particular CA, the Legion administrator can set up his own certification authority for Legion users. Some CA named in the CA chain on user certificates should be recognized by all the potential validators in a Legion system, but this is not a requirement; objects can have their own policies for handling method calls that include certificates they can’t validate.

The user’s X.509 certificate is propagated with requests and method calls made directly or indirectly on behalf of the user. The information in the certificate is used when making entries to access logs. It can also be the basis for alternative access control mechanisms, which we will discuss later.

All normal Legion objects also have X.509 certificates containing their public keys. However, the name fields of these certificates are empty, and the certificates are left unsigned. The main use of X.509 for Legion objects is to encode public keys in a standard way.

### 3.1.2 Credentials

For a resource, the essential step in deciding whether to grant an access request is to determine the identity of the caller. If a user communicates directly with the target object, he can establish his identity relatively easily with an authentication protocol, which typically involves performing an operation that only someone in possession of his private key can do. In a distributed object system, however, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf (for example, a user does not invoke the services of a Host Object directly, but instead relies on a Class Manager to use Host services). Though intermediate objects could in principle be given the user’s private key, the risk involved is too great. Given the user’s private key, an object can do anything the user can. That’s more privilege than is usually necessary.

To avoid the need for sharing the private key, we could have resources call back to the user or his trusted proxy when they receive access requests in the user’s name. This step puts control back in the user’s hands. There are several drawbacks to this approach, though. First, the fine-grain control afforded by authorization callbacks may be mostly illusory. It can be very difficult to craft policies for a user proxy (or even the real user himself?) that are much more than “grant all requests”—too much contextual and semantic information is generally missing from the request. Beyond this barrier, callbacks are expensive and do not scale well. Though there
is nothing in the Legion architecture that precludes using callbacks for particular objects or resources (and in some cases they may indeed be appropriate), calling back for authorization is not a universal solution. In Legion, after all, every object represents a resource of some type, and a callback on every method call would be a crippling performance hit.

The intermediate solution between these approaches is to issue credentials to objects. A credential is a list of rights granted by the credential’s maker, presumably the user. They can be passed through call chains. When an object requests a resource, it presents the credential to gain access. The resource checks the rights in the credential and who the maker is, and uses that information in deciding to grant access.

There are two main types of credentials in Legion: delegated credentials and bearer credentials. A delegated credential specifies exactly who is granted the listed rights, whereas simple possession of a bearer credential grants the rights listed within it. A Legion credential specifies:

- The period the credential is valid
- Who is allowed to use the credential
- The rights—which methods may be called on which specific objects or class of objects

If missing, fields default to “all.” The credential also includes the identity of its maker, who digitally signs the complete credential.

A sample delegated credential is “[Object A may call object B’s method M as Alice during the period T] signed Alice.” To use this credential, A must authenticate to B when it makes its request. We don’t have to worry about protecting the credential from theft, because only A can use it. Moreover, the specification of the target object, the method to be called, and the timeout closely limit how this credential can be used. Greater specificity lowers the risk of giving away rights that can be misused by other parties.

It is not always possible to specify a credential so narrowly. Call chains can be long, and the identity of the final object making a resource request may be unknown. If the call chain branches out, several different objects from different classes may need to make calls on the user’s behalf. We can loosen the specificity of credentials to handle these cases, but risk increases at the same time. The credential “[The bearer has all of Alice’s rights forever] signed Alice” is very convenient to give to objects, as there is no danger of accidentally restricting any actions, but should the credential be stolen, Alice is in trouble.

In Legion, tools or commands directly executed by the user create the credentials they need to carry out their actions. The credentials are made as specific as
possible. For example, if the user executes the command to create an object instance, that command will create a credential that authorizes the specified Class Manager to create an object instance on a Host Object. Of course, the Host Object that is contacted might reject the object creation request, but this rejection would be because the user was not authorized to use that resource, not because the Class Manager lacked the authorization to act on the user's behalf.

If a long-lived bearer credential is known to be stolen, the recovery strategy is to create a new LOID for the user. The new LOID has a different instance number but reuses the user's official X.509 certificate for its security field. The resource providers must then modify the appropriate system groups and replace the user's old LOID with the new one. The user's old objects will need new credentials to access resources, and we will see shortly how they can get those. The flaw, though, is in having long-lived bearer credentials to begin with.

3.1.3 Credential Refresh

The primary reason for limiting the duration of credentials, particularly bearer credentials, is to limit the period during which a credential is vulnerable to theft or abuse. If credentials expire too rapidly, however, valid objects will not be able to use them for their intended purposes. Class objects in particular must be able to hold credentials for long periods of time, so that the objects they manage can be reactivated without user intervention.

In Legion, we establish a balance between these conflicting goals. By default, we make credentials expire relatively quickly (for example, after some portion of an hour). A holder of an expired credential can get a fresh one by contacting a special Credential Refresh Object owned by the user. The Refresh Object then hands back an equivalent, fresh credential.

The purpose of the Refresh Object is to provide a single point of policy for handling credential revocation. If a security breach is suspected, there are a number of possibilities:

- The Refresh Object can stop renewing credentials issued before the security breach was discovered, so that all prior credentials (including those held by the attacker) will time out relatively quickly.

- The Refresh Object can log information about refresh requesters, or require authentication information.

- The Refresh Object can grant refreshes on some kinds of credentials but not others. For example, general bearer credentials may not be refreshable, or only refreshable once.
A Refresh Object needs the ability to sign credentials on behalf of the user. However, we don’t want to give it the user’s private X.509 key. That key should be treated very carefully, because handling its loss is likely to be expensive—the user's electronic identity would need to be revoked and reissued for all applications he uses it for. Normally the user’s private key will be kept in a form that requires the user’s direct participation to access. For example, it may be encrypted on disk or stored on a smart card.

Instead of using the X.509 private key, we generate a special public key pair when setting up a new Legion user for the first time. With his X.509 private key, the user then creates and signs a special proxy credential. The proxy credential specifies that regular credentials created with the new key pair are equivalent to credentials directly created by the user and his X.509 key. We call this key pair the proxy keys.

A Refresh Object is initially configured with the user’s proxy keys and proxy credential. When the Refresh Object receives a request for a fresh credential, it generates a new one using the proxy keys. The new credential is sent back along with the proxy credential. A resource validates a request by checking both of the credentials together.

Proxy credentials are long-lived; if they expired quickly, Refresh Objects would periodically need new proxy credentials from a potentially absent user. A proxy credential alone is worthless, however, and the normal credentials they accompany do time out, so the long life of proxy credentials is not a security problem.

If a user’s proxy keys are stolen, the attacker can easily create new credentials in the user’s name. Recovery is the same as if a long-lived bearer credential had been stolen: The user must change his LOID. To reduce the risk of theft, the Refresh Object should be assigned to a Host Object and Vault Object trusted by the user and less likely to be compromised.

The name of a user’s Refresh Object is sent along as an implicit parameter in method calls made on the object’s behalf. The Refresh Object is essential for objects that may need to hold a user’s credential for a long period of time. For example, a Class Manager may need to reactivate a user’s object multiple times during the object’s lifetime, which may be months or even years. The Refresh Object allows the Class Manager to get the credentials it needs even if the user is not available.

Though in normal use calls to the Refresh Object should be at a relatively low rate, it is possible that it could become a hot spot and affect scalability. There is nothing that prevents a user from having multiple Refresh Objects if load becomes a problem. However, there is an increased level of risk in having more outstanding copies of proxy keys. One mitigation strategy may be to have multiple sets of proxy keys and to break Refresh Objects into different trust categories. Less
trusted Refresh Objects might be given proxy keys and credentials that only allow refreshing of delegated credentials, not bearer credentials.

3.1.4 Command-Line Credentials

Command-line Legion programs also generate credentials on behalf of the user. However, we do not want them to use the user’s X.509 private key. Even if the commands are interactive and we can assume the user is present, the X.509 private key is too inconvenient to use. For example, we can’t have every Legion command prompt for a pass phrase so that it can decrypt the private key.

Just as with Refresh Objects, we use proxy keys and proxy credentials for command-line objects. The private proxy key is stored in the clear on disk, though protected by the local file system. The proxy credential is also available on disk so that commands may send it with the new credentials they generate. Simply by gaining access to his files, the user is “logged on” to Legion.

Another mechanism is also available if storing proxy keys on disk is considered too risky. To start using Legion, the user runs a special login tool that generates new proxy keys and credentials (and thus temporarily requires access to the user’s X.509 private key). This login tool stores the keys in memory and creates a new subshell for executing Legion commands. The commands use a private socket to obtain the proxy keys from the parent login program. When the subshell is exited (i.e., the user “logs out”), the keys and credentials are discarded, though they may live on in the user’s Refresh Object.

3.1.5 Authentication Credentials

To use a delegated credential that it holds, an object needs to authenticate itself to the target object. It does this by sending an authentication credential with the call. The authentication credential is the LOID of the target and is signed by the object. It is sent directly to the target and protected from theft en route. By “protected from theft” we mean that it cannot be extracted and combined with any other method call (how the communication layer handles this is discussed in Section 3.3). This protection is necessary because anybody holding both the authentication credential and the delegated credential can access the target, and the delegated credential may be known by many other parties.

By making the the authentication credential hold the target’s LOID, we ensure that the target object itself cannot misuse the authentication credential. Otherwise, the target might use the authentication credential along with some other credential delegated to the object to gain access to another resource.
3.1.6 Key Sharing

Public key pairs are expensive to generate. This expense is particularly noticeable for interactive commands, which have to create LOIDs for themselves, and for programs that create large collections of slave objects to carry out parallel computations.

We can largely eliminate this expense by sharing key pairs over collections of objects. The LOIDs are still distinct because they have different instance numbers, but the security fields are the same. In the case of command-line programs, we can simply use the proxy keys for the command-line objects' private keys and LOIDs. For other objects such as MPI slaves, the responsible Class Manager can generate one public key pair and use it for all the objects it creates.

The drawback of sharing keys is that if the private key of one object is stolen, all of its partners are immediately compromised as well. By only sharing a key within applications or for an interactive session, this risk is reduced. Otherwise, Class Managers can continue to generate new keys for each new object they create, and can improve performance by pregenerating a pool of keys when idle.

3.2 Access Control

In Legion, access is the ability to call a method on an object. The object may represent a file, a Legion service, a device, or any other resource. Access control is not centralized in any one part of the Legion system. Each object is responsible for enforcing its own access control policy. It may collaborate with other objects in making an access decision, and indeed, this allows an administrator to control policy for multiple objects from one point. The Legion architecture does not require this, however.

The general model for access control is that each method call received at an object passes through a MayI layer before being serviced. MayI is specified as an event in the configurable Legion protocol stack [12]. MayI decides whether to grant access according to whatever policy it implements. If access is denied, the object will respond with an appropriate security exception, which the caller can handle any way it sees fit.

MayI can be implemented in multiple ways. The trivial MayI layer could just allow all access. The default LRTL implementation provides a more sophisticated MayI that implements access control lists and credential checking. In this MayI, access control lists can be specified for each method in an object. There are two lists for each method, an allow and a deny. The entries in the lists are the LOIDs of callers that are granted or denied the right to call the particular method; a deny entry supersedes an allow. Default allow and deny lists can be specified to cover
methods that don't have their own entries.

The LOIDs in the allow and deny lists may specify particular users, the object's Class Manager, or the object itself. The lists can also include a special token that represents any LOID at all. The LOIDs of objects used to represent groups can also be contained in the lists. Group Objects simply represent a list of member LOIDs, providing methods for querying or modifying membership. Any user in the system can create their own group, listing whichever LOIDs they wish as members, and modifying membership dynamically over time. When a group object LOID is found on an access control list, all of the contained members are logically added to the list. For performance, the results of the membership lookup are cached, but with a short timeout (five minutes by default) so that group membership changes will be reflected relatively quickly. Groups provide one means for centralizing access control policy.

When a method call is received, the credentials it carries are checked by MayI and compared against the access control lists. For example, in the case of a delegated credential, the caller must have included proof of his identity in the call so that MayI can confirm that the credential applies. Multiple credentials can be carried in a call; checking continues until one provides access. Note that credentials provide an alternative way to define groups. If the group owner alone is on the access control list for a method, then he can give delegated credentials to all the members of the group, allowing them to call the method as well.

The default library MayI is configured when an object starts up. The configuration information is passed to it by its Class Manager, which in turn may have inherited the information, or part of it, from the user. For example, the user may have a default access control list for object-mandatory methods that all objects created on his behalf will inherit, while the Class Manager for those objects may specify additional access control lists specific to the particular kinds of objects they manage.

The form of access control provided by the default MayI is sufficient for some kinds of objects, such as file objects, but not for others. For example, Class Managers support a "deactivate" method that allows the caller to bring down an object managed by that Class Manager. Multiple clients of a single Class Manager Object may all need to call this method, but each should be allowed to deactivate only the objects he created. The default MayI doesn't have this ownership information. To solve this particular case, an additional MayI event handler is added to the Class Manager implementation that can check the arguments of the deactivate call against an internally maintained table of who created which object instance. The LRRL configurable, event-based protocol stack makes it easy to replace or supplement the default MayI with extra functionality such as this. The default MayI itself is relatively simple to modify if, for example, new forms of credentials or different
kinds of access control lists must be supported. With the Legion security architecture, these types of changes can be made on a local basis without affecting other parts of a Legion system.

3.3 Communication

A method call from one Legion object to another can consist of multiple Legion messages. Because Legion supports dataflow-based method invocation (as described in Section 2.1), the various arguments of a method call may flow into the target as messages from several different objects. The messages themselves are packetized and transmitted using one of a number of underlying transport layers, including UDP/IP, TCP/IP, or platform-specific message passing services (e.g., user-level message passing over an IBM SP2 switch).

It is at the level of Legion messages that we provide encryption and integrity services. When a Legion message is prepared for sending, various event handlers internal to the object are triggered in succession, as described in Section 2.3. One of these handlers implements a default message security layer. This layer inspects the implicit parameters accompanying a message to determine which security functions to apply.

3.3.1 Message Security Modes

A message may be sent with no security, in private mode, or in protected mode. In both private and protected modes, certain key elements of a message (e.g., any contained credentials) are encrypted. The functional difference between the two modes is in how the rest of the message (body plus other implicit parameters) is treated. In private mode it is encrypted, whereas in protected mode only a digest is generated to provide an integrity guarantee. Unless private mode is already on, protected mode is selected automatically if a message contains credentials. This is a failsafe measure to prevent credentials from being transmitted in the clear.

The purpose of encrypting credentials is to protect bearer credentials from theft. Delegated credentials do not need to be protected, but the security layer does not examine the credentials at this level of detail. Moreover, the distinction between the two types of credentials can be misleading: If a delegated credential grants rights to a large group because further specificity is impossible, it may be desirable to protect it just like a bearer credential.

In addition to protecting credentials, both protected mode and private mode encrypt a computation tag contained in every Legion message, a random number token that is generated for each method call. All the messages that make up a given method call contain the same computation tag. The tag is used to assemble
incoming messages from multiple objects into a single method call and to identify the return value for a call made earlier. If an attacker knows the computation tag for a method call, he can forge complete messages containing arguments or return values, even without holding any credentials. The computation tag is treated as a shared secret, and is never transmitted in the clear unless "no security" mode is selected.

Private mode works by RSA-encrypting the entire message using the recipient’s public key. For efficiency, the RSA toolkit (RSAREF 2.0) only encrypts a random DES session key using RSA encryption; this key is then used to encrypt everything else in cipher-block-chaining (CBC) mode. The use of DES in CBC mode ensures that the message cannot be broken into pieces and recombinated with other cryptotext to create a new valid message. For further efficiency, the sender and receiver cache the DES key so that if another message is sent in the same direction within a limited period, the DES key can be reused. The slow RSA encryption and decryption of the DES key can then be skipped.

Protected mode functions differently. First, a digest for the entire message is generated. Then, just the credentials and computation tag in the message are encrypted for the recipient. On receipt, they are decrypted, and the message digest is recalculated and compared with the transmitted value. An attacker cannot steal the encrypted block and use it to create another valid message because he is unable to create a digest that includes both his plaintext and the plaintext of the encrypted block. The latter can only be extracted by the intended recipient.

Protected mode is faster than private mode on large messages because digesting runs faster than DES encryption. Both modes pay the cost of RSA-encrypting a DES key, however.

Because the mode in use is stored in implicit parameters, it propagates through call chains. For example, a user can select private mode when calling an object. The calls that the object makes on behalf of the user will also use private mode, and so on down the line. In some cases this propagation is not desired, such as when a class object requests object creation on a Host Object. The class object always uses private mode for this call so that the new object’s private key is not exposed. However, the implicit parameters passed in this call will become the new object’s implicit parameters, and it may not need to run in private mode. The security layer recognizes a special nonpropagating implicit parameter to allow the specification of security for just a single message.

The security layer does not provide mutual authentication. The sender can be assured of the identity of the recipient, because only the desired recipient can read the encrypted parts of the message. The recipient usually doesn’t care who the actual sender is; its decisions are based solely on the credentials that arrived in the message.
3.3.2 Replay

Although credentials and computation tags cannot be extracted from a message by an eavesdropper, he can still attempt to replay a message. To prevent replay, each message includes a large random number and a timestamp. A recipient only accepts messages whose timestamps fall within a window based on the recipient's current time, e.g., thirty minutes into the past and ten minutes into the future. Old messages may be replays, excessively delayed, or victims of skewed clocks; messages from too far in the future also indicate overly skewed clocks. In these cases the recipient sends an exception back to the sender.

Assuming the message is arriving for the first time, the recipient then calculates how long it must log the random message number based on the message's timestamp and its own time-checking window. If another message then arrives with the same random number during the log period, it is rejected. In essence, the message timestamps allow coarse-grained detection of replays, while the random numbers provide the fine grain.

3.4 Object Management

So far we have discussed security at the level of Legion objects and facilities. Fundamentally, though, Legion software runs on existing operating systems with their own security policies. It is therefore also critical that the implementation of the Legion object model ensure that extra-Legion mechanisms cannot be used to subvert higher-level security mechanisms. Similarly, it is important to ensure that Legion does not break local security policies at a site. These issues are fundamental aspects of the Legion object management implementation—the interface between Legion core objects that represent resources, and the local system interfaces that provide access to resources.

Legion encapsulates the management of computational resources and data storage with Host and Vault Objects, respectively. The Host Object receives requests to create objects and controls them with the authentication and MayI mechanisms discussed earlier. It spawns new processes to run objects, monitors resource usage, and enforces allocation limits by killing user objects if necessary. In a complementary role, the Vault allocates OPRs as directories on the local file system or on other storage media. Authenticated objects use these allocated directories to store their persistent state. The Vault can reclaim managed storage space as necessary.

The local system administrator is generally concerned with who can create processes on his system via Legion, what those processes can do, and who pays for their resource use. If there is a security problem, he needs a way to trace the responsible party. On Legion's side, there is a need to prevent user objects from
interfering with one another or with system objects (e.g., Host and Vault Objects), and to maintain the privacy of persistent state (OPRs). The latter is particularly significant because objects store their private keys in their OPRs.

The needs of Legion are common to any multi-user operating system, and our approach to providing them is to leverage off of existing operating system services. In the following sections we show how we meet these needs and also satisfy two types of local system requirements.

3.4.1 Process Control Daemon

Our general strategy for isolating Legion objects from one another is to run them in separate accounts on the host system. The accounts that can be used for this purpose fall into two categories. For those Legion users who happen to have accounts on the local system, objects can run on their normal user accounts. For other users, there can be a pool of generic accounts that are assigned for Legion use. The generic accounts usually have minimal permissions (e.g., no home directory, no group memberships, etc.). The local Host and Vault Objects also have their own accounts.

Object creation requests arrive at the Host Object as normal method invocations, and can thus be controlled using the standard Legion access control mechanism for methods. For each request, the Host checks the credentials against the user LOIDs and groups that are allowed to create objects on it. If everything is acceptable, it next selects an account for the new object to run in; depending on the credentials in the creation request and its local configuration, it may choose a local user account or one of the generic accounts. The accounts are subject to scheduling and resource control just like CPU time, memory usage, and so on; an object’s lease on an account, especially a generic account, is usually limited.

Before starting an actual process for the new object in the allocated account, the Host needs to change the ownership of the object’s directory from the Vault user-id to the newly allocated user-id. The location of the directory that will contain the new object’s persistent state is passed to the Host as part of the activation request (this location was obtained through a method on the local Vault performed by the object’s creator, likely its class). Ownership of this directory must be changed to both protect the object’s state from access by other objects (which will run under different user-ids), and to make the state accessible to the new object.

Finally, the Host needs to spawn the actual process that will execute the object on the appropriate account. To carry out this step, and to change ownership of the object’s persistent state, the Host requires access to some privileged operations. However, the Host does not execute with root permissions. Access to these required privileged operations is encapsulated in a Process Control Daemon (PCD) that
executes on the host, providing services to the Host Object in a controlled fashion. The PCD is a small, easily vetted program that runs with root permissions. It is configured only to allow access by the host account. Two of its key functions are to permit changing directory ownership and to create new processes on a designated account. The PCD limits the accounts for which this can be done to a set configured by the local system administrator. The set includes the generic Legion accounts and potentially the accounts of local Legion users.

As the PCD starts the object running, the Host logs an audit trail using the X.509 information for the user whose credentials accompanied the request. The audit trail provides essential information if the new object misuses local resources.

While the object is allocated its account, the Host Object can reactivate it as necessary via the PCD (idle objects may deactivate themselves, or their class object may deactivate them). If the object has exceeded its use of local resources, the Host can request that the PCD kill it directly. When an object loses or relinquishes its use of an account, the Host object uses the PCD to change the ownership of its persistent state back to the Vault Object. If the object is reactivated later on a different account, ownership of the state can be changed to the appropriate user-id. After an account is reclaimed, the PCD terminates all processes running on it and generally cleans it up.

The Vault's storage is of course a limited resource as well, and scheduling and account policies apply to it. A Vault can clean up an object's state in several ways. Normally, the object's class will inform the Vault that the storage may be reclaimed (e.g., after a class migrates an instance off of a Vault, or deletes an instance, it calls the Vault's "delete storage" method). If the Vault initiates the clean-up, it proceeds conservatively, checking with the class object if possible and perhaps archiving the state before deleting it.

The implementation just described is sufficient for some local system administrators. Legion authentication is used to determine who gains access to local resources, and the resources made available are also constrained to those usable from a limited set of accounts. Detailed logging provides accountability.

3.4.2 Non-Legion Authentication

One aspect of the previous approach which may be unacceptable at some sites is the use of Legion authentication mechanisms to control access to a host. For example, a site may require that only users with local accounts may access the system, and that those users must be authenticated by a locally adopted authentication system such as Kerberos [10]. Once authentication succeeds, though, normal Legion objects can be created. To make the discussion of how these requirements can be met concrete, we will use Kerberos as a sample authentication mechanism.
The Kerberos authentication protocol is fundamentally based on clients obtaining tickets for the use of services. Tickets are unforgeable tokens obtained from a distribution server through a protocol that involves the actual authentication of the user through password entry. To avoid requiring the repeated entry of passwords, a special Ticket Granting Ticket (TGT) is obtained by clients. This TGT is a credential that can then be used for a limited time to obtain further tickets that are required to access individual services. Clients can obtain specially marked TGTs that can be forwarded to proxies for use within a limited time period [9].

The use of these forwarded TGTs is the basis for employing Kerberos as an authentication mechanism within Legion. The TGT is sent in the implicit parameters of an object creation request to the eventual Host Object. The TGT is equivalent to a bearer credential, and it is treated as such by not being sent in the clear, etc.

The Host Object uses the TGT it receives to request a new TGT from the local Kerberos server. If authentication fails, it will not get a TGT. The new TGT grants access to a Kerberized version of the PCD. The Host proceeds to make an object creation request to the Kerberized PCD, supplying the local TGT and the name of the user’s local account along with the standard job request information (e.g., the executable path, the path of the object’s persistent state directory, etc.). The PCD performs the normal actions of changing the ownership of the directory for the object’s state and spawning the object.

The TGTs expire relatively rapidly, and the Host discards them immediately after use. However, to support the continued management of local objects in the absence of the object owners, the PCD will perform certain limited actions on behalf of the Host without a TGT. It can stop (i.e., kill) a managed object, it can restart the object, and it can switch the ownership of the object’s state directory back and forth between the Vault ownership and user account ownership. The PCD ensures that only accounts of users authenticated via Kerberos are used.

An important restriction covers object restarts. The PCD will only restart the same objects under the same accounts that it did when TGTs were presented. It will not restart an object that is already running (thereby creating multiple copies), and it keeps track of its children to prevent this. These conditions prohibit the Host, or anybody contacting the Host, from leveraging off a previously authenticated use of the PCD. Consider the example of a user object running on a Host. If this object becomes deactivated, and then is needed to service a method called by a user other than its owner, it will need to be reactivated without a TGT. The PCD will reactivate the given object only if it had been previously started on the Host using a valid TGT. The PCD will not start additional objects of the same class (i.e., additional processes running the same executable), nor will it start the given object under a different user-id than was originally selected. The effect is the same as if the object had never been deactivated, but had instead continued to execute. This retains
the level of access control required by site administrators: processes can only be effectively started by authorized users, and then only through the locally mandated authentication mechanism. For the purposes of long-lived objects, we simply extend this to support the temporary suspension of objects created by authenticated users, and the subsequent reactivation of these processes by other clients.

The services provided by Kerberos (e.g., obtaining forwardable user credentials) are available in other systems, such as the Secure Sockets Layer (SSL). In fact, both Kerberos and SSL can be called through a generic interface: the Generic Secure Service Application Program Interface (GSSAPI) [8]. By using GSSAPI, straightforward extensions to other systems such as SSL are possible.

4 Policy Examples

The Legion architecture presented in Section 2 is highly flexible, allowing the implementation of a wide variety of security mechanisms, important examples of which were described in Section 3. However, application developers and site administrators typically have higher-level policy specifications in mind when using software. The particular underlying mechanisms are less important, as long as the user can be assured that high-level policy requirements are being met. In this section, we consider illustrative examples of how the Legion system architecture and existing Legion tools can be organized to meet sample site and application policies.

4.1 Site-Local Policy Examples

4.1.1 Site Isolation

As described in Section 2.2, Legion systems can consist of multiple domains, each possibly in a different organization or trust domain. For example, consider the example of a Regional Health Organization Network. Such a system would benefit from Legion's ability to allow enhanced collaboration, information sharing, and so on. However, this system would certainly interconnect institutions in disjoint trust domains. It is often desirable to system administrators contributing resources to a larger metasystem to ensure that certain site-isolation properties are guaranteed. For example, consider a site that makes resources available to Legion. It is managed by a local Legion administrator, who we will call Admin. A perfectly reasonable policy that would likely be required by Admin would be as follows: no matter how subverted any external sites in the Legion system might be, no intruder can invoke methods on local Legion resources as Admin. Such a policy is clearly desirable, since Admin is likely to have administrative control over critical local resources: who can use which machine, and for how long; who can access which
locally stored OPRs; etc. The ability to invoke methods as Admin is tantamount to complete control of the local Legion software.

The desired isolation policy can be achieved through a number of straightforward safeguards enabled by the Legion design. First and foremost, the local Legion resources should be started as a separate Legion domain (as described in Section 2.2). All of the core objects managing the local site can be started and configured by Admin, resulting in no external trust dependencies on outside systems. Clearly, to achieve the desired functionality of a metacomputer, the local domain will then need to be connected to some set of external Legion domains.

After this link to the external (and untrusted) system is made, Admin must take further precautions to prevent subversion of the local site. While invoking methods with bearer credentials based on Admin's proxy credential, Admin must be sure that no messages are sent to off-site objects. If we assume that any external site may be arbitrarily subverted and malicious, we cannot risk passing Admin's credentials outside the local site—they might immediately be used to break the isolation policy. However, simply stating that Admin should not pass credentials off-site is not generally good enough—Admin might make a simple mistake that could break the policy, so we would like automated enforcement of this safety measure. This automated enforcement is simple to achieve in Legion: Admin simply uses a version of the LRTL with the flexible protocol stack configured with an extra event handler for the message-send event. If a message is inadvertently directed off-site that contains Admin’s credentials, the message is blocked and the event handler raises an exception. With this simple modification to Admin’s Legion environment, he can be assured that his credentials will not be dispersed to untrustworthy off-site objects.

Ensuring that Admin does not communicate with off-site objects has a desirable secondary effect. Since Admin cannot communicate with external, untrustworthy sites, he cannot place critical objects such as his Refresh Object on resources at these sites (see Section 3.1.3). This benefit extends to an array of potentially critical, but not necessarily obvious, resources. For example, suppose Admin maintains a local Group Object listing the set of users that are allowed to start objects on local resources. If this object were allowed to execute on an untrustworthy site, its contents could be modified by a malicious resource owner, and local resource usage policy could be broken.

The two mechanisms described above, in combination with carefully configured access control for local core objects such as Hosts, Vaults, and critical Class Managers, ensure that the desired isolation policy will be met. Off-site objects will neither be able to generate nor steal Admin’s credentials. External callers will be prevented from invoking unauthorized methods on local critical resources, ensuring that local access control is not tampered with, local resource usage policies are
not modified, and that security failures in other domains do not have dire consequences for the local site.

4.1.2 Site-Wide Required Access Control

The Legion access control model as presented in Section 3.2 is based on the assumption that users will configure access control for their own objects. This concept adds a powerful level of flexibility to the system—for example, it makes arbitrary site-local resource access policies possible. However, on first examination it appears to relinquish the ability for a system administrator to set access control policies uniformly across his site. For example, the default Legion access control configuration does not grant the administrator user for a Legion domain access to other users' objects within the domain—there is no root user who can read any file or use any program in the domain. The inability to configure required site-wide access control policies may be unacceptable at some sites. However, the flexibility of the Legion architecture allows us to address this issue in a straightforward fashion, using the existing tools provided with the Legion software.

As an example of a site-wide required access control policy, we consider the problem of strictly limiting access to files by outside users. The Legion system defines a basic File Object that can be used to represent a file in the system. Access control for the normal Legion File Object is based on the default Legion ACL MayI mechanism, which places no restrictions on what LOIDs (i.e., what users) may be placed on access control lists. However, consider a site that wishes to enforce the policy that files may not be accessed by outside users. Effectively, we want a way to control which LOIDs may be placed on the ACL for local file objects. We can achieve this policy using the power of local Host Objects to control access to local resources. The Host Objects at the site (which are owned and controlled by the local administrator) are a point of resource access policy—they define which types of objects may run at the site. Using this feature, the site administrator can strictly limit the classes of objects that may run at the site. In particular, the allowable set of classes can be limited to those that are approved by the system administrator. The list of allowable classes can be configured to only include file objects with an alternate MayI layer—an extended version of the default ACL mechanism that also verifies that allowed LOIDs are in a well-known group containing only the local site users. Given this simple configuration, the site administrator can ensure that files are not inadvertently exported to outside users through Legion.\footnote{Of course, the described approach does not prevent malicious users from exporting data off site in any number of ways, through Legion and otherwise.} Furthermore, this approach generalizes to other site-wide access control restrictions, and other similar site-wide policy enforcement problems.
4.1.3 Firewalls

Firewalls are a simple fact of life at many security-conscious institutions. While firewalls are not addressed explicitly in the Legion model, the Legion architecture is flexible enough to accommodate firewalls with ease. As is typical in firewall situations, a proxy on the firewall host is the natural solution. However, the ability to use custom versions of the Legion core objects, and the flexible protocol stack model of the LRLT, allow proxy-based solutions to be employed in Legion in an especially straightforward, user-transparent way.

Objects started on hosts behind a firewall automatically have a Proxy Object on the firewall host assigned to them by their Host Object (in some cases, each user might desire their own proxy object; in other cases, a shared proxy object is acceptable; either model is simple to support). The object address for a newly activated object behind the firewall that is reported to the object's Class Manager is actually the address for the Proxy Object—when callers of the object execute the binding process, they will be given the address of the Proxy Object. The Proxy Object then acts as a simple reflector, forwarding any received messages to their intended destinations behind the firewall. Use of the Proxy Object to forward outbound messages from callers behind the firewall is automated by a transparent add-in event handler in the LRLTL protocol stack.

4.2 Application Policy Examples

4.2.1 Resource Selection Policy

In principle, a user of a metacomputer shouldn't need to care which resources are used to execute his jobs. In practice, however, the trustworthiness of the resources that are selected for certain applications is of critical interest to the user. Policies regarding which resources may be used to execute objects are logically localized within the Class Managers of a user's object classes. Any site selection policy can be encoded in a user's Class Manager Objects, giving the user total control over the selection and use of trustworthy sites.

Although this problem is solved cleanly at the architectural level in Legion, we deemed this issue of site selection for application users important enough to warrant special features in the default Class Manager Object reference implementations. All default Class Managers in the Legion implementation check for certain implicit parameters that can be used to limit resource selection. By setting these implicit parameters in his Legion environment (using a provided tool), the user can configure a resource selection policy that will propagate to all "create instance" methods called on Class Manager objects on behalf of the user. Of course, the architectural principle that users can encode any resource selection policy they
wish in their own Class Manager implementations still holds; in fact, a convenient model for such customization is supported by the default Class Manager's ability to be configured to use an external Scheduler Object with a well-known interface. However, in the common case, where a user can generate a list of sites that he deems trustworthy and indicate this in his environment, the default implementation provides the needed mechanism to implement a basic, effective resource selection policy.

4.2.2 Customized Access Control Policies

The default, ACL-based access control mechanism provided in the LRTL basic MayI implementation is useful for specifying many common access control policies. For example, the basic Legion File Object has methods such as read, write, and truncate. By specifying allow and deny lists of users and groups for these methods, we can achieve the traditional file access policies familiar to users of common existing file systems. This is also true for other Legion services, such as Context Space. However, in some cases, access control lists are not sufficient to specify the required policy. Consider the example of an object that represents a database of patient records in a hospital. Suppose that this database object has methods to create, query and update the record for a patient. We would certainly want all of the doctors in the hospital to have access to these methods. However, we might also want to enforce the policy that a patient’s record is only available to his health care providers. Access control lists do not let us express this policy.

Solving this problem is straightforward in Legion. Since MayI is an event in the configurable protocol stack, we can introduce a new MayI event handler to enforce the desired policy. The new MayI handler would check the record in question on query and update methods against the method caller (indicated as the signer of a credential granting access to the method in question). If the caller is listed as one of the doctors for the patient whose record was being accessed, the call would be allowed to proceed. Otherwise, MayI would reject the call and raise an exception.

In practice, we employ exactly this sort of add-on MayI functionality in the reference implementations of the Legion core objects. For example, Vault objects provide access to object persistent state. All users of a given Vault need to access some of the OPRs contained in the Vault, but we want to ensure that users can’t access one another’s OPRs. We use an extra MayI layer to ensure that only the appropriate object owners (or the owner of the Vault itself) can access OPRs. Similar functionality is used to control object management operations on Host Objects and Class Managers.
5 Related Work

Two projects that incorporate security into large-scale distributed computing platforms are Globus and WebOS. Globus [2] is a "bag of services" model for metacomputing, in contrast to Legion's integrated environment approach. Whereas Legion security is fundamentally built into the architecture of the system, Globus security services are provided as add-on modules. Other Globus toolkit modules vary in the degree to which they integrate with, or use the services of, the security modules. In Legion, we have adopted the approach of defining a set of simple but powerful abstractions that may be easily composed to implement new security policies, as our examples demonstrate. This approach is inherently more flexible and adaptable.

CRISIS [1] is the security architecture for WebOS. WebOS is fundamentally different from Legion in terms of the basic services provided. WebOS provides a single, traditional file system and a fixed interface for authenticated remote process creation. CRISIS defines careful, effective security policies for these basic services. However, the CRISIS solution does not provide a means for easily developing security policies for new mechanisms as they are added to WebOS, nor does it provide a means for modifying the security policies supported for the existing services.

Two other projects related to security efforts in Legion, although not with the focus on metasystems, are Java and CORBA. The computational model of Java [3] (JDK 1.2) requires identity and authentication in order to execute digitally signed code downloaded from a remote site. The JDK provides per-class (or per-application) protection domains. However, it differs significantly from Legion in its lack of support for per-site security mechanisms, delegation, and user authentication.

The security model of CORBA [11] encompasses identification and authentication, authorization and access control, auditing, security of communication, non-repudiation, and security information administration. Typically, an ORB vendor implements CORBA security using existing technology such as GSSAPI, Kerberos, and SESAME. Many of the goals of the CORBA security model are similar to the goals of the Legion security model, including simplicity, scalability, usability, and flexibility. However, CORBA is not a metacomputing system—it does not construct an operating system-like environment using underlying distributed resources. Given this fundamental difference in target use, CORBA does not address the metacomputing security problem.
6 Conclusions

We have presented the basic security architecture of the Legion system, and we have demonstrated that our design is sufficiently flexible to accommodate a wide variety of security-related mechanisms. This flexibility is critical to the successful deployment and use of metacomputing software. One-size-fits-all software dictated by a single group will never satisfy the requirements of the wide range of users and resource providers in a large-scale, cross-domain environment. We have also demonstrated that flexibility does not come at the price of complete lack of control. Within the flexible Legion framework, we showed how a number of important site-wide and application-wide security policies could be achieved. Naturally, the set of policies presented is only a small fraction of the policies that will be needed across the complete Legion environment.

The Legion system, including the security features described here, is currently publicly available. It is widely deployed on hundreds of machines at dozens of sites spanning multiple trust domains. Key portions of the software, such as the PCD described in Section 2.2, have been vetted and approved by system administrators at sites such as the San Diego Supercomputing Center and the US Naval Oceanographic Office (NAVO). In the future, we plan to continue deployment of Legion, developing additional mechanism and adapting to new site-local policies as required. We are also in the process of measuring the performance impact of key Legion security mechanisms.

References


The design of the current generation of desktop software technology differs from that of past generations in a fundamental way. The new paradigm states that applications should be built by composing off-the-shelf components, much as hardware designers build systems from integrated circuits, and that, furthermore, these components may be distributed across a wide area network of compute and data servers. Components are defined by their public interfaces, which specify the function as well as the protocols that they may use to communicate with other components. In this model, an application program becomes a dynamic network of communicating objects. This basic distributed-object design philosophy is having a profound impact on all aspects of information-processing technology. We are already seeing the software industry move away from handcrafted, standalone applications and toward investment in software components. A technology war over the design of component composition architecture is being fought within the industry.

High-performance computing cannot remain immune to this paradigm shift. As the Internet continues to scale in both size and bandwidth, it is not unrealistic to imagine applications that incorporate 10,000 active components distributed over 10,000 compute hosts. Furthermore, pressure from the desktop software industry will eventually lead to the integration of applications that currently run only on supercomputer systems into distributed problem-solving environments that use object technology. Computational grids that couple massively parallel processor (MPP) servers, advanced networked instruments, database servers, and gigabit networks will require a robust and scalable object model that supports high-performance application design.
This chapter is divided into two parts. In the first half (Sections 9.1 through 9.3), we explore the concepts underlying current distributed-object and component system architectures. We describe the basic features of the designs that, for now, constitute the standards of the desktop software industry. As good as they are, however, the designs fall short of what is needed for a high-performance national grid object system. By looking at three of the applications described in other chapters of this book, we can extract the requirements that software component middleware must meet in order to build these types of applications.

The second half of this chapter (Sections 9.4 through 9.6) focuses on the large-scale architecture of a complete grid software architecture, based on object-oriented design concepts and using the Legion system as an example.

9.1 BASIC CONCEPTS

Before discussing any applications, we should define some of the terms used in the chapter to classify computation and communication types. The key ideas in object-oriented software design, and in this discussion, are as follows:

- Data and the functions that operate on the data should be bound together into objects. These objects are instances of an abstract data type called a class. The data associated with an object are called data members, or attributes, and the functions that are associated with a class of objects are called member functions.

- Interfaces describe a set of functions that can be used to interact with a family of objects. Those classes of objects that respond to a particular interface are said to implement that interface. A class may implement more than one interface.

- A new object class may be built from an existing class by adding new data attributes or member functions. Instances of the new class each contain an instance of the original (parent) class and thus can implement the same interfaces as the parent. This process of extending one class to build another is called inheritance: the extended class is said to inherit from the original class. The extended class can override its parent class's definition of a member function and specialize or modify the parent's behavior (i.e., it responds to the same functions, but not in the same way).

- A new interface definition can also be created by simply adding new functions, thereby extending the definition of one or more other interfaces.
These object-oriented software design principles are only the first step in building the next generation of grid applications. As the desktop software industry has learned, it is also necessary to understand the process whereby an object class instance becomes a component in a distributed system. Component architecture is used to describe the framework for designing and using components, but it usually has two parts: components and containers. The architecture also defines a set of rules that prescribe the required features all components must support in order to be integrated into functioning applications. A container is an application that handles the integration of a set of components, or their proxies. We will discuss containers in more detail later.

Components often (but not necessarily) share the following characteristics: they are objects, they have persistent state, they have visual interfaces, they can be manipulated with graphical representations of component container toolkits, and they can communicate with other components—either locally or remotely—by one or more mechanisms (events, method invocations, procedure calls, message passing, etc.). The way in which a component presents a visual interface (if it has one), responds to events, and communicates with other components is defined by the component architecture. The three most important commercial component architectures are Microsoft ActiveX, OMG's CORBA/OpenDoc, and JavaBeans Java Studio. But, since our interest here is grid systems, not graphical user interfaces, we will focus on aspects of component systems that describe the composition and communication behavior of most component architectures.

There are two common models of component integration:

- **Client-server communication:** In this model a client is an application that acts as a container of components or their proxies. The application makes requests of objects by invoking the public member functions defined by the component objects' interfaces. The individual components are servers, which respond to the client as illustrated in Figure 9.1. The control flow is based on a function call from and return to the client. Microsoft ActiveX follows this model [117, 427]. CORBA was also designed with this model in mind, but as a distributed-object system it is flexible enough to support other models [424, 428, 427],

- **Software ICs:** An electronic IC is a component that has input buffers and output ports. A design engineer can connect any output port of the right signal type to an input port of another IC. Software IC systems have the same nature. A software module has input ports and output ports, and a graphical container or script-based composition tool can be used to create object instances and to define the connections between the components.
Client-server component models consist of a client container application, which holds object components that often act as proxies for remote objects. Such an architecture may support multiple protocols between the proxies and the remote components.

The input port's type is an interface describing the message that the port can receive. These ports can be connected to other ports, whose interface descriptions describe what type of messages are sent. As with electronic ICs, an output port's messages can be multicast to matching input ports on several other components, as shown in Figure 9.2. The control flow of messages is based on macro-data-flow techniques.

In addition to this data stream style of communication between object ports, there are two other standard forms of component system communication.

- **Control signals**: Every component implements a standard control message interface, which is used by the component control container framework to query other components about their properties and state.

- **Events and exceptions**: Events are messages generated by a component and broadcast to any other components that are "listening" for events of that type. Most user input, as well as other GUI management tasks, is handled by events.

Sun's JavaBeans and Java Studio systems follow the software IC component model closely. Other commercial systems based on this architecture include AVS and its descendent, NAG Explorer, which are used to build visualization tools from components. Unfortunately, Explorer has a limited and inflexible
A software IC component architecture breaks the client-server hierarchy. Each component has three standard modes of communication: data streams that connect component ports (solid lines), control messages from the component container (dashed lines), and events (star bursts), which are broadcast to all "listening" objects.

type system that restricts its extensibility to larger distributed applications. The CORBA-based OpenDoc system uses a similar object model.

The final piece of a component system architecture that distinguishes it from other types of software infrastructure is the concept of a component container framework. The container, an application that runs on the user's workstation, is used to select components, connect them, and respond to event messages. The container uses the control interface of each component to learn its properties and to initialize it.

Microsoft's Internet Explorer is an example of a component container for ActiveX. Java Studio provides a graphical user interface for composing and connecting components that is similar to the layout system used by Explorer and other component breadboards. We will return to more of the technical
requirements for high-performance components and container frameworks later in this chapter.

9.2 THREE APPLICATION SCENARIOS

Having defined the basic object-oriented software concepts, we can now look at three application case studies. Each case illustrates a different set of design requirements, but they all share certain features.

9.2.1 Example: Distributed Algorithm Design

An important class of grid-based programming environments is based on problem-solving environments (PSEs), software frameworks for integrating algorithmic modules into distributed scientific computations. As discussed in Chapter 6, these systems make it possible for a user to exploit the resources of the grid without having to deal with the complexities of low-level communication and resource management. These systems differ from generic component architectures by providing a high-level framework that allows users to approach a problem in terms of the application area semantics for which the PSE was designed.

SCIRun (see Chapter 7) is an excellent example of a PSE system architecture for scientific problem solving. Based on the NAG Explorer-style component composition model, SCIRun currently supports a small but powerful set of data types for communication, including mesh, fields, surfaces, and matrix types. However, the type system cannot in principle be extended to arbitrary types.

Two other distributed scientific PSEs include NetSolve (see Chapter 7) and WebFlow (Chapter 10). NetSolve is not based on a object component architecture, but uses a combination of a client-server model together with a novel approach to agent-based design. WebFlow is based on component design but is built on and derives its power and versatility from commodity Web technologies. Another example is the Linear System Analyzer (LSA), designed by Bramley et al. [222] (illustrated in Figure 9.3).

LSA was built to simplify the process of solving large sparse systems of linear equations. While many may consider the task of solving matrix equations to be a "solved problem," nothing could be further from the truth. This job remains one of the most difficult problems in most large-scale scientific simulations. The difficulty arises because no single method works for all problems and little theory exists to guide the user in selecting the correct method for a
9.2 Three Application Scenarios

9.3 Building distributed algorithms by composing components using LSA.

given problem. Furthermore, the most successful methods involve a combination of matrix preconditioning and iterative solvers or careful reordering and scaling and a direct solver. Bramley observed that the problem can be broken down into the following steps:

1. Read the matrix (or extract it from another part of a larger problem).
2. Analyze the matrix for obvious properties that help guide the solution process. For example, is it symmetric, banded, or strongly diagonally dominant?
3. Apply a reordering or scaling transformation, such as Markovitz pivoting or blocking.
4. Select and apply a preconditioner, such as MG, ILU, MILU, RILU, ILUT, or SSOR.

5. Select a solver from the many that are available, such as Direct, AMG, BiCG, CGS, Bi-CGSTab, GMRES, GCR, or OrthoMin.


In Figure 9.3 the reordered system is sent to a sparse direct solver (SuperLU) and an iterative library (SPLIB). LSA provides a library of components that implement these steps in the solution process. By connecting a matrix analysis component to a preconditioner that is connected to an iterative solver and a solution extractor, the user can build a custom solver for the problem at hand. The components can be linked to form a single library, which can then be added to a larger application. Suppose, however, that the best solver is located on a specific remote parallel machine or that the problem is so large that it can be solved only on a remote machine with a large memory. Since LSA allows components to be placed on remote machines by assigning a host IP address to that component, the underlying component container architecture works with the grid scheduler to make sure that the component is initialized and running at that location.

Requirements Imposed on the Component System

Note that LSA and other distributed algorithm systems impose special requirements that are not part of the conventional desktop software component model. Many of these will be common to all the examples in this book.

First, network quality of service and performance characteristics may deteriorate when a problem is distributed. Large-scale problems have large-scale bandwidth demands, but moving a large sparse matrix over a network link should not take longer than the combined execution time of the sending and receiving components. Otherwise, it may not make sense to distribute the computation. (There are important exceptions to this rule. For example, if the host system has special capabilities or the component objects are proprietary, it may be necessary to distribute the computation in spite of reduced performance.) None of the commercial architectures (ActiveX, CORBA, JavaBeans) has a standard model for associating performance characteristics or requirements with the communication infrastructure. Other network performance factors affect performance time and service as well, and they should be considered when examining component models. (Quality of service is a major concern; it is described in greater detail in Chapter 19.)
Second, scheduling the execution of a large distributed computation can be extraordinarily complex. For an application such as LSA, some components can execute interactively, while other components must wait in batch queues. Consequently, the synchronization between components must be flexible enough to allow the network of components to work asynchronously with long latencies.

Third, it is important to have a scripting interface to supplement the graphical composition model, since a network of components may be executed several times with different inputs and different parameter configurations. A scripting language such as Python or Perl will allow iterative control of the execution as well as composition of large graphs of components.

Finally, mixed language components are essential for linking scientific applications, such as linear algebra solvers, with Java-based graphical interfaces and component architectures. In LSA, approximately 40% of the system is Fortran plus MPI, 30% is Java, and 30% is HPC++ [221], which encapsulates the parallel Fortran and communicates with the Java front end.

9.2.2 Example: Teleimmersive Collaborative Design

Consider the following application of the teleimmersion environment described in Chapter 6. A car company uses a collaborative design system to reduce costs and time in its new product design process. For each new car, there is a master design database at the main factory, and each subcontractor maintains a separate design database with details about the components that they supply. Some of the information in the subcontractors' databases is proprietary and does not appear in the master design database, but it is possible for the master design database to extract any required performance information from a subcontractor by means of simple RPC transactions. These performance responses can be used to create a simulation of the car on a remote supercomputer at the company headquarters. The simulation results can be transmitted to teleimmersion systems at the main facility and at the subcontractors' facilities over a high-bandwidth network. The teleimmersion environment displays the car responding, in a virtual environment, to the user's control.

Suppose, though, that the designers want to see how an altered engine design affects the car's handling (Figure 9.4). The main designers would ask the engine builder to update the main database with the new engine model. A virtual mountain road scenario could be loaded and used to create a simulation. The designers could then interactively experiment with the new handling characteristics of the simulated vehicle.
9.4 Object-Based Approaches

Two CAVE environments connected to a distributed design database and a remote simulation facility.

This system has several components. The *teleimmersion environment* can be viewed as one large component, but it probably consists of several smaller components, such as the following:

- The data input stream consists of updates sent to the associated *visual database* and then rendered by the *display object*.

- As with any graphical user interface system, the *user interface control components*, which include pointers, head trackers, and other haptic devices, detect user events and then output the information as a stream to other components that are associated with the application.

- The *application components* receive information from the control device components. With this information the application components can query and update the visual database.

The *design database* is a description of the car and is used in the simulation and the manufacturing process. This database can also be viewed as a large component or as a collection of smaller ones. Output from this object includes
9.2 Three Application Scenarios

the polygon model used in the rendering component and the finite-element model used by the simulation.

The required inputs for the simulation object are the finite-element models of the car and the road, as well as a sequence of control inputs that "drive" the car during the simulation.

Requirements Imposed on the Component System

The application's most important aspect is its dependence upon managed bandwidth, which makes realtime performance possible. The need to manage many different types of data streams between components makes this dependence more complex. Currently, realtime CORBA implementations are being investigated in the research community [488], but the standard implementations of CORBA, DCOM (ActiveX), and Java communication mechanisms would be insufficient for the teleimmersion application. The object architecture must provide a mechanism that allows performance constraints and quality-of-service mechanisms to be associated with the logical data paths between component ports.

In addition, the application needs support for multicast communication in the object model. While it is likely that we will see extensions of Java RMI (Remote Method Invocation) to multicast, it is not part of the CORBA or ActiveX model. (More details about multicast communication and CORBA are found in Chapter 18.)

9.2.3 Example: The Digital Sky Project

The Digital Sky project (detailed in Chapter 5) illustrates a different set of distributed-object needs. Multiple databases contain billions of metadata objects, each of which describes a visible object such as a star or galaxy. Each metadata object is linked to a digital image in the archival system. The collection of metadata objects is organized as a relational database. In addition to a reference to the appropriate archived image object, each metadata object contains basic reference information and a list of data extraction methods that can be applied to the image object.

Suppose, then, that a scientist at some remote location decides to search for all galaxies that exhibit a particular set of properties. Some of these properties may relate to information stored in the metadata, but some may require an analysis of stored images. The request is formulated as a database query and sent to the database, where a set of objects that satisfy conditions associated with the metadata can be extracted. Then, for each of these objects, a
1. Apply initial relational select operation. Send message to archive to complete request.
2. Apply data parallel image analysis operation.
3. Send selected references and image objects to a second survey for more analysis.

The Digital Sky project couples a distributed collection of database and archive components. Heavy lines between components indicate communication channels that require high bandwidth.

request to apply the remaining tests to the images is sent to the data archive. This is a data-parallel operation and results in a set of references that satisfy all conditions to the subset of galaxies. The result of this query might then be used as part of a second query submitted to another remote repository. This may involve the transmission of a large stream of data from the first repository host to the second.

As shown in Figure 9.5, the components of the solution process are the relational databases and the repositories. To set up a complex analysis of the data, two or more components may need to be connected by a high-bandwidth link. The information that is communicated between components consists of image objects, object references, metadata information, and relational database queries.

Requirements Imposed on the Component System

While many of the problems associated with this application can be found in the preceding two examples, there are also some unique features in this particular problem.
The first is the extensive use of database technology. While there is a Java database interface standard, it may not scale to the problems described here. In particular, the interaction between the object-relational database and image archive requires the implementation of the data-parallel remote method invocation described above.

The second feature is related to the communications that must occur between components with parallel implementations. Existing commercial technologies require that a single logical channel be implemented as a single network stream connection. However, if both components have parallel implementations, it may be possible to implement the communication as a set of parallel communication streams. Pardis, a parallel implementation and extension of CORBA [307], is an example of a system that supports this feature. Pardis demonstrates that it is possible to significantly improve the utilization of network bandwidth by arranging parallel streams that can implement remote method calls.

We will return to this example again at the end of this chapter.

9.3 GRID COMPONENT FRAMEWORKS

To build a component-based, high-performance application, certain additional problems must be considered. First, objects in the framework need to know about each other in order to be able to transmit the data and member function messages as indicated. For example, a CAD database may be located in one city and a flow simulation may be running on a parallel processing system in another city, while the coupled visualization system may be an immersive environment such as a CAVE at yet another location. To complicate matters further, some objects, such as the design database, may be persistent, while other objects, such as the grid generation filter, may exist only for the duration of the computation.

One solution to this configuration problem would be a visual programming system that allows the user to draw the application component graph. NAG Explorer uses this technique, as does the LSA example described above and Java Studio. Unfortunately, NAG's type system is not very rich, and it is unclear whether graphical composition tools will scale to networks of more than a few dozen objects. We would also like to be able to describe networks that are dynamic and can incorporate new component resources as they are discovered. Systems such as Explorer and the current version of SCIRun use a fixed-type system, but most distributed-object systems allow arbitrary user-defined types to be transmitted over the channels between components.
9.3.1 Serialization Problem

A system must also know how to transmit application-specific objects over the network. This is called the serialization problem, and its solution requires a protocol for packing and unpacking data structure components so that they may be reliably transmitted between different computer architectures in a heterogeneous environment. The traditional solution is to use an Interface Definition Language (IDL) to describe the types of the objects being transmitted. IDL is a simple C++-like language for describing structures and interfaces and was first used in the DCE infrastructure [352]. The DCE IDL was adopted and extended for use in Microsoft DCOM and CORBA. The CORBA extension is the most complete, and it is used as the foundation of the specification of the entire CORBA system. Java RMI, on the other hand, is a strictly Java-to-Java communication model, so that Java serves as its own IDL. However, there is now a Java-to-HPC++ link that uses a combination of IDL and Java RMI, and JavaSoft has agreed to reimplement RMI so that it runs over the CORBA communication protocol known as IIOP. (The use of CORBA, Java, and DCOM in a commodity-based grid architecture is described in much greater detail in Chapter 10. Chapter 18 provides a good introduction to IIOP as well as other relevant protocols.)

9.3.2 Performance Issues

A persistent problem with existing commercial technologies is the poor performance of serialization and communication. Java RMI provides the most sophisticated serialization model, but its performance is several orders of magnitude below the requirements of the grid applications described above.

The Agile Objects project, at the University of Illinois, is exploring techniques for high-performance implementation of component object standard interfaces and protocols that focus on lowering the cost of crossing component boundaries (lower invocation overhead) and reducing the latency of an RPC (lower invocation latency). In particular, these efforts are focusing on Java RMI and DCOM invocation mechanisms and are building on technologies from the Illinois Concert run time, which executes RPC and message calls within a cluster of workstations in 10–20 μs. In addition, the Gigabit CORBA project [236, 490, 489] and the Indiana JAVA RMI-Nexus projects [80] are addressing the same problem in the case of heterogeneous environments.
9.3.3 Additional Issues

Most good object-oriented systems also include mechanisms for the following additional problems.

**Naming**

Any application-level programming framework must provide a mechanism that uniquely identifies the objects being integrated into a distributed computation. Naming mechanisms must be incorporated into the design of the system at a fundamental level.

**Persistence and Storage Management**

An object may need to be “frozen” so that its state is preserved on some storage device and “thawed” when it is needed. A system with this ability is said to support persistence, which is closely related to serialization as described above.

**Object Sharing**

If each object instance belonged to only one application, this would be problem enough, but when objects are used in multiple applications concurrently—a design database may be used simultaneously by several applications, for example—additional problems arise. To overcome these problems, the programmer can associate a session identifier with each circuit of objects, so that when an object receives a message, there is an accompanying session identifier. The identifier identifies which objects need to receive any outgoing messages associated with that transaction. An important related concept is that of collaboration: distributed applications, such as the teleimmersive design example, are often based on the ability of multiple users to share views and access to an object. The Infospheres system [114] is an excellent example of a component architecture that treats collaboration as a central design objective.

**Process and Thread Management**

Most instances of distributed objects are encapsulated within their own processes, but some situations require that multiple objects belong to the same process or that an object respond concurrently to different requests for the same method invocation. This capability requires that the object system be
integrated with a thread system. An important associated concern is that the thread model used to implement the communication and events for the component must be consistent with a thread model that might be used in the computation kernel. For example, an application that uses Fortran OpenMP may generate threads with one runtime system, but the component architecture may use another. These thread systems often have difficulties existing in the same process.

**Object Distribution and Object Migration**

An object implementation may itself be distributed. This is especially relevant to parallel programming, but it can occur in other situations, as when one part of a particular interface needs to be implemented on one system and another part on another system. An object may also need to migrate from one host to another, as when a host's compute resources are too limited and an object could be more efficiently implemented on a second, more powerful host.

**Network Adaptability**

As the examples described above illustrate, it is essential that the object middleware layer be able to adapt to dynamic network loads and fluctuating alternative pathway availability.

**Dynamic Invocation**

As described thus far, the interfaces to distributed objects must be known at compilation time, as must the IDL description used to generate the proxies/stubs and interface skeleton for the remote objects. However, a component system may be required to provide the mechanism for an application to discover these interfaces to an object at run time. This will allow the application to take advantage of special properties of the component without having to recompile the application.

**Reflection**

Both object migration and network adaptability are examples of object behavior depending upon an object’s implementation and its runtime system. The ability to obtain information about an object, such as its class or the interfaces it implements at run time, is called *reflection*. It also refers to an object’s capability to infer properties about its implementation and the state of the
environment in which it is executing. Reflection can be used to implement
dynamic invocation, for example. While reflection can be implemented in
any system, Java is the only conventional language that supports reflection
directly.

A closely related concept is that of a metaobject [311], which can be thought
of as a runtime object that is bound to each application-level object. In some
systems, metaobjects are used to implement method invocations, so that the
choice of network protocol for executing a particular method invocation can
be controlled by the associated metaobject. This strategy allows the object
making the method call to be written without concern for how the call is
implemented, since that is the metaobject's job. It also allows greater variety
in implementing some of the features listed in this section. For example, an
alternative to object migration is to endow a system with a pseudomigration
capability, which works as follows. The metaobject associated with an object
caches each of that object's requests for a member function call. If the meta-
object can detect that the current compute host is too busy, it can create an
instance of the controlled object on another host and forward the call to the
new instance.

**Event Logging**

Debugging distributed systems is difficult and requires a mechanism that
can log the timing and explanation of events associated with a given set of
distributed interactions.

**Fault Tolerance**

An exception-handling mechanism is the first step toward building reliable
systems, but it falls far short of providing a mechanism that reliably tolerates
failure. The system must be able to restart applications automatically and to
roll back transactions to a previous known state.

**Authentication and Security**

Authentication allows us to identify which applications and users are allowed
to access system components. Security ensures that interactions can be ac-
complished safely for the data as well as the implementations. It is an issue
that goes far beyond the domain of the object system, but the object system
must provide a way to allow the user access to available authentication and
security tools.
Beyond Client-Server

For high-performance computation, future distributed-object systems must support a greater variety of models than simple client-server schemes. As illustrated in Section 9.2, there are paradigms that include peer-to-peer object networks, and we can imagine future massive networks of components and software agents that work without centralized control and dynamically respond to changing loads and requirements.

Support for Parallelism

Beyond multithreaded applications are those involving the concurrent activity of many components. An object system must allow both asynchronous and synchronous method calls, as well as multicast communication and collective synchronization, both of which are essential for supporting parallel operation on large numbers of concurrently executing objects.

9.4 THE LEGION GRID ARCHITECTURE

In the preceding sections we have outlined many of the technical problems that are associated with extending contemporary component and distributed-object technology to support grid applications. In the remainder of this chapter, we address the problem of delivering this type of programming infrastructure to the application builder. More specifically, the component architecture that the programmer uses is a high-level programming model, which must provide easy-to-use abstraction for complex grid services.

The task of implementing application-level programming abstractions in terms of basic grid functionality is a major challenge. There are three ways to address this problem. One approach, explored in Chapter 10, is to extend existing commodity technology. A second approach is to layer an application-level component architecture on top of a grid architecture such as the Globus toolkit, described in Chapter 11. The current versions of HPC++ and CC++, for example, use Nexus, the Globus communication layer, to support object-oriented RMI, and Java RMI has been ported to run over Nexus [80]. An effort to extend this object layer to a Globus-compatible component model is under way. The third approach is to provide a single, coherent virtual machine that addresses key grid issues such as scalability, programming ease, fault tolerance, security, and site autonomy completely within a reflective, object-based metasystem. The University of Virginia’s Legion system is the best example of this type of grid architecture.
Legion is designed to support millions of hosts and trillions of objects existing in a loose confederation and tied together with high-speed links. The user can sit at a terminal and manipulate objects on several processors, but has the illusion of working on a single powerful computer. The objects the user manipulates can represent data resources, such as digital libraries and video streams; applications, such as teleconferencing and physical simulations; and physical devices, such as cameras, telescopes, and linear accelerators. Naturally, the objects being manipulated may be shared with other users. It is Legion's responsibility to support the abstractions presented to the user; to transparently schedule application components on processors; to manage data migration, caching, transfer, and coercion; to detect and manage faults; and to ensure that the user's data and physical resources are adequately protected.

9.4.1 Legion Design Objectives

The Legion design is based on 10 central objectives as follows.

1. **Site autonomy**: Legion will not be a monolithic system. It will be composed of resources owned and controlled by an array of organizations. These organizations, quite properly, will insist on having control over their own resources—for example, specifying how much of a resource can be used, who can use it, and when it can be used.

2. **Extensible core**: It is not possible to know or predict many current and future needs of all users. Legion's mechanism and policy must be realized via extensible and replaceable components that permit Legion to evolve over time and allow users to construct their own mechanisms and policies to meet their specific needs.

3. **Scalable architecture**: Because Legion will consist of millions of hosts, it must have a scalable architecture rather than centralized structure. This means that the system must be totally distributed.

4. **Easy-to-use, seamless computational environment**: Legion must mask the complexity of the hardware environment and of the communication and synchronization involved in parallel processing. Machine boundaries, for example, should be invisible to users, and compilers acting in concert with runtime facilities must manage the environment as much as possible.

5. **High performance via parallelism**: Legion must support easy-to-use parallel processing with large degrees of parallelism. This requirement includes task and data parallelism and their arbitrary combinations.
6. Single persistent name space: One of the most significant obstacles to wide area parallel processing is the lack of a single name space for file and data access. The existing multitude of disjoint name spaces makes writing applications that span sites extremely difficult.

7. Security for users and resource owners: Because Legion does not replace existing operating systems, we cannot significantly strengthen existing operating system protection and security mechanisms. In order to ensure that existing mechanisms are not weakened by Legion, it must provide mechanisms that allow users to manage their own security needs. Legion should not define the user's security policy or require a "trusted" Legion.

8. Management and exploitation of resource heterogeneity: Legion must support interoperability between heterogeneous hardware and software components, as well as take advantage of the fact that some architectures are better than others at executing particular applications (e.g., vectorizable codes).

9. Multiple language support and interoperability: Legion applications will be written in a variety of languages. It must be possible to integrate heterogeneous source-language application components in much the same manner that heterogeneous architectures are integrated. Interoperability requires that Legion support legacy codes.

10. Fault tolerance: In a system as large as Legion, it is certain that at any given instant several hosts, communication links, and disks will fail. Dealing with these failures and with the resulting dynamic reconfiguration is a necessity for both Legion and its applications.

In addition, the Legion design is shaped by the following three constraints. First, Legion cannot replace host operating systems. Organizations will not permit their machines to be used if their operating systems must be replaced. Operating system replacement would require them to rewrite many of their applications, retrain many of their users, and possibly make their machines incompatible with other systems in their organization.

Second, Legion cannot legislate changes to the interconnection network, but must assume that the network resources and the protocols in use are outside any one group's control and should be accepted as an ungovernable element in large-scale parallel processing.

And, finally, Legion cannot insist that it be run as "root" (or the equivalent). Indeed, quite the contrary: most Legion users will want it to run with the fewest possible privileges in order to protect themselves.
9.4.2 Legion System Architecture

Legion is a reflective object-based system that endows classes and metaclasses (classes whose instances are themselves classes) with system-level responsibility. Legion users will require a wide range of services on various levels, including security, performance, and functionality. No single policy or set of policies will satisfy every user; hence, whenever possible, users must be able to decide which trade-offs are necessary and desirable. Several characteristics of Legion's architecture reflect and support this philosophy.

*Everything Is an Object*

The Legion system consists of a variety of hardware and software resources, each of which is represented by a Legion object (defined as an active process that responds to member function invocations from other objects in the system). Legion describes the message format and high-level protocol for object interaction, but not the programming language or the communications protocol.

*Classes Manage Their Instances*

Every Legion object is defined and managed by its class object, which is itself an active Legion object. Class objects are given system-level responsibility: classes create new instances, schedule them for execution, activate and deactivate them, and provide information about their current location to client objects that wish to communicate with them. In this sense, classes act as managers and make policy, as well as define instances. Classes whose instances are themselves classes are called *metaclasses*.

*Users Can Provide Their Own Classes*

Legion allows users to define and build their own class objects, which permits programmers to determine and even change the system-level mechanisms that support their objects. Legion 1.0 (and future Legion systems) contains default implementations of several useful types of classes and metaclasses. Users are not forced to use these implementations, however, particularly if the implementations do not meet the users' performance, security, or functionality requirements.
Core Objects Implement Common Services

Legion defines the interface and basic functionality of a set of core object types, which support basic system services such as naming and binding, and object creation, activation, deactivation, and deletion. Core Legion objects provide the mechanisms that classes use to implement policies appropriate for their instances. Examples of core objects include hosts, vaults, contexts, binding agents, and implementations.

9.4.3 The Legion Object Model

Legion objects are independent and logically address-space-disjoint active objects that communicate with one another via nonblocking method calls, which may be accepted in any order by the called object. Each method has a signature that describes the parameters and return value, if any, of the method. The complete set of method signatures for an object fully describes its interface (which is determined by its class). Legion class interfaces can be described in an IDL, several of which will be supported by Legion.

Naming System

Legion implements a three-level naming system. At the highest level, users refer to objects using human-readable strings, called context names. Context objects map context names to LOIDs (Legion object identifiers), which are location-independent identifiers. Each identifier includes an RSA public key. Since LOIDs are location independent, they are insufficient for communication by themselves. A LOID is therefore mapped to an LOA (Legion object address) for communication. An LOA is a physical address (or set of addresses, in the case of a replicated object) that contains sufficient information to allow other objects to find and communicate with the object, for example, an (IP address, port number) pair.

Object States

A Legion object can be in one of two different states, active or inert. As designed, Legion will contain too many objects for all to be represented simultaneously as active processes and therefore requires a strategy for maintaining and managing representations of these objects in their inert state in persistent storage. An inert object is represented by an object-persistent representation (OPR), which is a set of associated bytes residing in stable storage somewhere in the Legion system. The OPR contains information about an object's state that enables the object to move to an active state. An active object runs as a
process that is ready to accept member function invocations; an active object's state is typically maintained in the address space of the process, although this is not strictly necessary.

**Core Objects**

Several core object types implement the basic system-level mechanisms required by all Legion objects. Like classes and metaclasses, core objects are replaceable system components; users (and in some cases resource controllers) can select or implement appropriate core objects.

*Binding agents* are Legion objects that map LOIDs to LOAs. A (LOID, LOA) pair is called a binding. Binding agents can cache bindings and organize themselves in hierarchies and software combining trees, in order to implement the binding mechanism in a scalable and efficient manner.

*Context objects* map context names to LOIDs, allowing users to name objects with arbitrary high-level string names, and enabling multiple disjoint name spaces to exist within Legion. All objects have a current context and a root context, which define parts of the name space in which context names are evaluated.

*Host objects* represent processors in Legion. One or more host objects run on each computing resource that is included in Legion. Host objects create and manage processes for active Legion objects. Classes invoke member functions on host objects in order to activate instances on the computing resources that the hosts represent. Representing computing resources with Legion objects abstracts the heterogeneity that results from different operating systems having different mechanisms for creating processes. Further, it provides resource owners with the ability to manage and control their resources as they see fit.

Just as the host object represents computing resources and maintains active Legion objects, the *vault object* represents persistent storage, but only for the purpose of maintaining the state, in OPRs, of the inert Legion objects supported by the vault.

*Implementation objects* allow Legion objects from other Legion systems to run as processes in the system. An implementation object typically contains machine code that is executed when a request to create or activate an object is made. More specifically, an implementation object is generally maintained as an executable file that a host object can execute when it receives a request to activate or create an object. An implementation object (or the name of an implementation object) is transferred from a class object to a host object to enable the host to create processes with the appropriate characteristics.

Legion specifies functionality and interfaces, not implementations. Legion 1.0 provides useful default implementations of class objects and of all the core
system objects, but users are never required to use the defaults. In particular, users can select (or build their own) class objects, which are empowered by the object model to select or implement system-level services. This feature of the system enables object services (e.g., creation, scheduling, security) to be made appropriate for the object types on which they operate, and eliminates Legion's dependence on a single implementation for its success.

9.5 A CLOSER LOOK AT LEGION

Space limitations do not permit a detailed discussion of how Legion realizes its objectives. Thus, rather than attempt to compress a large and complex system into a few pages, we will briefly expand on three aspects of Legion that are of interest to the high-performance computing community: security, high performance, and scheduling and resource management.

9.5.1 Security

Legion offers the opportunity of bringing the power and resources of millions of interlinked computers to the desktop computer. While that possibility is highly attractive, users will adopt Legion only if they feel confident that it will not compromise the privacy and integrity of their resources. Without security, Legion systems can offer some limited uses. But if the full Legion vision of a worldwide metacomputer is to become a reality, reliable and flexible security is essential.

Security Problems

Security has been a fundamental part of the Legion design from the beginning. Early work identified two main problems: users must be able to install Legion on their sites without significant risk, and they must be able to protect and control their Legion resources as they see fit.

The solution to the first problem is reflected in the broad design goals for Legion. Specifically, Legion does not require any special privileges from the host systems that run it. Administrators have the option of taking a very conservative approach while installing the system. Furthermore, Legion is defined as an architecture, not an implementation, allowing individual sites to reimplement functionality as necessary to reflect their particular security constraints.
The second problem, protecting Legion resources, requires multiple solutions. In an environment where users may range from students to banks to defense laboratories, it is impossible for Legion to dictate a single security policy that can hope to satisfy everyone. Therefore, Legion uses a flexible framework that adapts to many different needs. Individual users can choose how much they are willing to pay in time and convenience for the level of security they want. They can also customize their Legion system’s security policies to match their organization’s existing policies.

Placing policy in the hands of users is much more than just an attractive design feature. A decentralized system does not use security architectures based on control and mediation by "the system." Nor is there a single owner who sets and enforces global policies. In such an environment, users must ultimately take responsibility for security policies. Legion is designed to facilitate that goal.

The Security Model

The basic unit in Legion is the object, and the Legion security model is therefore oriented toward protecting both objects and object communication. Objects are accessed and manipulated via method calls; an object’s rights are centered in its capabilities to make those calls. A file object may support methods for read, write, seek, and so forth, so that the read right for a file object might permit read and seek, but not write. The user determines the security policy for an object by defining the object’s rights and the methods they allow. Once this step is done, Legion provides the basic mechanism for enforcing that policy.

Every object in Legion supports a special member function called “MayI” (objects with no security have a NULL MayI). MayI is Legion’s traffic cop: All method calls to an object must first pass through MayI before the target member function is invoked. If the caller has the appropriate rights for the target method, MayI allows that method invocation to proceed.

To make rights available to a potential caller, the owner of an object gives the caller a certificate listing the rights granted. This certificate cannot be forged. When the caller invokes a method on the object, it presents the appropriate certificate to MayI, which then checks the scope and authenticity of the certificate. Alternatively, the owner of an object can permanently assign a set of rights to a particular caller or group. In that case, MayI’s responsibility is to confirm the caller’s identity and membership in one of the allowed groups and then to compare the rights authorized with the rights required for the method call.
Besides regulating user access control, Legion also protects underlying communications between objects. Every Legion object has a public-key pair; the public key is part of the object's name (its LOID). Objects can use the public key of a target object to encrypt their communications to it. Likewise, an object's private key can be used to sign messages, thereby providing authentication and nonrepudiation. This integration of public keys and object names eliminates the need for a certification authority. If an intruder tries to tamper with the public key of a known object, the intruder will create a new and unknown name.

The combined components of the security model encourage the creation of a large-scale Legion system with multiple overlapping trust domains. Each domain can be separately defined and controlled by the users that it affects. When difficult problems arise, such as merging two trust domains, Legion provides a common and flexible context in which they can be resolved.

9.5.2 High Performance

Legion achieves high-performance computing in two ways: by selecting processing resources based on load and job affinity, and by parallel processing.

Even single-task jobs can have better performance when presented with a range of possible execution sites. The user can choose the host with the lowest load or the greatest power. Power, in this context, might be defined by performance on the SPEC benchmarks adjusted for load, or by using the application itself as a benchmark. Similarly, different components of a coarse-grained meta-application may be scheduled on different hosts (based on the component's affinity to that type of host), leading to a phenomenon known as superconcurrency [214]. In either scenario, Legion's flexible resource management scheme lets user-level scheduling agents choose the right resource.

Alternatively, Legion can be used for traditional parallel processing, as when executing a single application across geographically separate hosts, or supporting meta-applications (e.g., scheduling the components of a single meta-application on the nodes of an MPP). Legion supports a distributed-memory parallel computing model in four ways: supporting parallel libraries, supporting parallel languages, wrapping parallel components, and exporting the runtime library interface.

Supporting Parallel Libraries

The vast majority of parallel applications written today use MPI [250] or PVM [227]. Legion supports both MPI's and PVM's libraries via emulation
libraries, which use the underlying Legion runtime library. Existing applications need only to be recompiled and relinked in order to run on Legion.

**Supporting Parallel Languages**

Legion supports MPL (Mentat Programming Language, described in [569]), BFS (Basic Fortran Support), and Java. MPL is a parallel C++ language in which the user specifies those classes that are computationally complex enough to warrant parallel execution. Class instances are then used like C++ class instances: the compiler and runtime system take over, construct parallel computation graphs of the program, and then execute the methods in parallel on different processors. Legion is written in MPL. BFS is a set of pseudocomments for Fortran and a preprocessor that gives the Fortran programmer access to Legion objects. It also allows parallel execution via remote asynchronous procedure calls, as well as the construction of program graphs. The Java interface allows Java programs to access Legion objects and to execute member functions asynchronously.

**Wrapping Parallel Components**

Object wrapping is a time-honored tradition in the object-oriented world, but Legion extends the notion of encapsulating existing legacy codes into objects one step further by encapsulating a parallel component into an object. To other Legion objects, the encapsulated object appears sequential, but executes faster. Thus, one could encapsulate a PVM, HPF, or shared-memory threaded application in a Legion object.

**Exporting the Runtime Library Interface**

The Legion team cannot provide the full range of languages and tools that users need. The designers of Legion, rather than developing everything at the University of Virginia, intended the system to be an open community artifact to which other languages and tools are ported. To support third-party software development, the complete runtime library interface is available and may be directly manipulated by user libraries. The Legion library is completely reconfigurable: It supports basic communication, encryption/decryption, authentication, exception detection and propagation, and other features. One feature of particular interest is program graph support.

Program graphs (Figure 9.6) represent functions and are first class and recursive. Graph nodes are member function invocations on Legion objects or subgraphs. Arrows model data dependencies. Graphs are constructed by starting
with an empty graph and adding nodes and arcs. Graphs may be combined, resulting in a form of function composition. Finally, graphs may be annotated with arbitrary information, such as resource requirements and architecture affinities. The annotations may be used by schedulers, fault tolerance protocols, and other user-defined services.

9.5.3 Scheduling and Resource Management

The Legion scheduling philosophy is one of reservation through a negotiation process between resource providers and resource consumers. Autonomy is considered to be the single most crucial aspect of this process, for two reasons.

First, site autonomy is crucial in attracting resource providers. In particular, participating sites must be assured that their local policies will be respected by the system at large. Therefore, final authority over the use of a resource is placed with the resource itself.

Second, user autonomy is crucial to achieving maximum performance. A single scheduling policy will not be the best answer for all problems and programs: Users should be able to choose between scheduling policies, selecting the one that best fits the problem at hand or, if necessary, providing their
own schedulers. A special, and vitally important, example of user-provided schedulers is that of application-level scheduling. This allows users to provide per-application schedulers that are specially tailored to match the needs of the application. Application-level schedulers will be commonplace in high-performance computing domains.

Legion currently provides two types of resources: computational resources (hosts) and storage resources (vaults). Network resources will be incorporated in the future. As seen in Figure 9.7, the Legion scheduling module consists of three major components: a resource state information database, a module that computes request mapping to resources (hosts and vaults), and an activation agent responsible for implementing the computed schedule. These items are called the Collection, Scheduler, and Enactor, respectively.

The Collection interacts with resource objects to collect information describing the system’s state (Figure 9.7, step 1). The Scheduler queries the Collection to determine a set of available resources that match the Scheduler’s requirements (step 2). After computing a schedule, or set of desired schedules, the Scheduler passes a list of schedules to the Enactor for implementation (step 3). The Enactor then makes reservations with the individual resources (step 4) and reports the results to the Scheduler (step 5). Upon approval by the Scheduler, the Enactor places objects on the hosts and monitors their status (step 6).

If the user does not wish to select or provide an external scheduler, the Legion system (via the class mechanism) provides default scheduling behavior that supplies general-purpose support. Through the use of class defaults, sample schedulers, and application-level schedulers, the user can balance the effort put into scheduling against the resulting application performance gain.
9.6 APPLICATION SCENARIOS AND LEGION

To conclude this overview of Legion, let us revisit the application scenarios described in the first half of this chapter. A Legion implementation of the teleimmersion collaboration design application would closely follow the general object-oriented design presented earlier. The display object, the visual database object, the simulation components, the design database, and the user interface control objects would all be Legion objects that would communicate via method invocations.

The Digital Sky project is a more interesting example of a system that can exploit Legion. In a Legion implementation, the application "object databases" that contain the observations would be Legion objects—perhaps instances of an observation_db class. An observation_db object would have an interface tailored to the application. Thus, rather than generic (and therefore hard to optimize for the application) functions such as read() and write() or select-from-where(), an observation_db object would have functions such as get_sky_volume() or get_object(). In fact, the interface is completely arbitrary, allowing the designer the choice of query and update interfaces.

Access to observation_db instances would be location transparent because of the nature of Legion LOIDs. They could also be online versus archive transparent. That is, the Legion vault storing the persistent state of the objects could mask whether the state is on disk or stored on an archival medium such as tape.

The implementation of observation_db could then be optimized for the type of data stored and the most common data requests. For example, high performance for large sparse databases can be realized by using a PLOP file [502] or a quad-tree [487], as has been done for radio astronomy data [303, 302]. Data could also be previously fetched and cached based on access predictions provided by the user via special member functions, rather than using a demand-driven strategy and naive assumptions about temporal and spatial locality, as is typically the case.

The internal implementation of observation_db could be internally parallel as well. The data could be horizontally partitioned across multiple sub-objects, each of which resided on a separate device. Queries against the observation_db could then be executed in parallel, with multiple devices active at the same time, resulting in greater bandwidth.

Legion further supports the Digital Sky requirements by providing the following:
Further Reading

- A flexible access control policy that can easily be tailored to meet a variety of needs
- Support for flow-oriented processing via MPL, BFS, and the underlying graph support mechanism
- Support for user-level scheduling that allows either the computation to be moved to the data or the data to be moved to the computation
- The ability to dynamically insert object-monitoring code for performance debugging
- The ability to encapsulate legacy databases in Legion objects by wrapping them in an object and restricting the object's placement to the particular host or hosts where the legacy system can run

Finally, using Legion's unique metaclass system, one could create a metaclass for the observation_db class that supported object replicas. Replicas could be generated and placed as needed at multiple sites for both faster access and increased availability in the event of equipment failure. Replicas could also be transparently generated on demand close to a data consumer, acting as intelligent prefetching and caching agents.

ACKNOWLEDGMENTS

Fritz Knabe, Steve Chapin, and Mike Lewis assisted in preparing the Legion material. Portions of the Legion material have appeared elsewhere, specifically the 10 design objectives and the three constraints, which have appeared in many Legion papers. The Legion work has been supported by the following grants and contracts: DARPA (Navy) contract no. N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, DOE D459000-16-3C, DARPA (GA) SC H607305A, and Northrop-Grumman.

FURTHER READING

For more information on the topics covered in this chapter, see www.mkp.com/grids and also the following references:
Books by Orfali, Harkey, and Edwards [427] and Chappell [117] provide good introductions to distributed objects.

Lockhart's book [352] provides information on DCE, which plays a critical, historical role in the evolution of this technology.

Schmidt's papers discuss high-performance CORBA and the ACE Adaptive Communication Environment [490, 488].

Chandy's work in the Infospheres project [114] also provides many unique and inventive approaches to the problems described here.
A Flexible Security System for Metacomputing Environments*

Adam Ferrari, Frederick Knabe, Marty Humphrey,
Steve Chapin, and Andrew Grimshaw

University of Virginia
Department of Computer Science

Abstract. A metacomputing environment is a collection of geographically distributed resources (people, computers, devices, databases) connected by one or more high-speed networks, and potentially spanning multiple administrative domains. Security is an essential part of metasystem design—high-level resources and services defined by the metacomputer must be protected from one another and from corrupted underlying resources, and underlying resources must minimize their vulnerability to attacks from the metacomputer level. We present the Legion security architecture, a flexible, adaptable framework for solving the metacomputing security problem. We demonstrate that this framework is sufficiently flexible to implement a wide range of security mechanisms and high-level policies.

1 Introduction

Legion [5, 6] is a distributed computing platform for combining very large collections of independently administered machines into single, coherent environments. Like a traditional operating system, Legion provides convenient user abstractions, services, and policy enforcement mechanisms over a diverse set of lower-level resources. The difference is that in Legion, these resources may consist of thousands of heterogeneous processors, storage systems, databases, legacy codes, and user objects, all distributed over wide-area networks spanning multiple administrative domains. Legion provides the means to pull these scattered components together into a single, object-based metacomputer that accommodates high degrees of flexibility and site autonomy.

Security is an essential part of the Legion design. In a metacomputing environment, the security problem can be divided into two main concerns: (1) protecting the metacomputer's high-level resources, services, and users from each other and from corrupted underlying resources, and (2) preserving the security policies of the underlying resources that form the foundation of the metacomputer and minimizing their vulnerability to attacks from the metacomputer level. For example, restricting who is able to configure a metacomputer-wide scheduling service would fall in the first category. Its solution requires metacomputer-specific definitions of identity, authorization, and access control. Meanwhile, enforcing a policy that permits only those metacomputer...
users who have local accounts to run jobs on a given host falls in the second category. Its solution might require a map between local identities and verifiable metacomputer identities.

To satisfy users and administrators, a full security solution must address and reconcile both of these security concerns. Users must have confidence that the data and computations they create within the metacomputer are adequately protected. Administrators need assurances that by adding their resources to a metacomputer (and thus making those resources more accessible and valuable to users), they are not also introducing unreasonable security vulnerabilities into their systems.

Attempting to incorporate security as an add-on late in the implementation process has been problematic in a number of first-generation metacomputing systems such as PVM, MPI, and Mentor. To avoid this pitfall, the Legion group has addressed security issues since the earliest design phases [10]. Our metacomputing security model has three interrelated design goals: flexibility, autonomy, and breadth. Flexibility demands that the framework be adaptable to many different security policies and allow multiple policies to coexist. Autonomy is essential so that organizations and users within a metacomputing environment can select and enforce their desired security policies independently. Finally, breadth refers to the ability of the metacomputer's architectural framework to enable a rich set of security policy features.

These goals are strongly driven by our view that a fundamental capability of a metacomputer is its ability to scale over and across multiple trust domains. A Legion "system" is really a federation of meta- and lower-level resources from multiple domains, each with its own separately evaluated and enforced security policies. As such, there is no central kernel or trusted code base that can monitor and control all interactions between users and resources. Nor is there the concept of a superuser—no one person or entity controls all of the resources in a Legion system.

If it is to satisfy a broad range of security needs, our architecture must allow the implementation of a number of different security features. These include:

- Isolation
- Access control for resources
- Identity of principals
- Detection and recovery
- Communication privacy and integrity
- Integration with standard mechanisms

The first point, isolation, refers to the ability of components in the metacomputer to insulate themselves from security breaches in other parts of the system. This feature is particularly important in large Legion networks, where we must generally assume that at least some underlying hosts have been compromised or may even be malicious.

In this paper we elaborate a metacomputing architecture based on our design goals that addresses both parts of the metacomputing security problem. In our discussion, we present examples of mechanisms we have designed or implemented within the architecture that enable a number of useful security policies, and provide examples of those policies.

2 Architectural Support for Security

Legion is composed of independent, active objects. All entities of interest within the system—processing resources, storage, users, etc.—are represented by objects [7]. Le-
Legion objects communicate via asynchronous method calls supported by an underlying message passing system. Each method call contains actual parameters and an optional set of implicit parameters, metadata that is available to called objects. Objects are instances of classes that define their interface, which is required to be a superset of a minimal object-mandatory interface. Object-mandatory methods include functions such as an interface query and methods to implement object persistence.

Legion objects are persistent, and are defined to be in one of two states: active or inert. When an object is active, it is hosted within its own running process and can service method calls. When an object is inert, its state (called its Object Persistent Representation, or OPR) is preserved on a storage device managed within the system. Objects implement internal methods to store and recover their dynamic state.

**Legion Runtime Library** The implementation of Legion objects is supported by a Legion Runtime Library (LRTL) interface. The LRTL defines the interfaces to services such as message passing, object control (e.g., creation, location, deletion), and other basic required mechanisms.

A critical element of the LRTL is its flexible, configurable protocol stack [9]. All of the processing performed in the construction of method calls at the sender and in handling them at the recipient is configured using a flexible, event-based model. This feature makes it especially convenient for tool builders to provide drop-in protocol layers for Legion objects. For example, adding message privacy through a cryptographic protocol is simply a matter of registering the appropriate message processing event handlers into the Legion protocol stack—the added service is transparent to the application developer.

**Core Objects** Within the Legion object model we define the interfaces to a set of basic classes that are fundamental to the operation of the system and that support the implementation of the object model itself.

*Host Objects* in Legion represent processing resources. When a Legion object is activated, it is a Host Object that actually creates a process to contain the newly activated object. The Host Object thus controls access to its processing resource and can enforce local policies, e.g., ensuring that a user does not consume more processing time than allotted.

*Vault Objects* in Legion represent stable storage available within the system for containing OPRs. Just as Host Objects are the managers of active Legion objects, Vault Objects are the managers of inert Legion objects. For example, Vaults are the point of access control to storage resources, and can enforce policies such as file system allocations.

Hosts and Vaults provide the system with interfaces to processing and storage resources. The use of these interfaces is encapsulated by *Class Manager Objects.*

---

1. In some environments, the Host Object may enter the object as a new job to run in a queue management system, but this difference is transparent to the rest of Legion.

2. In many of the cited Legion references, Class Manager Objects are referred to simply as "Class Objects."
Managers are responsible for managing the placement, activation, and deactivation of a set of objects, or instances, of a given class. They provide a central mechanism for specifying policy for a set of like objects. Policies set by the Class Manager include defining which implementations are valid for instances, which hosts are suitable for execution of instances, which users may create new instances, and so on. In addition to setting policy for instances, Class Managers serve as location authorities for instances, supporting the binding of object ids to low-level object addresses (typically an IP address plus port number).

A critical aspect of the Legion core object classes is that they define interfaces, not implementations. The Legion software distribution provides a number of default reference implementations of each core object type, but the model explicitly enables and encourages the configuration, extension, and even replacement of local core object implementations to suit site- and user-specific requirements. For example, by replacing the implementation of the Host Object, a site can define arbitrary mechanisms and policies for the usage of their computational resources.

3 Security Features in Legion

The Legion architecture is the critical foundation for satisfying the flexibility, autonomy, and breadth goals for our metacomputing security model. We now consider how those goals are met in the current system implementation.

Identity In Legion, every object is identified by a unique, location-independent Legion Object Identifier, or LOID. LOIDs consist of a variable number of binary fields. As a default Legion security practice, we use one of the LOID fields to store an X.509 certificate including (at a minimum) an RSA public key. By including an object’s public key in its LOID, we make it easy for other objects to encrypt communications to that object or to verify messages signed by it. Objects can just extract the key from the LOID, rather than looking it up in some separate database, which eliminates some kinds of public key tampering.

Users in Legion also have LOIDs. A user creates his own LOID, which is then registered with the system and entered in appropriate system groups and access control lists by resource providers. When an object makes a call on behalf of the user, the user’s LOID and associated credentials provide the basis for authentication and authorization. The ownership of a user’s LOID resides in the user’s unique knowledge of the private key that is paired with it. The private key is kept encrypted on disk, on a smart card, or in some other safe place.

For a resource, the essential step in deciding whether to grant an access request is to determine the identity of the caller. If a user communicates directly with the target object, he can establish his identity relatively easily with an authentication protocol. In a distributed object system, however, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf. To transfer the user’s identity in Legion, we issue credentials to objects. A credential is a list of rights granted by the credential’s maker, normally the user or his proxy. A credential is passed through call chains, and is presented to a resource to gain access. The resource checks the rights
in the credential and who the maker is, and uses that information in deciding to grant access.

There are two main types of credentials in Legion: delegated credentials and bearer credentials. A delegated credential specifies exactly who is granted the listed rights, whereas simple possession of a bearer credential grants the rights listed within it. A credential specifies the period for which it is valid, who is allowed to use the credential, and which method calls it can be used for. The credential also includes the identity and digital signature of its maker.

Tools or commands directly executed by the user create the credentials they need to carry out their actions. The credentials are made as specific as possible to avoid unnecessary dispersion of authority. Short timeouts in credentials, coupled with user-specific Refresh Objects that can revalidate expired credentials, permit a variety of recovery tactics if a credential or user key is stolen. Additional details concerning credentials and credential refresh can be found in [2].

Access Control In Legion, access is defined as the ability to call a method on an object. The object may represent a file, a Legion service, a device, or any other resource. Access control is not centralized in any one part of the Legion system. Each object is responsible for enforcing its own access control policy. It may collaborate with other objects in making an access decision, and indeed, this allows an administrator to control policy for multiple objects from a single point. The Legion architecture does not require this, however.

The general model for access control is that each method call received at an object passes through a MayI layer before being serviced. MayI is defined on a per-object basis, and is specified as an event in the configurable LRTL protocol stack [9]. MayI decides whether to grant access according to whatever policy it implements. If access is denied, the object will respond with an appropriate security exception.

MayI can be implemented in multiple ways. The default LRTL MayI implementation is based on access control lists and credential checking. In this MayI, allow and deny access control lists containing user and group LOIDs can be specified for each method in an object. When a method call is received, the credentials it carries are checked by MayI and compared against the access control lists. Multiple credentials can be carried in a call; checking continues until one provides access.

The form of access control provided by the default MayI is sufficient for some kinds of objects, such as file objects, but not for others. The LRTL configurable, event-based protocol stack makes it easy to replace or supplement the default MayI with extra functionality. Furthermore, the default MayI itself is relatively simple to modify if, for example, new forms of credentials or different kinds of access control lists must be supported. With the Legion security architecture, these types of changes can be made on a local basis without affecting other parts of a Legion system.

Communication Privacy and Integrity Encryption and integrity services are provided at the level of Legion messages. When a Legion message is prepared for sending, an event handler that implements a message security layer is triggered. This layer inspects the implicit parameters accompanying a message to determine which security functions
to apply. In the current LRTL, a message may be sent with no security, in private mode, or in protected mode. In both private and protected modes, certain key elements of a message (e.g., any contained credentials) are encrypted using the public key of the recipient. The functional difference between the two modes is in how the rest of the message is treated. In private mode it is encrypted, whereas in protected mode only a digest is generated to provide an integrity guarantee. Unless private mode is already on, protected mode is selected automatically if a message contains credentials. This is a failsafe measure to prevent credentials from being transmitted in the clear. Details of the encryption mechanisms can be found in [2].

Because the mode in use is stored in implicit parameters, it propagates through call chains. For example, a user can select private mode when calling an object. All subsequent calls made by objects on behalf of the user will also use private mode. The default security layer does not provide mutual authentication. The sender can be assured of the identity of the recipient, because only the desired recipient can read the encrypted parts of the message. The recipient usually doesn’t care who the actual sender is; its decisions are based solely on the credentials that arrived in the message.

Object Management and Isolation The management of active and inert objects by Legion core objects is an important point of local security mechanism and policy in Legion. Fundamentally, Legion software runs on existing operating systems with their own security policies. It is therefore critical that the implementation of the Legion object model ensure that extra-Legion mechanisms cannot be used to subvert higher-level security mechanisms. Similarly, it is important to ensure that Legion does not break local security policies at a site. A local system administrator is generally concerned with who can create processes on his system via Legion, what those processes can do, and who pays for their resource use. On Legion's side, there is a need to prevent user objects from interfering with one another or with core system objects (e.g., Hosts and Vaults), and to maintain the privacy of persistent state (OPRs). The latter is particularly significant because objects store their private keys in their OPRs.

The needs of Legion are common to any multi-user operating system, and our approach to providing them is to leverage off of existing operating system services. Our general strategy for isolating objects from one another in the default Legion implementation is to use separate accounts to execute different user objects. Similarly, we use local accounts and storage system protections to protect OPRs.

Accounts that can be used for these purposes fall into two categories. For those Legion users who happen to have accounts on the local system, processes and storage that represent the user’s objects can be owned by the user’s local account. For other users, we support the use of a pool of generic accounts that are designated for Legion use. The generic accounts usually have minimal permissions (e.g., no home directory, no group memberships, etc.). The local Host and Vault Objects use their own dedicated local accounts to ensure isolation from other user objects.

We encapsulate the privileged operations necessary for this policy in a Process Control Daemon (PCD) that executes on the host, providing services to the Host and Vault in a controlled fashion. The PCD is a small, easily vetted program that runs with root permissions. It is configured only to allow access by the user account on which the Host
and Vault Objects are running. Its key functions are recursive change ownership of a directory, process creation under a designated account, and process termination. The PCD limits the user-ids to which these operations can be applied to a set configured by the local system administrator. The set includes the generic Legion accounts and potentially the accounts of local Legion users.

Alternatively, a local site policy may require that Kerberos be used to authenticate access to all local user accounts. Depending on the local Kerberos configuration, the Host Object can use forwarded Kerberos credentials, entries in users' Kerberos authorization files, or callbacks to user credential proxies to start objects on the appropriate accounts. The point is not how this is done, but that it can be done: Legion can adapt to a large range of security standards as necessary.

4 Policy Examples

Although Legion's flexibility allows the implementation of a wide variety of security mechanisms, application developers and site administrators typically have higher-level policy specifications in mind when using software. The particular underlying mechanisms are less important, as long as the user can be assured that high-level policy requirements are being met. In this section, we consider illustrative examples of how the Legion system architecture and existing Legion tools can be organized to meet sample site and application policies.

Site Isolation A Legion system can consist of multiple domains, each possibly in a different organization or trust domain. System administrators contributing resources to a larger metasystem typically require certain site-isolation properties. For example, consider a site that makes resources available to Legion, and is managed by a given local Legion administrator, who we will call Admin. A reasonable policy is that no matter how subverted any external sites in the Legion system might be, no intruder can invoke methods on local Legion resources as Admin. Such a policy is clearly desirable since Admin is likely to have administrative control over critical local resources: who can use which machine, and for how long; who can access which locally stored OPRs; etc. The ability to invoke methods as Admin is tantamount to complete control of the local Legion software.

The desired isolation policy can be achieved through a number of straightforward safeguards enabled by the Legion framework. First and foremost, all of the core objects managing the local site should be started and configured by Admin. This isolated domain startup avoids any external trust dependencies on outside systems. However, to achieve the desired functionality of a metacomputer, the local domain will be connected to some set of external Legion domains. After this link to the external system is made, Admin must ensure that no messages containing his credentials are sent to off-site objects, as a subverted or malicious external site could then use Admin's credentials to break the isolation policy. However, simply stating that Admin should not pass credentials off-site is not good enough—Admin might make a simple mistake that could break the policy, so we would like automated enforcement of this safety measure. Such automated enforcement is easy in Legion: Admin simply uses a version
of the LRTL in which the protocol stack is configured with an extra event handler for the message-send event. If a message is inadvertently directed off-site while containing Admin credentials, the message is blocked and the event handler raises an exception. With this simple modification to Admin’s Legion environment, he can be assured that his credentials will not be dispersed to untrustworthy off-site objects.

Ensuring that Admin does not communicate with off-site objects has a desirable secondary effect. Since Admin cannot communicate with external, untrustworthy sites, he cannot place critical objects on resources at these sites. This benefit extends to an array of potentially critical, but not necessarily obvious, resources. For example, suppose Admin maintains a local Group Object listing the set of users that are allowed to start objects on local resources. If this object were allowed to execute on an untrustworthy site, its contents could be modified by a malicious resource owner, and local site-resource usage policy could be broken.

The two mechanisms described above, in combination with carefully configured access control for local core objects such as Hosts, Vaults, and critical Class Managers, ensure that the desired isolation policy will be met. Off-site objects will neither be able to generate nor steal local Admin’s credentials. External callers will be prevented from invoking unauthorized methods on local critical resources, ensuring that local access control is not tampered with, local resource usage policies are not modified, and that security failures in other domains do not have serious consequences for the local site.

Site-Wide Required Access Control The Legion access control model as presented in Section 2 is based on the assumption that users will configure access control for their own objects. This concept adds a powerful level of flexibility to the system—for example, it makes arbitrary resource access policies possible. However, on first examination it appears to relinquish the ability for a system administrator to set site-wide policies about access control for user objects. For example, the default Legion access control configuration does not grant the administrator for a Legion domain access to other users’ objects within the domain—there is no root user who can read any file or use any program in the domain. Such lack of ability to configure global, site-wide, mandatory access control policies may be unacceptable at some sites. However, the flexibility of the Legion architecture allows us to address this issue in a straightforward fashion using existing tools.

As an example of a site-wide access control policy, we consider the problem of prohibiting access to files by outside users. The Legion system defines a basic File Object that can be used to represent a file in the system. Access control for the normal Legion File Object is based on the default Legion MayI mechanism, which places no restrictions on what LOIDs (i.e., what users) may be placed on access control lists. To enforce the policy that files may not be accessed by outside users, we effectively want a way to control which LOIDs may be placed on the ACL for local file objects. We can achieve this policy using the power of local Host Objects to control access to local resources. The Host Objects at the site (which are owned and controlled by the local administrator) are a point of resource access policy—they define which types of objects may run at the site. Using this feature, the site administrator can strictly limit the classes of objects that may run at the site. In particular, the allowable set of classes can
be limited to those that are approved by the system administrator. The list of allowable
classes can be configured to only include file objects with an alternate May1 layer—an
extended version of the default ACL mechanism that also verifies that allowed LOIDs
are in a well-known group containing only the local site users. Given this simple con-
figuration, the site administrator can ensure that files are not inadvertently exported to
outside users through Legion. Furthermore, this approach generalizes to other site-wide
access control restrictions, and other similar site-wide policy enforcement problems.

Firewalls Firewalls are a simple fact of life at many security-conscious institutions.
While firewalls are not addressed explicitly in the Legion model, the Legion architecture
is sufficiently flexible to accommodate firewalls with ease. As is typical in firewall
situations, a proxy on the firewall host is the natural solution. However, the ability to
use custom versions of the Legion core objects, and the flexible protocol stack model
of the LRTL, allow proxy-based solutions to be employed in Legion in an especially
straightforward, user-transparent way.

Objects started on hosts behind a firewall automatically have a Proxy Object on the
firewall host assigned to them by their Host Object (in some cases, each user might
desire their own proxy object; in other cases, a shared proxy object is acceptable; either
model is simple to support). The object address for a newly activated object behind
the firewall that is reported to the object’s Class Manager is actually the address for
the Proxy Object—when callers of the object bind its LOID to an object address, they
will be given the address of the Proxy Object. The Proxy Object then acts as a simple
reflector, forwarding any received messages to their intended destinations behind the
firewall. Use of the Proxy Object to forward outbound messages from callers behind
the firewall is automated by a transparent add-in event handler in the LRTL protocol
stack.

Resource Selection Policy In principle, a user of a metacomputer shouldn’t need to
care which resources are used to execute his jobs. In practice, however, the trustworthi-
ness of the resources that are selected for certain applications is of critical interest to the
user. Policies regarding which resources may be used to execute objects are logically
localized within the Class Managers of a user’s object classes. In principle, any site
selection policy can be encoded in a user’s Class Manager Objects, giving the user total
control over the selection and use of trustworthy sites.

Although this problem is solved cleanly at the architectural level in Legion, we
decided this issue of site selection for application users important enough to warrant
special features in the default Class Manager Object reference implementations. All de-
fault Class Managers in Legion check for certain implicit parameters that can be used
to limit resource selection. By setting these implicit parameters in his Legion environ-
ment (using a provided tool), the user can configure a resource selection policy that will
propagate to all “create instance” methods called on Class Manager objects on behalf
of the user. Of course, the architectural principle that users can encode any resource
selection policy they wish in their own Class Manager implementations still holds; in
fact, a convenient model for such customization is supported by the default Class Man-
ager’s ability to be configured to use an external Scheduler Object with a well-known
interface. However, in the common case, where a user can generate a list of sites that he deems trustworthy and indicate this in his environment, the default implementation provides the mechanism to implement an effective resource selection policy.

5 Related Work

Two projects that incorporate security into large-scale distributed computing platforms are Globus and WebOS. Globus [3] is a “bag of services” model for metacomputing, in contrast to Legion’s integrated environment approach. The Globus Security Infrastructure is a single sign-on authentication system that is deployed at each site in a Globus network. Different underlying authentication protocols such as Kerberos and SSL may be plugged into the infrastructure via GSS-API modules. A local Globus site uses the authentication information it receives in a request to make authorization decisions; it can also call back to a user’s proxy to confirm the request.

The Globus Security Infrastructure essentially focuses on one component of the overall metacomputing security problem. Legion, with its “network operating system” perspective, addresses broader issues that allow the development of sophisticated security policies to manage metacomputer resources, as described in Section 4. The Legion architecture fundamentally permits greater autonomy and flexibility in the choice of security technologies and approaches. Globus does address an important part of the metacomputing security puzzle, however, and it could be chosen as an alternate mechanism to the current RSA approach for implementing identity and integrity in a Legion system.

CRISIS [1] is the security architecture for WebOS. WebOS is fundamentally different from Legion in terms of the basic services provided. WebOS provides a single, traditional file system and a fixed interface for authenticated remote process creation. CRISIS defines careful, effective security policies for these basic services. However, the CRISIS solution does not provide a means for easily developing security policies for new mechanisms as they are added to WebOS, nor does it provide a means for modifying the security policies supported for the existing services.

Two other projects related to security efforts in Legion, although not with the focus on metasystems, are Java and CORBA. The computational model of Java [4] (JDK 1.2) requires identity and authentication in order to execute digitally signed code downloaded from a remote site. The JDK provides per-class (or per-application) protection domains. However, it differs significantly from Legion in its lack of support for per-site security mechanisms, delegation, and user authentication.

The security model of CORBA [8] encompasses identification and authentication, authorization and access control, auditing, security of communication, non-repudiation, and security information administration. Typically, an ORB vendor implements CORBA security using existing technology such as GSS-API, Kerberos, and SESAME. Many of the goals of the CORBA security model are similar to the goals of the Legion security model, including simplicity, scalability, usability, and flexibility. However, CORBA is not a metacomputing system—it does not construct an operating system-like environment using underlying distributed resources. Given this fundamental difference in target use, CORBA does not address the metacomputing security problem.
6 Conclusions

We have presented the basic security architecture of the Legion system, and we have demonstrated that our design is sufficiently flexible to accommodate a wide variety of security-related mechanisms. This flexibility is critical to the successful deployment and use of metacomputing software. One-size-fits-all software dictated by a single group will never satisfy the requirements of the wide range of users and resource providers in a large-scale, cross-domain environment. We have also demonstrated that flexibility does not come at the price of complete lack of control. Within the flexible Legion framework, we showed how a number of important site-wide and application-wide security policies could be achieved. Naturally, the set of policies presented is only a small fraction of the policies that will be needed across the complete Legion environment.

The Legion system, including the security features described here, is currently publicly available. It is widely deployed on hundreds of machines at dozens of sites spanning multiple trust domains. Key portions of the software, such as the PCD described in Section 2, have been vetted and approved by system administrators at sites such as the San Diego Supercomputing Center and the US Naval Oceanographic Office (NAVO). In the future, we plan to continue deployment of Legion, developing additional mechanism and adapting to new site-local policies as required. We are also in the process of measuring the performance impact of key Legion security mechanisms.

References

Enabling Flexibility in the Legion Run-Time Library *

Charles L. Viles, Michael J. Lewis, Adam J. Ferrari
Anh Nguyen-Tuong, Andrew S. Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA, U.S.A.
http://legion.virginia.edu

Abstract This paper describes the design and implementation of the Legion run-time library (LRTL), focusing specifically on facilities that enable extensibility and configurability. These facilities include management of heterogeneous communication, an event-based mechanism for inter-component communication, and automated memory management. The paper provides several examples that illustrate the inherent flexibility of the LRTL implementation.

Keywords: configurable protocol stack, extensible run-time library, events, implicit parameters

1 Introduction

The widespread deployment of gigabit networks will effectively shrink the distance between computing resources and will enable wide area distributed-object computing systems that will consist of many heterogeneous, distributed, unreliable resources. Legion [1, 2] will be one such system. Without significant software support, users will not be able to manage the complexity of this environment. Meta-systems software [3]—software that resides "above" physical resources and operating systems and "below" users and applications programs—is needed. Legion metasystems software will include a run-time system, Legion-aware compilers that target this run-time system, and programming languages that present applications programmers with a high level abstraction of the system. Thus, Legion will allow users to write programs in several different languages, and will transparently create, schedule, and utilize distributed objects to execute the programs.

Legion's users will require a wide range of services in many different dimensions, including security, performance, monetary cost, and functionality. No single policy or static set of policies will satisfy every user, so users must be allowed to implement their own solutions and to determine their own trade-offs as much as possible.

Legion supports this philosophy in several different ways. For example, Legion provides the mechanisms for system-level services such as naming, binding, and migration, but does not mandate these services' policies or implementations. Legion requires a certain object-mandatory functional interface to all Legion objects; the implementation of the interface is left up to the object. Legion also specifies the minimum functional interface to a set of core system object types, the implementation of which can and will vary. Furthermore, Legion delegates much of what is usually considered system-level responsibility to classes, which are special Legion objects. For instance, classes are responsible for creating and locating their instances, and for selecting appropriate security and object placement policies.

The Legion run-time library (LRTL)—the subject of this paper—is the cornerstone of our
Legion meta-systems software. Legion object implementations are linked with LRTL, which provides the basic mechanisms to allow Legion objects to communicate with one another using Legion-compliant mechanisms. LRTL is intended to be used both by Legion-targeting compilers and by user-level code; thus, when we refer to LRTL’s “users,” we mean both compiler writers and applications programmers. In building LRTL, we were driven by several subgoals and constraints. First of all, we wanted to abstract much of the complexity that is inherent to heterogeneous distributed computing. For example, we wanted to alleviate the need for LRTL’s users to deal directly with the varying data formats on different machine architectures. More importantly, in accordance with the overall Legion philosophy that one size does not fit all, we wanted LRTL to become a useful software tool with which users could build different policies and algorithms along many different dimensions, without having to build an entirely different library. Thus, we built LRTL itself to be extensible and configurable.

This paper describes how LRTL supports this flexibility, and provides several examples of how the flexibility can be exploited to implement different programming styles and useful distributed systems components. Section 2 begins with a brief system description, Section 3 explains the major mechanisms through which LRTL enables flexibility, and Section 4 illustrates several examples of this flexibility. We conclude with related work and a brief summary. In the interest of brevity, we have omitted many details, especially in our description of LRTL itself. For a more complete exposition of LRTL, please see [4].

2 System description

Before discussing the the flexibility and extensibility features of LRTL, we must put it in context with the overall Legion system. In this section, we examine the general features of Legion, including its basic object model, the core objects that implement primitive system mechanisms, and the typical usage of the system.

Legion is an object-based system—all entities of interest to the system are objects. Legion objects are logically independent address space disjoint collections of data and associated methods. Objects can contain one or more associated threads of control, and communicate via asynchronous method invocations. Objects are identified by unique names called Legion object identifiers (LOIDs). LOIDs are location independent, and must be mapped to low-level, ephemeral object addresses when inter-object communication takes place. Legion objects are persistent and may be in one of two states: active or inert. Active objects contain one or more threads of control and are ready to service method calls. Inert objects exist as passive object state representations on persistent storage. Legion transitions objects between active and inert states to use resources efficiently, to support object mobility, and to enable some approaches to failure resilience.

All Legion objects are described by an interface specified using an Interface Description Language (IDL). A Legion object’s interface describes the methods that it supports and the types of their parameters and return values. In addition to a user defined interface, all Legion objects support a set of object-mandatory methods that help implement basic Legion mechanisms such as object persistence, migration, and security. Methods on Legion objects are executed using a macro dataflow model [5]. This model requires that any method invocation sent to an object include (in addition to its parameters) a description of where the results produced by the method should be forwarded. For example, instead of being returned to the caller as in an RPC model, the result of a method invocation might be forwarded directly to some other object as a parameter to one of its methods.

Legion objects are instances of classes, which are themselves Legion objects. In addition to supporting object-mandatory methods, class objects support a set of class-mandatory methods that implement instance creation, location, migration, and deletion. Thus, classes provide
basic object management services, but they also act as policy makers for their instances. For example, classes have final authority in placing their instances, and can thus implement various performance and security policies.

Legion defines the interfaces to several special object types that implement traditionally system-level functionality. The activation and deactivation of object instances on physical hardware is managed by host objects. Each CPU resource in a Legion system is managed by at least one host object. The persistent storage on which inert Legion objects reside is managed by vaults; vaults manage inert Legion objects in much the same way that host objects manage active Legion objects. Finally, binding agents resolve LOIDs to object addresses.

Legion provides a variety of programming interfaces on several different levels. Some programmers will use Legion by writing programs in high level parallel languages such as Mentat [6]. The programs will then be transformed automatically by Legion-targeting compilers into Legion object implementations. Other programmers will use Legion by specifying an object interface in an IDL, using an IDL compiler to generate client and server stubs, and then providing the method implementations in a high level sequential language; this process corresponds to the most common method of building CORBA [7] objects. Still other programmers will write PVM [8] code and link it with a Legion PVM library, which will rely on Legion objects for its implementation. Finally, another set of users will require low level programming details, possibly for the implementation of system level objects such as hosts or vaults; these users will be able to program directly to the LRTL API.

3 Mechanisms for flexibility

This section describes several components of LRTL that help make it configurable and extensible. The components enable this flexibility in one of two different ways, (1) by abstracting some type of complexity that is inherent to building software for distributed-object computing, or (2) by providing mechanism designed specifically to allow LRTL to be extended or configured easily. Legion buffers and automatic reference counting (Sections 3.1 and 3.3) fall in the former category; events, program graphs, and implicit parameters (Sections 3.2, 3.4, and 3.5) fall in the latter.

3.1 Legion buffers

An lbuffer is the fundamental data container in LRTL. An lbuffer exports operations to read and write data from and to physical storage. Instances of different kinds of lbuffers export the same interface and perform the same basic function, but can have different characteristics from one another. For example, one kind of lbuffer may copy the data it contains into heap-allocated memory, another may simply maintain pointers to the data, and a third may read and write its data from and to a file. Further, lbuffers may also choose to compress or encrypt data, or both. To define its characteristics, each lbuffer contains a storage, a packer, an encryptor, and a compressor.\footnote{For brevity, we omit descriptions of encryptors and compressors in this paper.} Eight bytes of meta-data also accompany lbuffers; the metadata indicates the format in which the data is stored, and the algorithms, if any, that were used to encrypt and compress the data.

The storage associated with an lbuffer determines where and how the data is contained. One type of storage provided in LRTL is scattered storage. Scattered storage maintains data as a linked list of pointers to data chunks. Scattered storage can be configured to copy the data into chunks that it allocates, or to maintain pointers to data "owned" by other parts of LRTL. Another type of storage is persistent storage, which stores data in a file. Obviously, scattered storage and persistent storage will have very different performance characteristics.

A packer determines the data format conversion operations, if any, that are performed on the contained data when it is written to
and read from the lbuffer. A packer is the primary mechanism in LRTL for dealing with the fact that machines with different architectures, which store data in different formats (i.e. big vs. little endian, 32-bit vs. 64-bit words, etc.), need to communicate with one another. LRTL currently supports three different equivalence classes of architectures, “Alpha,” “Sparc,” and “x86.” For efficiency reasons, Legion assumes a “receiver makes right” [9] data conversion policy; the sender of a message (i.e. the original creator of an lbuffer) packs the message in its own native format, and the receiver is responsible for converting the data to the format appropriate for the architecture on which it resides.

Legion lbuffers enable the concept of packable C++ object classes in LRTL. A class is packable if it exports “pack()” and “unpack()” functions. Both pack() and unpack() take a single reference parameter that names an lbuffer. The pack() function writes the object’s state into the lbuffer parameter in such a way that the unpack() function of an object of the same class can read it out. LRTL’s C++ object classes are made packable for two reasons, (1) so that they can be passed between heterogeneous architectures within an lbuffer, and (2) so that they can be written to a persistent lbuffer to save the state of an object before it is deactivated or migrated.

3.2 Events

One of the fundamental issues in the implementation of a significant piece of software is the definition of how logical components of the software communicate with one another. This is especially important when the software will be extended by its users. The many approaches that exist range from the careful definition of interfaces and interactions, to more ad-hoc approaches that require intimate knowledge of many aspects of the code. LRTL employs a well-understood technology, events, and prescribes how to use it to facilitate flexible and extensible inter-component communication.

The prescription is straightforward. When component X wishes to communicate with component Y, X announces an event, which contains user-defined data and a tag that denotes an event kind. Each event kind has one or more associated event handlers (C++ functions) which may be called whenever an event of that kind is announced. Handlers for a particular event kind are given a priority that determines the order in which they are called. Any handler of a particular priority can prevent the execution of handlers with lower priority.

Component X communicates with component Y using events in four steps: (1) Y registers a handler for the particular event kind that X will announce. This ensures that when X announces the event, Y will receive it. (2) X creates an event using one of the provided event kinds as a template. If data needs to be sent along, then the data is attached to the event as well. (3) X announces the event to an event manager, an entity that enqueues events and ensures that its handlers are executed in priority order and only if preceding handlers have not prevented further execution. (4) The event manager dequeues and processes the event by calling its handlers.

Outside of application specific data manipulation, each of these actions requires only one or two lines of C++ code. The entire process is depicted in Figure 1.

The event mechanism provides flexibility in a number of ways. Handlers can be added, modified, and removed, event kinds can be added, and handler priorities can be set and reset to effect the order in which they are processed. One of the primary uses of events in LRTL is a configurable protocol stack, which we describe in Section 4.1.

3.3 Reference counting

LRTL implements a mechanism for automatic reference counting and safe dynamic memory management. The mechanism is intended for heap allocated C++ objects. It keeps track of references to each object that is “shared” by different parts of LRTL, and automatically
Figure 1: Communication between two components using events: (1) handler registration, (2) event creation, (3) event announcement by the event originator, and (4) handler execution by the event manager.

deletes the object when all meaningful references to it have disappeared. The mechanism is based on references, which take the place of C++ pointers, and reference counting objects. Each reference counting object is derived from an LRTL-provided base class that enables it to maintain a non-negative integer that indicates the number of references that "point to" that object. When a new reference is made to an object, the reference count within that object is incremented automatically. When a reference gets overwritten with another value, or when a local variable reference falls out of scope, the reference count in the object to which the reference points is automatically decremented. When the reference count falls to zero, the object is deleted automatically. All of this happens without any intervention by the programmer or user of references. With a couple of minor exceptions, using the reference counting mechanism is exactly like using pointers to dynamically allocated objects. The main thing to remember is not to delete the object that the reference points to, since that is done automatically.

The decision to include an automatic reference counting mechanism in LRTL was motivated by two observations: (1) memory copies are expensive and often hinder the performance of message passing code, and (2) keeping track of shared pointers and deciding which parts of the code are responsible for deleting which chunks of heap-allocated memory is extremely error prone and difficult to document effectively. The automatic mechanism combines the better performance that comes from avoiding memory copies with the safety and the correctness that comes from not having to worry about managing dynamically allocated memory. Obviously, the automatic reference counting mechanism introduces some overhead over simple pointer copies, but it is cheaper than garbage collection, and we believe the benefits outweigh the costs.

3.4 Program graphs
At a high level, computation is expressed in the Legion system using program graphs. A program graph is a data-flow graph whose nodes represent method invocations on Legion objects, and whose arcs represent data dependencies between the method invocations. This computation model is exactly the one described in [5], so we omit a detailed descrip-
tion. Figure 2 shows a simple user program and the resultant data dependencies expressed as a program graph.

The implementation of program graphs in Legion enables two important properties of the Legion system, (1) support for concurrency and parallel processing, and (2) support for graphs as first class objects. The latter aspect is important because some applications may require the specification of a computation in one object and the initiation of that computation in a different object. One conceptually simple way to accomplish this is to support program graphs as first class objects.

```c
main() {
    int a = 10, b = 15, x, y, z;
    MyObject A, B;
    x = A.op1(a);
    y = B.op1(b);
    z = A.op2(x, y);
    printf("%d\n", z);
}
```

Figure 2: Example user code and the resulting program graph.

3.5 Implicit parameters

In a wide area distributed-object system, it is often desirable and necessary to append meta information to high level entities such as method invocation requests and return results. LRTL uses implicit parameter lists to provide this capability. For example, programmers use implicit parameters, which accompany all message transmissions, to specify that they want all messages to be encrypted before transmission, or to send context information that indicates how error conditions should be handled and who should be notified when errors occur.

An implicit parameter is a \( (\text{tag, type, value}) \) triple. The tag is a string that names the item, the type is an integer whose value corresponds to one of several well-known parameter types, and the value is a packable entity that contains whatever data is associated with the parameter. Together, the tag and type determine how the value field of the implicit parameter should be interpreted.

LRTL provides two default implicit parameter lists, one for messages and one for method requests. This separation is supported because some activities take place at a message level granularity, while others occur at a method level granularity. Implicit parameters can be used in a variety of ways for a variety of purposes. In Sections 4.2 and 4.3 we explore some of these uses.

4 Examples of use

In this section, we provide several examples of how the mechanisms described in Section 3 have been or can be used to expand or configure LRTL.

4.1 The Legion protocol stack

Legion maintains a configurable protocol stack to handle location independent method requests by transforming them into machine and processor specific messages, and to perform the reverse operation on the receiving side. Figure 3 depicts a high level view of the stack. Since one of Legion's design goals is to provide flexibility and extensibility within a working implementation, we implemented the configurable protocol stack using the event mechanism described in Section 3.2. Figure 4 shows the shape of this implementation.

As an example, consider how an invoked (receiving) object passes messages up the communication protocol stack. When the data delivery layer of the receiver (the layer of the protocol stack that abstracts various low level communications APIs such as BSD sockets) receives a raw message off the wire, it creates a message-received event and attaches the raw message as the data part of the event. The message layer, the layer of the protocol stack that interprets raw messages to construct Legion message data structures, registers a handler with the message-received event kind. When the message layer handler is invoked, it extracts the raw message data from
the event, constructs a Legion message data structure, and attaches this Legion message as the new data associated with the event. Higher layers of the protocol stack (i.e. those with lower priority handlers for the message received event kind) will then be invoked in turn, and will manipulate the Legion message contained in the event.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Invocation Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Message</td>
</tr>
<tr>
<td>Data Delivery</td>
<td>Data Delivery</td>
</tr>
</tbody>
</table>

![Table: Reliable Communication](image)

Figure 3: LRTL’s default configurable protocol stack.

![Diagram: LRTL's default protocol stack using events](image)

Figure 4: Instantiation of LRTL’s default protocol stack using events.

### 4.3 Message and method logging

One useful feature of a wide area distributed object system is method- or message-level logging. Logging is important for a variety of reasons, including debugging, performance tuning, accounting, and replaying computation.

Suppose we have an object A that wants to know how many messages are sent on its behalf as part of a computation that it has initiated. We assume that there is already a message logger object in existence, or that A creates the message logger object before the computation of interest begins (Figure 5). Object A sets up a logging-on implicit parameter, and specifies which message logger object should receive the notification. This specification is contained in the value part of the implicit parameter either as the OID of the object, or as a self-contained program graph that will be executed by the receiving object when the message arrives. In either case, the receiver must register an event handler that looks specifically for a logging-on parameter, and if present, takes the appropriate action. Of course, both sender and receiver must agree on how to name the implicit parameter and how to interpret the attached value. Each message sent between objects (solid lines in Figure 5) carries an implicit parameter identifying a message logger to notify once the message is received. Dotted lines from each object to the message logger represent this notification. If B performs method calls on A’s behalf, then the implicit parameters that arrived with A’s method invocation should be propagated accordingly. LRTL’s implementation ensures that the correct implicit parameter lists are propagated. So if A calls B which calls C, then Battaches the implicit parameters it received from A, not its own default parameters.

Another possible use of implicit parameters is for visualization. For example, a graphical display might show the locations and communication patterns of a set of objects. Figure 5 illustrates one method for doing so. The message logger uses the message traces to drive a graphical display showing object locations and interactions. The locational informa-
tion needed to drive this geographical display might be specified by the communicating objects themselves or by some other entity that has knowledge of LOID to object address mappings.

![Diagram](image)

Figure 5: Using implicit parameters to provide logging and visualization information.

### 4.4 Active messages

The active messages programming model [10] is a message passing scheme that is intended to integrate communication and computation in order to increase the compute/communicate overlap, thereby masking the latency of message passing and increasing performance. The basic idea behind active messages is simple. Messages are prepended with the address of a handler routine that is automatically invoked upon receipt of the message. Active messages are not buffered and explicitly received, as is common with standard message passing interfaces. Instead, the receiving process invokes the handler routine specified for the message immediately upon message arrival. The handler may execute as a new thread of control, or may interrupt the running computation. The job of the active message handler is to incorporate the received message into the on-going computation.

In the current Legion implementation, when a method invocation is received by an object, it is normally inserted into a database of ready method requests (the invocation store), and a method-ready event is announced. Thus, it is entirely possible that the invocation request may be buffered for some time before it is handled.

A Legion version of active messages could be constructed by making Legion methods serve as message handlers, and by replacing the method-ready event handler with one that creates a new thread to service incoming methods instead of buffering them in an invocation store. This method-ready event handler would need to be registered with the method-ready event kind at object startup.

In some ways, the model supported here is more general than the traditional active messages model. For example, if a method (i.e., a handler) required two messages from different sources for activation, this requirement would be enforced in the invocation matcher, the level just below the invocation store. Programs might be entirely composed of standard single-token active messages, providing a programming model as flexible as the original. On the other hand, programs might also include multi-token active messages, for a more general programming model that might best be called "active methods."

### 4.5 Path expressions

The various method invocation semantics covered thus far have offered a "one size fits all" concurrency control mechanism. A more general approach to customizing the concurrency control requirements of operations on an object can be designed based on path expressions [11]. Path expressions permit the programmer to specify (1) sequencing constraints among operations; (2) selection (mutual exclusion) between operations; and (3) allowable concurrency between operations. These concurrency control primitives let programmers maintain the sequential consistency of their programs and at the same time indicate potential concurrency to a run-time environment.

Path expression based method sequencing could be implemented for Legion objects, again
by utilizing the inherent configurability of LRTL's protocol stack. As with active messages, supporting different method invocation semantics requires replacing the method-ready event handler. In this case, the method ready handler must examine the function numbers of available operations and determine if they may be safely fired given the ordering constraints specified by the program's path expressions. If a method can be fired safely, a new thread is created and allowed to run, starting at the entry point for the given member function (as in the active messages case). On the other hand, if the ordering constraints of a newly arrived method are not satisfied, the method must be buffered (e.g. in an LRTL provided invocation store) and later extracted and fired when safe. This need to defer the firing of methods requires that code be executed whenever methods complete execution. One possible way to satisfy this requirement is to use the method-done event kind, and announce events of this kind when methods complete execution. A handler for method-done events can then be used to re-evaluate buffered methods with respect to the path expression ordering constraints whenever a running operation completes.

The result of this configuration of LRTL would be a run-time environment that could be used to support path expression style method invocation semantics. This run-time system might be used explicitly by a programmer, or might be the target of a compiler that accepted a Path-Pascal like implementation language for Legion methods.

5 Related work

The majority of the techniques used by LRTL have been successfully applied in other systems. For example, events are well understood and have been used for a variety of purposes, from graphical user interfaces such as X Windows, to the construction of operating system kernels such as SPIN [12]. The versatility of an event-based abstraction resides in its ability to decouple communication between various components of a system both temporally and spatially. This feature is essential to LRTL, which is required to support a variety user requirements. A diverse set of dynamically configurable user and system components must interoperate—the event paradigm serves as the "glue" that binds these components together.

The design of LRTL was influenced by the x-Kernel project [13], which demonstrated the importance of unifying inter-layer communication within a single framework. While the emphasis was on building network protocols, the lessons are very much applicable to our goal of supporting diverse functionality in LRTL. The x-Kernel does not use an event-based abstraction for inter-layer communication though recent extensions [14] have added an event-based model for composing components within a layer. The result is a two-tiered framework wherein communication between components is no longer unified.

While the SPIN and x-Kernel projects seek to provide flexibility and extensibility at the kernel level, LRTL operates entirely in user space. This design decision reflects the Legion philosophy that we cannot mandate changes to underlying host operating systems. Except for low-level components in the transport layer, LRTL does not contain operating system specific code, and hence is portable to many operating systems and hardware architectures.

Message logging is an important aspect of distributed and parallel systems [15, 16]. While this feature has typically been hard coded into system interfaces (e.g. as in the PVM message logger facility [17]), Legion program graphs and implicit parameters provide a flexible and generic way of specifying information propagation between objects for the purpose of message logging, program visualization, and other meta-applications.

6 Summary

LRTL, in conjunction with the Legion core system objects and class system, provides a flexible and extensible foundation for constructing distributed meta-systems based on objects.
A prototype implementation of the core LRTL mechanisms as described in this paper has been built and is currently being used as the basis for experimental meta-systems construction and application programming. Recent Legion systems have been run across multiple institutions spanning the continental United States, and prototype applications have included domains such as multimedia, biochemistry, electrical engineering.

Future work on LRTL will provide advanced configurations to support alternate method invocation semantics, additional security features, and message logging as described in Section 4. While the core mechanisms and basic LRTL configuration have been implemented, advanced add-on features such as these are still under active development.

References


DISTRIBUTION LIST

1 – 3  Department of Energy
       Acquisition and Assistance Group
       Chicago Operations Office
       9800 South Cass Avenue
       Argonne, IL 60439

       Attention:    Fred Sienko, Contracting Officer

4 – 5  M. Rodeffer, Clark Hall

6 – 7  B. Spangel

8 - 9  SEAS Postaward Administration

10  SEAS Preaward Administration Files

JO#0602:ph