To appear in the *Journal of Parallel and Distributed Computing*, 2003.

Support for Extensibility and Site Autonomy in the Legion Grid System Object Model[1]

Michael J. Lewis[†], Adam J. Ferrari[*], Marty A. Humphrey[*], John F. Karpovich[*],

Mark M. Morgan[*], Anand Natrajan[*], Anh Nguyen-Tuong[*], Glenn S. Wasson[*] and

Andrew S. Grimshaw[*]

[*]*{ ferrari | humphrey | jfk3w | mmm2a | anand | nguyen | wasson | grimshaw }@cs.virginia.edu*
Department of Computer Science, University of Virginia
[†]*mlewis@binghamton.edu*
Department of Computer Science, SUNY - Binghamton

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

Abstract

Grid computing is the use of large collections of heterogeneous, distributed resources (including machines, databases, devices, and users) to support large-scale computations and wide-area data access. The Legion system is an implementation of a software architecture for grid computing. The basic philosophy underlying this architecture is the presentation of all grid resources as components of a single, seamless, virtual machine.

Legion's architecture was designed to address the challenges of using and managing wide-area resources. Features of the architecture include: global, shared namespaces; support for heterogeneity; security; wide-area data sharing; wide-area parallel processing; application-adjustable fault-tolerance; efficient scheduling and comprehensive resource management. We present the core design of the Legion architecture, with focus on the critical issues of extensibility and site autonomy. Grid systems software must be extensible because no static set of system-level decisions can meet all of the diverse, often conflicting, requirements of present and future user communities, nor take best advantage of unanticipated future hardware advances. Grid systems software must also support complete site autonomy, as resource owners will not turn control of their resources over to a dictatorial system.

---

1. The Legion project is partially supported by NFS CDA-9724552, NSF Career Award ACI-0133838, NSF EIA-9911099, 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

Grid systems are becoming increasingly important for current state-of-the-art applications in domains from high-performance scientific computing to business. Many of today's grid systems [29][22][7] are just beginning to attack the challenges of truly wide scale deployment across large numbers of sites that are only loosely connected and have no central managing body. For such systems, grid software—middleware above the physical resources and below end-user applications–is vital. Without grid systems software, constructing applications that exploit the full potential of these new grids will be difficult or impossible.

Critical requirements of grid systems include extensibility and site autonomy. The designers of grid systems software cannot predict the varied demands of present and future uses. Moreover, infrastructure changes are common, ranging from down-time, load changes and upgrades to new generations, new configurations and new access policies. A grid system must adapt to changing user demands and resource supplies, which requires that the grid systems software be constructed with extensibility as a design goal. Furthermore, grid systems can not restrict the ability of resource owners to control their resources. Truly large-scale grid systems cannot be achieved unless resource owners are allowed to fully control their resources (e.g., to determine who can use their resources, how and when they can be used, and how much it will cost). Such autonomy can also be used to manage complexity in grid systems. Since local personnel are typically more adept maintaining their resources than a grid-administrator, allowing them to dynamically alter resource access policy provides for better resource availability.

Legion, an object-based grid system developed at the University of Virginia, is based on a software architecture that addresses extensibility and site autonomy as first-class design requirements. Legion provides a single virtual machine view of heterogeneous resources that are part of a grid [15][16]. A Legion grid consists of workstations, vector supercomputers, parallel supercomputers and specialized equipment connected by a variety of networks. Legion tackles a wide range of distributed systems problems, including application-adjustable fault-tolerance, wide-area parallel processing, interoperability, scalability, security, efficient scheduling, and comprehensive resource management. An object-based approach supports the diverse policies and priorities of users, developers, and resource providers. Each grid resource is an object. Legion defines mechanisms for services such as object creation, naming, deletion, state management, migration and method invocation but does not mandate the implementation of these services or the policies for their use.

The primary contribution of this paper is to describe the properties of the Legion object model and architecture. Many important issues in Legion are beyond the scope of this paper. Specific areas of the Legion system have been described in depth, including the security model [5][8], scheduling [24], fault-tolerance [33][34], I/O systems [25][40], the run-time library architecture [9][38] and support for high-performance computing [14][30][31]. In this paper, Section 2 discusses the high-level Legion design, objectives, constraints, and philosophy. Section 3 presents the Legion object model and the concept of Legion classes. Section 4 introduces the core Legion system elements at a high level. Section 5 presents some basic Legion services and how they are achieved through the combined effort of system objects. We conclude with related work (Section 6) and a summary (Section 7).

2. Legion Objectives, Constraints and Philosophy

Given that computational grids are composed of resources that span multiple administrative domains, there exists a complex set of design principles for grid systems. These principles have

been the foundation of the Legion architecture.

- Site autonomy: Legion is composed of resources owned by many organizations, which properly insist on retaining control over their resources. For each resource the owner is able to limit or deny use by particular users, and specify when and how it can be used. An important aspect of autonomy is implementation selection, i.e. control over the code used to implement system services. Sites are able to choose or create their own Legion components to implement arbitrary resource policies.
- Extensible core: Legion must suit the wide variety of current user demands and be capable of evolving to meet unanticipated future needs. Therefore, mechanism and policy are realized via replaceable and extensible components, including and especially 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. For example, integrating Legion with the Storage Resource Broker [28] was a simple matter of extending the core system component that handles data storage to use the SRB as a back-end repository.
- Scalable architecture: Since Legion was designed to build grid systems consisting of millions of hosts and objects, it has a scalable software architecture. In other words, the system is fully distributed with no centralized structures or servers. This allows, for example, new hosts to be added to Legion without expensive grid-wide communication operations.
- Easy-to-use, seamless computational environment: Legion abstracts the complexity of its hardware environment and simplifies the communication and synchronization involved in parallel processing. Machine boundaries, for example, are invisible to users when using Legion's cross-platform MPI system.
- Single, persistent object space: Developing parallel and distributed systems in an environment without a single name space for accessing data and resources is still a significant obstacle for users of many grid systems. Having a multitude of disjoint name spaces greatly impedes developing applications that span sites. By contrast, all objects in the Legion system can transparently access all other Legion objects without regard to location or replication, but subject to security constraints. This is useful, for example, in creating code repositories where any host in the grid can execute an application stored there simply by knowing the name of that application (i.e., the application need not be stored on the local machine and the host need not know the physical location where the application is stored).
- Security for users and resource owners: Security is a fundamental aspect of grid systems, both to protect the integrity of user computations and to preserve the availability of resources. Legion was designed on the principle that security must be built firmly into the foundation of a grid computing system. Of course, no single security policy is suitable for all users and so Legion provides mechanisms that allow users and resource owners to select policies that fit their security and performance needs and meet their local administrative requirements. For example, Legion supports authentication based on Legion credentials and Kerberos [20].
- Fault-tolerance: In a large-scale grid system, resource failure (hosts, communication links, disks, etc.) is commonplace. Therefore, the Legion system itself was designed to address failures, through fault-tolerant components and dynamic system reconfiguration, as well as by providing mechanisms to support a wide range of user application fault-tolerance needs. For example, important system components that were executing on hosts that have failed, can be restarted by Legion on other, operable, hosts.

The objectives listed above were framed by several important practical constraints that guided our design. Legion was designed knowing that we:

- Cannot change host operating systems. Organizations will not permit their machines to be used if their operating systems must be replaced. Our experience with both Legion and Mentat [17] before shows that grid systems can be built on top of host operating systems.
- Cannot change network interface. Just as we must accommodate existing operating systems, the design of Legion assumes 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 require Legion software to run with the lowest possible privileges.

Our overall objective in the design of Legion was to create a grid system that was suitable to as many users and for as many purposes as possible. One thing was clear: a rigid system design—one in which policies were limited, trade-off decisions were pre-selected, or all semantics were pre-determined and hard-coded—would not achieve this goal. Indeed, if Legion dictated a single grid-wide solution to almost any of the technical objectives, we would have precluded 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 Class/Object Model

Legion is object-based in that each of its components is an object. Legion objects are independent of one another, which means that they are disjoint in address-space and communicate with one another via remote method invocation. Each Legion object belongs to a class and each class is itself a Legion object.

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 example, Legion classes are responsible for creating and locating their instances (objects of that class) 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 4), effectively eliminates the danger of imposing inappropriate policy decisions and provides a wide range of possibilities for the grid system developer.

Every Legion object is defined and managed by its class object. Class objects are *managers* and *policy makers* and have system-level responsibility for creating new instances, activating and deactivating them, and providing bindings (object address data that is useful in the context of the transport protocol) for client objects. Legion encourages users to define their own class objects; this can be as simple as a single command that creates a new Legion object for a user's existing application. These two features—class object management of their instances and the ability of 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.

Each Legion class defines the interface for its instances. This consists of the method calls that can be made on objects of that class. This interface includes the class-mandatory interface (Figure 1) and the class-specific functionality[1]. The class-mandatory interface encompasses the methods that determine various policies for the instances. This includes `createInstance()`, which creates a new instance of the class and returns the new location independent name for that instance (called a LOID or Legion Object Identifier - see section Section 5.1); `createMultipleInstances()`, which creates several instances of the class at once (useful for placing large numbers of parallel cohorts); and `activateInstance()`, which migrates an existing instance to a new location or restarts an instance that was deactivated to free resources. 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 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). An implementation is the Legion representation of an executable. The `getBinding()` functions support the translation of location-independent names to physical addresses of instances using the binding process as described in Section 5.1. There are several other functions (not shown in Figure 1) that allow clients to retrieve information about the location and characteristics of the class' instances, such as the instances' interface, their current host, and their current state (active or inert).

While classes allow users to specify policy and management information for objects in the Legion grid, it would be inconvenient for users to have to create a new class object for each new type of Legion object they wished to build. In fact, while the *instances* of a class often perform very different functions, class objects themselves perform similar functions. Therefore, Legion defines *metaclass objects*. Metaclass objects provide a mechanism for users to develop new objects, but reuse existing class object functionality. Metaclass objects are class objects whose instances are themselves class objects. Just as a normal class object manages and maintains implementations objects for its instances, so too does a metaclass object. When a user has a new object they wish to incorporate into the grid, they create a new instance of a metaclass object. This object is a class object that can then be used to manage the user's new object. To use a metaclass object, a

```
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 1. A subset of the Legion class-mandatory interface

1. Note that an object can disallow any function invocation request, typically based on the credentials of the caller (see section 5.4). This is especially relevant to the system-level functions implemented in core objects[5].

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. Legion provides a choice of metaclass objects, the instances of which export the same class-mandatory interface, but provide different functionality for that interface.

For example, some objects may be designed to service method invocations by creating a new instance to process each request. Other objects may instead redirect requests to one of a number of existing “service” objects for processing. Each of these “service” policies could be embedded in a metaclass. By creating a new instance of the appropriate metaclass, and populating it with implementations of a particular application, a developer can easily create an object with a particular service policy.

Class objects and metaclass objects help achieve extensibility and site autonomy in a Legion system. Legion does not mandate the kinds of objects that are permitted within a grid system. Users are free to create objects of their choice and incorporate them into Legion’s object management framework. Moreover, users may also choose between several object management systems. Finally, once users have incorporated their objects into Legion, they may create as many instances of their objects as they desire. In this respect, Legion provides grid system functionality similar to more recent web service systems such as Enterprise Java Beans [36] and servers such as JBOSS [23]. The policies that can be set on class objects often reflect the requirements of site autonomy. For example, it is common to set policies on class objects that restrict the creation of their instances to a subset of the machines in the grid.

4. Core Object Types

A Legion grid consists of a set of interoperable objects managed by their classes. Some objects encapsulate the applications that users wish to run on the grid. Other objects provide important system-level functionality that eases the process of developing application objects by providing abstractions of important elements in grid systems. We discuss these core object types and their role in a grid system below.

4.1 Host Objects

Legion host objects abstract processing resources in Legion. They may represent a single processor, a multiprocessor, a linux box, a Cray T3E, 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, deactivating 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.

Host objects are an important part of providing extensibility and site autonomy in Legion grids. In addition to implementing the host-mandatory interface (Figure 2), 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. This type of

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

FIGURE 2. Basic Legion host object interface.

extensibility allows Legion host objects to adapt to emerging execution models being defined for the Open Grid Services Architecture [10].

While host objects provide a uniform interface to different resource environments, they also provide a means for resource providers to enforce security and resource management policies within a Legion grid. 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 [32] demonstrating certain security properties, or rejecting implementations containing certain "restricted" system calls).

Consider two of our implementations for host objects. Our default host object 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 flaws. It is severely limited in terms of security—it executes all objects under a single Unix user id. While useful for creating "generic" accounts that allow access to many users, this type of host allows objects from different Legion users to potentially interfere with or examine one another's state. An alternative implementation extends the default host object to map Legion users to particular Unix user-ids when running different users' objects. This host object provides better interobject isolation and attribution of resource usage to users.

We have implemented a spectrum of host objects, e.g. hosts for batch queuing systems and hosts for Windows NT machines, that tradeoff 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.

4.2 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 (deactivated) objects on persistent storage. In a typical Legion system the number of objects may be orders of magnitude larger than the number of processors. Therefore, while some Legion objects will be *active* on processors managed by host objects, most objects will be *inert*, and have their relevant state stored in a vault object. 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. Vault objects provide for extensibility by presenting an abstraction of a storage resource. When a new

type of storage system needs to be added to the grid, a new vault object can be created. Since this new vault object will export the mandatory interface, it will be compatible with existing objects.

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 the "address" of its OPR (called the OPA for Object Persistent representation Address). 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 3). The `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

```
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 3. The Legion vault object interface.

`getOPRAddress()` method is used to determine the location of the OPR associated with any of its managed objects. The `giveOPR()` and `getOPR()` methods transfer a linearized (i.e., transmissible) OPR to and from vaults, respectively, facilitating object migration. The `deleteOPR()` method is used to terminate a vault's management of an OPR. Finally, `markActive()` and `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. Since vault objects control access to local storage resources, they can be used to implement site-specific policy with respect to those resources. For example, restricting usage to certain quotas, users, or objects, etc., further enables site autonomy.

4.3 Implementation Objects

Implementation objects encapsulate 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: `read()`, `write()`, and `sizeOf()` (Figure 4). Implementation objects

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

FIGURE 4. The Legion implementation object interface

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.

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. 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. One particularly useful aspect of this is that versions of a particular program, compiled on a variety of platforms, can be managed by a single class object. This, in turn, allows the user to execute their object on a variety of platforms without having to manage multiple binaries by hand, instead allowing Legion to automatically select the correct implementation for that platform.

The implementation objects provide extensibility by encapsulating “legacy applications”, i.e., applications which are not Legion-aware. By making arbitrary user code into a Legion object, Legion can distribute and manage that application across the grid! Collections of implementation objects and their associated class object allow Legion users to run applications written in any programming language, using any programming paradigm, on any computational resource for which the user has an executable.

4.4 Implementation Caches

Implementation caches avoid storage and communication costs by storing implementations for later reuse. The interface to the implementation cache object is depicted in Figure 5—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

```
class ImplementationCache {
   pathName getImplementation(LOID impl);
};
```

FIGURE 5. The Legion implementation cache interface

`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 almost as inexpensive as 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, will time-out and can be discarded from caches.

4.5 Binding Agents

Binding agents are objects which facilitate the translation of names (LOIDs) from Legion’s global, grid-wide namespace to actual addresses usable by transport protocols. Binding agents, in part, implement location-transparency in grid computing. This problem is addressed in the OGSA in the GSH-to-GSR mapping [10]. A binding agent is essentially a cache of LOID to object address (OA) mappings that is shared by all of its clients. Although every Legion object

has a cache of bindings for other Legion objects, the binding agent abstracts away the mechanism for determining a callee object's actual address if it is not in the caller's internal cache. While details of the Legion's naming and binding mechanisms are provided in section Section 5.1, Figure 6 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 object of the object with 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, which allows a client to pass in a stale binding (i.e. one for an object that has migrated or become inert) and directs the binding agent to get 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.

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

FIGURE 6. The Legion binding agent interface

Binding agents are not, strictly speaking, necessary for the correct execution of the binding process, since client objects can make the same sequence of method calls to determine a binding. 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 [39], thereby off loading some of the responsibility for providing bindings away from classes.

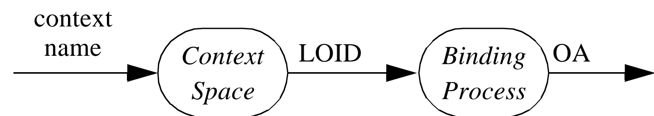
5. Key Legion Mechanisms

We have seen how users can create objects that make site or resource specific policy decisions using Legion's classes. The interaction of groups of Legion objects that make up a Legion grid implement various Legion mechanisms that other objects can use. These mechanisms include mechanisms for global naming/binding, scheduling, fault tolerance and security. Each is described briefly below and its impact on extensibility and autonomy is discussed.

5.1 Naming and Binding

Legion objects are identified through a three-level naming hierarchy, depicted in Figure 7. 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* (*LOID*s). For direct object-to-object communication, LOIDs must be bound to low-level *object addresses* (*OAs*) 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 7).

FIGURE 7. 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



Legion's global namespace provides for autonomy both within sites by not fixing the host on which an object resides. In fact, objects can be migrated between sites and references to those objects will be transparently redirected by the Legion binding process.

<u>LOIDs:</u> 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 usually contains a public key for encrypted communication with the named object (see section 5.4). 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.

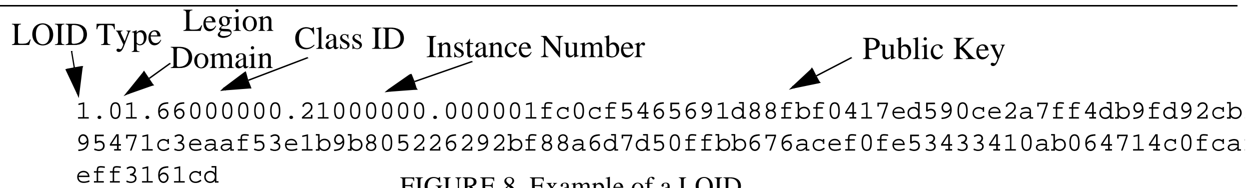
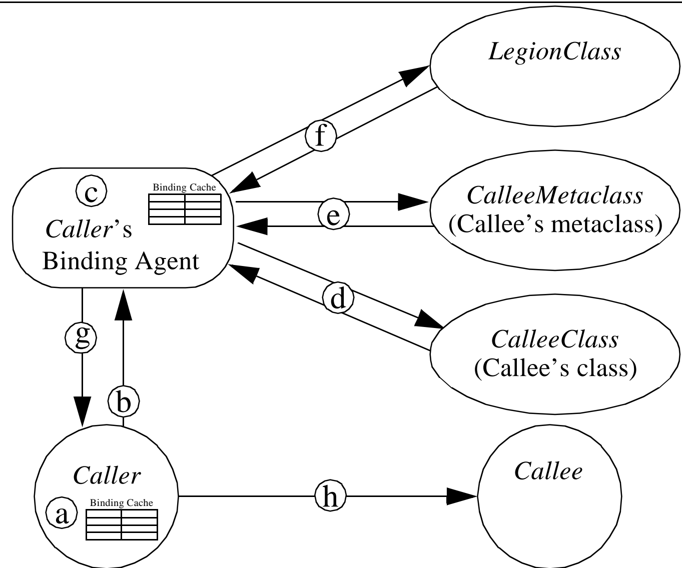


FIGURE 8. Example of a LOID

<u>Object Addresses:</u> 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 [18]. Alternatively, OAs could describe ports as defined in the OGSA [10].

<u>Bindings:</u> 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 can be passed around the system and cached within objects. The binding mechanism itself is best illustrated with the following example. Image one Legion object (the Caller) wishes to invoke a method of another Legion object (the Callee). The Caller must bind Callee's LOID to Callee's current OA. This process is

FIGURE 9. Potential steps in the Legion binding and class-of mechanisms—Caller must bind the LOID of Callee to an OA for low-level communication. Caller may already have a cached binding for Callee (a), or may need to consult a binding agent (b). The binding agent may have a cached binding for Callee (c), or may need to consult Callee's class, *CalleeClass*, for the binding (d). In order to communicate with CalleeClass, the binding agent needs a binding for CalleeClass. If the binding agent does not have CalleeClass's binding, it may need to consult CalleeClass's metaclass (e). If the binding agent does not know the binding for this metaclass, the process repeats itself. The recursion is guaranteed to terminate at the root of the binding tree, LegionClass (f). Eventually, the binding agent returns Callee's binding (g) and Caller can send messages directly to Callee (h).



depicted in Figure 9.

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 9a). 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. If a cache miss occurs, Caller contacts its *binding agent* (Figure 9b). If the binding agent does not have the requested binding, it can consult an alternative external source. It can forward the request to another binding agent or it can consult Callee's class object, CalleeClass, since class objects are responsible for knowing the current binding of all of their instances (Figure 9d). Determining an object's class is called the *class-of mechanism* and operates in a similar manner to the binding system, mapping objects to their classes (with the final class-map held by LegionClass).

Once the binding agent obtains CalleeClass's LOID, it can request Callee's binding from CalleeClass. However, the binding agent may need to begin another execution of 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.

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.

5.2 Scheduling

When a new instance is created, the class object must select an appropriate processor for that

instance. In general, this decision involves the available implementations for that object and dynamic system information such as CPU load, available memory, network bandwidth, etc. To assist in determining an appropriate placement, the Legion object placement mechanism can invoke scheduler objects. The integrated approach of the Legion architecture facilitates having a scheduler invoked for *every* object placement (user computation, data object, etc.)

A Scheduler queries a Collection object which contains static and dynamic information about grid resources. Using this information and its scheduling policy, the Scheduler creates an ordered series of mappings of objects to potential resources. These mappings are given to an Enactor object which communicates with Host and Vault objects to reserve the resources in the mapping. If reservations are successfully made on the resources in a mapping, the class object is directed to place its new instance on the selected resources. Otherwise, the Enactor attempts to reserve the resources for the next mapping (see [4] for more details). There can be many different scheduler objects implementing different scheduling algorithms and each scheduler object can serve different classes (though they can be replicated to prevent contention).

5.3 Fault Tolerance

As described in Section 3, the straightforward implementation of a class object's createInstance() function selects a host and vault pair for the object's associated process and OPR. This leaves the object potentially vulnerable to a particular host or disk failure. Legion provides an alternative class object that places copies of the OPR in multiple locations, any one of which is appropriate for activating the object. Further, a class that creates process replicants that coordinate with one another to represent a single object is easily realized within the model. This preference for fault tolerance is hidden behind the class mandatory interface; users simply call createInstance() as they would on any other class object. Since the binding process is controlled by the class object as well, bindings for different processes can be handed out in response to multiple getBinding() calls to the class. This can help make "multi-process objects" more scalable, in addition to being more fault-tolerant.

Another way that Legion enables fault tolerance is by allowing objects to write their persistent state to their OPRs as frequently as they determine appropriate. OPRs are intended to contain a snapshot of an object's persistent state information. If this snapshot is updated frequently, then upon failure of a process and reactivation of the associated object, less work will be lost. We have designed a simple set of functions, compiled into all Legion objects via the Legion run-time library, that allow programmers to access directly an object's associated OPR and write data into it from within user-level code. This allows, for example, restarting objects that the system has determined to be down (by timeouts on messages for example). If an object's state is accessible in the OPR, the object's class can spawn a new instance with the same OPR data, effectively making this new instance equal to the old instance. This object restart may also involve a new scheduling decision since objects to not have to be restarted on the same host where they were last.

In both examples above, the Legion model provides an appropriate hook for allowing programmers to achieve more fault tolerant applications, without requiring that they do so and to thereby pay the associated performance cost of fault tolerance. The kind and level of fault tolerance can be determined on an application by application basis, without altering core object interfaces.

5.4 Security

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 is independent of the underlying security mechanisms and policies of the host systems. Three concepts are described: identity and authentication, access control, and communication between objects.

Identify and Authentication: Identity is fundamental to higher-level security services such as access control. The security field of the LOID is largely used for this purpose (see Figure 8). Fundamentally, if each object has its own public-private key pair, each object has the ability, via public-key authentications mechanisms (e.g., RSA), to prove its own identity. Public keys can be self-signed or can be signed via Certificate Authorities (CAs). The LOID and its security implications have recently been the foundation of a grid computing naming scheme in the Global Grid Forum (GGF) called SGNP [2].

The 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.

In Grid computing, the user typically accesses resources indirectly, and objects need to be able to perform actions on his behalf. One way 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 providing necessary authentication information. 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 cripple performance.

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.

Access Control: Each Legion object is responsible for enforcing its own access control policy. The general model for access control is access is only available through method calls, and that each method call received at an object passes through a *MayI* layer before being serviced (see figure 10). 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.

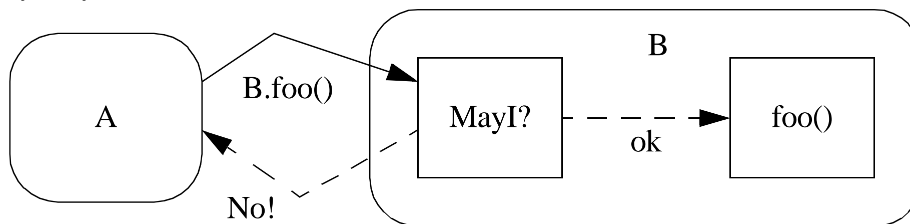


FIGURE 10. Legion Implementation of Access Control

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.

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 from one Legion object to another Legion object 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 with the public key of the recipient Legion object. 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 RSA.

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.

6. Related Work

Many recent grid computing projects address a portion of the issues addressed by Legion. Here, the discussion is focused on systems that attempt to provide comprehensive solutions, such as Globus [11][12] and Globe [37]. However, it is worth noting that these projects, Legion, and other grid computing projects are all outgrowths of the significant existing work in first-generation network parallel computing systems, such as PVM [13] and MPI [19], and in modern transparent distributed computing systems, such as the Berkeley NOW project [1] and DCE [27].

6.1 Globus

The Globus project [11][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. Both systems abstract access to processing resources: Legion via the host object interface; Globus through the Globus Resource Allocation Manager (GRAM) interface [6]. Both systems also support applications developed using a range of programming models, 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 combination of working components into a composite metacomputing toolkit.

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 number of services grows in such a system, the lack of a common programming interface to Globus components and the lack of a unifying model of their interaction 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 grid computing 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 in the long-term advantages of basing a grid computing system on a cohesive, comprehensive and extensible design.

6.2 Globe

The Globe [37] 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) [35] shares a number of elements with the Legion architecture, although it is not intended for grid computing. Similar to Legion's idea of many possible object implementations that share a common interface, 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

Grid systems are here and operating around the world. They are enabled by the tremendous increase in the available network bandwidth, the large number of available resources and the continuing demand for resources by users. Constructing grid system software to meet the needs of a diverse user and resource owner community is not easy; grid system software must 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.

Legion operates on a variety of platforms, ranging from workstations (e.g., Sun, SGI, IBM, DEC) and PCs (Linux over Alpha or Intel, Windows NT) to supercomputers such as the IBM SP2,

Cray T90, and SGI Origin 2000 and queuing systems such as LoadLeveler [21], LSF [41], and PBS [3]. More information about Legion is available at http://legion.virginia.edu.

References

[1] Anderson, T., Culler, D., Patterson, D., and the NOW team. 1995. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1): 54-64.

[2] Apgar, J., Grimshaw, A., Harris, S., Humphrey, M., and Nguyen-Tuong, A. 2002. Secure Grid Naming Protocol (SGNP). Global Grid Forum draft, February 2002.

[3] Bayucan, A., Henderson, R., Lesiak, C., Mann, N., Proett, T., Tweten, D. 1999. "Portable Batch System: External Reference Specification". Technical Report, MRJ Technology Solutions.

[4] Chapin, S., Katramatos, D., Karpovich, J. and Grimshaw, A. 1999. "Resource Management in Legion", *Future Generation Computing Systems*, vol. 15: 583-594.

[5] Chapin, S. Wang, C., Wulf, W. and Grimshaw, A. 1999. A Security Model for Metasystems. Future Generation Computing Systems.

[6] Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C. 2001. Grid Information Services for Distributed Resource Sharing. *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10).*

[7] European Union DataGrid Project, `http://www.eu-datagrid.org`

[8] Ferrari, A., Knabe, F., Humphrey, M., Chapin, S. and Grimshaw, A. 1999. A Flexible Security System for Metacomputing Environments, *High Performance Computing and Networking Europe*.

[9] Ferrari, A., Lewis, M., Viles, C., Nguyen-Tuong, A., and Grimshaw, A. 1996. "Implementation of the Legion library," University of Virginia Computer Science Technical Report CS-96-16.

[10] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. 2002. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Global Grid Forum Document.

[11] Foster, I. and Kesselman, C. 1999. The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann.

[12] Foster, I. and Kesselman, C., 1997. "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, 11(2): 115-128.

[13] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. 1994. "PVM: Parallel Virtual Machine". MIT Press.

[14] Grimshaw, A., Ferrari, A., Knabe, F., Humphrey, M. 1999. Wide-Area Computing: Resource Sharing on a Large Scale. *IEEE Computer*, 32(5).

[15] Grimshaw, A., Ferrari, A., Lindahl, G., and Holcomb, K. 1998. Metasystems. *Communications of the ACM*, 41(11).

[16] Grimshaw, A., Wulf, W., and the Legion team. 1997. The Legion vision of a worldwide virtual computer. *Communications of the ACM* 40(1).

[17] Grimshaw, A., Ferrari, A., West, E. 1996. "Mentat, Parallel Programming Using C++". MIT Press.

[18] Grimshaw, A., Weissman, J., and Strayer, W. 1996. Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. *ACM Transactions on Computer Systems* 14(2).

[19] Gropp, W., Lusk, E., and Skjellum, A. 1994. "Using MPI: Portable Parallel Programming with the Message-Passing Interface", MIT Press.

[20] Humphrey, M., Knabe, F., Ferrari, A., and Grimshaw, A. 2000. Accountability and Control of Process Creation in Metasystems. *Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS2000).*

[21] IBM Corporation. 1994. "IBM LoadLeveler: User's Guide (SH26-7226-02)," IBM Publication number ST00-9696.

[22] Information Power Grid, `http://www.ipg.nasa.gov`

[23] JBoss Group, `http://www.jboss.org`

[24] Karpovich, J. 1996. "Support for object placement in wide-area heterogeneous distributed systems," University of Virginia Computer Science Technical Report CS-96-03.

[25] Karpovich, J., Grimshaw, A., and French, J. 1994. Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O. *9th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA).*

[26] Lewis, M., and Grimshaw, A. 1996. The Core Legion Object Model. *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press.

[27] Lockhart, Jr., H. 1994. "OSF DCE Guide to Developing Distributed Applications", McGraw-Hill, Inc. New York.

[28] Moore, R., *et. al.* 2000. Collection-Based Persistent Digital Archives, *D-Lib Magazine* 6(3 & 4).

[29] National Partnership for Advanced Computing Infrastructure, `http://www.npaci.edu`

[30] Natrajan, A., Humphrey, M., and Grimshaw, A. 2001. Capacity and Capability Computing in Legion. *Intl. Conf. on Computational Science*.

[31] Natrajan, A., Crowley, M., Wilkins-Diehr, N., Humphrey, M., Fox, A., Grimshaw, A., and Brooks, C. III. 2001. Studying Protein Folding on the Grid: Experiences using CHARMM on NPACI Resources under Legion. *10th Intl. Symp. on High Perf. Dist. Computing*.

[32] Necula, G. 1997. Proof-Carrying Code. *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*: 106-119.

[33] Nguyen-Tuong, A. 2000. Integrating Fault-tolerance Techniques in Grid Applications. Ph.D. Dissertation, Computer Science Department, University of Virginia.

[34] Nguyen-Tuong, A, Grimshaw, A., and Hyett, M. 1996. Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System. *Proceedings of the 15th International Symposium on Reliable and Distributed Systems*: 1-11.

[35] Object Management Group. 1996. "The Common Object Request Broker: Architecture and Specification," Revision 2.0, July 1995 (updated July 1996).

[36] Shannon, B., Hapner, M., Matena, V., Davidson, J., Pelegri-Llopart, E., and Cable, L. 2000. Java 2 Platform, Enterprise Edition: Platform and Component Specification. Addison-Wesley.

[37] van Steen, M., Homburg, P., and Tanenbaum, A. 1997. “The architectural design of Globe: a wide-area distributed system,” Internal report IR-422, Vrije Universiteit.

[38] Viles, C., Lewis, M., Ferrari, A., Nguyen-Tuong, A., and Grimshaw, A. 1997. Enabling flexiblity in the Legion run-time library. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’97)*: 265-274.

[39] Yew, P., Tzeng, N., and Lawrie, D. 1987. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4).

[40] White, B., Grimshaw, A., and Nguyen-Tuong, A. 2000. Grid-Based File Access: The Legion I/O Model. *High Performance Distributed Computing* 9.

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