Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems[1]

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

Technical Report CS-98-12

June 3, 1998

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, focussing 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

Table of Contents

1. Introduction

The next several years will see the widespread deployment of high-speed gigabit and terabit networks, both as national and international backbones and as local, campus, and metropolitan area networks. These networks will act as sinews to bind together computation and data resources throughout the world. The challenge facing the computer science community is to provide software abstractions that can glue the diverse resources into a single entity—to construct one system from many. We call this *metasystems* software: the software above the physical resources and below applications. Without metasystems software it will be difficult if not impossible to construct the applications that can unleash the full potential brought about by the new, more powerful networks.

The design of these new metacomputing software abstractions is the subject of this paper. We believe that, above all else, metasystem software must be extensible and must provide complete site autonomy. It must be extensible because no single system can meet all of the diverse, often conflicting, needs and requirements of the entire present and future user community, nor can a system constructed today take best 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 for the simple reason that resource owners will not turn their resources (hosts, databases, devices, etc.) over to a dictatorial system as the price of admission. Metasystems software must instead allow resource owners to decide who can use their resources, which binaries a user can execute on their processors, how much of a given resource a user can access, how much it will cost, and so on.

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 will consist of workstations, vector supercomputers, and parallel supercomputers connected by local-area and wide-area networks, but the system is designed to provide users with the illusion of a single virtual machine. This 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. We envision a system in which a user sits at a Legion workstation and has the illusion of working on a single, very powerful computer. When the user invokes an application on a data set, it is Legion's responsibility to transparently schedule application components on processors, manage data transfer and coercion, and provide communication and synchronization. System boundaries, data location, and faults are invisible.

Legion's components 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 high-level languages, and will transparently create, schedule, and utilize distributed objects to execute the programs. Legion users will require a wide range of services in many different dimensions, including security, performance, and functionality. No single policy or static set of policies will satisfy every user, so users must be allowed to determine their own priorities and to implement their own solutions as much as possible. Legion supports this philosophy by providing the mechanisms for system-level services such as object creation, naming, binding, and migration, and by not mandating these services' policies or

implementations. We explain this philosophy in more detail in Section 3 and throughout Section 6.

1.1 Outline

The primary purpose of this paper is to describe the Legion run-time architecture. Sections 2 and 3 describe our high-level objectives and philosophies and explain the motivation for the Legion object model and architecture. Section 4 then introduces some key Legion concepts that are required for the remainder of the discussion. Section 5 introduces the core system elements by explaining the behavior of a simple narrative example, an RPC-style interaction between two Legion objects.[1] This discussion demonstrates how the components of Legion work together at a high level. In Section 6 we discuss in detail the interface and functionality of the core system objects: class objects, metaclass objects, host objects, vaults, context objects, binding agents, implementation objects, and implementation caches. These objects combine to implement basic Legion services. For each core object, we discuss examples of the options that are available to programmers who wish to augment or replace different parts of our implementation. Section 7 discusses the current status of the Legion implementation effort. We conclude with related work (Section 8) and a summary (Section 9).

2. Legion Objectives

To realize the Legion vision is not a trivial matter. We have distilled ten design objectives that are essential to the success of the project: site autonomy; an extensible core; scalability; an easy-to-use, seamless computational environment; high performance via parallelism; a single persistent object space; security for both users and resource providers; resource management and exploitation of heterogeneity; multi-language support and interoperability; and fault tolerance.

1. This is a “simplified” example because Legion does not restrict objects to using RPC-style interactions. On the contrary, Legion allows objects to create and execute generalized macro data-flow *program graphs*, enabling a much richer set of semantics for object interaction.

- **Site autonomy**: Legion will not be a monolithic system; it will be composed of resources owned and controlled by an array of organizations. Organizations, quite properly, will insist on having control over their own resources; for example, they may insist on specifying how much of a particular resource can be used, when it can be used, and who can and cannot use the resource. One important aspect of site autonomy is implementation autonomy. Sites must be able to choose which implementations of Legion components to use. For example, users may trust the security mechanisms of one implementation over those of another, or they may require the performance guarantees of a particular implementation.
- **Extensible core**: A metacomputing system must be flexible enough to suit the wide variety of current user demands and capable of evolving to meet unanticipated future needs. Mechanism and policy must be realized via extensible replaceable components. This will permit Legion to adapt over time and will allow users to construct their own mechanisms and policies to meet specific needs. Consistent with our site autonomy objective, the core system components themselves must be extensible and replaceable. This will allow third party or site-local implementations that provide value-added services to the system.
- **Scalable architecture**: Because Legion will consist of millions of hosts, it must have a scalable software architecture; there must be no centralized structures or servers—the system must be fully distributed.
- **Easy-to-use, seamless computational environment**: Legion must mask the complexity of the hardware environment and of communication and synchronization of parallel processing. Machine boundaries, for example, should be invisible to users. As much as possible, compilers acting in concert with run-time facilities must manage the environment for the user. If Legion is not transparent and easy to use, then it will provide little benefit over the

status quo and will not be used. Tempering our transparency objective is the knowledge that there are “power users” with demanding applications that will require the capability to make low-level decisions and to interface with low-level system mechanism. Therefore we must accommodate both end users who don’t want to worry about the details, and power users who are compelled to tune their applications.

- **High performance via parallelism**: Legion must support easy-to-use parallel processing with large degrees of parallelism. This includes arbitrary combinations of task and data parallelism. Not all applications will be parallel; Legion will necessarily best support relatively coarse-grain applications. This does not mean that we think a single huge application will ever use all of the computers in the country; most parallel applications will use only a small subset of the available resources at any given time.
- **Single, persistent object space**: The lack of a single name space for data and resource access is one of the most significant obstacles to wide-area parallel processing. The current multitude of disjoint name spaces makes writing applications that span sites extremely difficult. Any Legion object should be able to access transparently (subject to security constraints) any other Legion object without regard to location or replication.
- **Security for users and resource owners**: Security must be built firmly into the core of a metacomputing system. Attempting to patch security on as an afterthought (as is being attempted today in many contexts) is a fundamentally flawed approach. We also believe that no single security policy is perfect for all users. Although we cannot significantly strengthen existing operating system protection and security mechanisms (because we cannot replace existing host operating systems), we must ensure that existing mechanisms are not weakened by Legion. Therefore, we must provide mechanisms for users to select policies that fit their

needs and meet their local administrative requirements.

- **Management and exploitation of resource heterogeneity**: Clearly, Legion must support interoperability between the heterogeneous hardware and software components that will be used in the system. In addition, some architectures are better than others at executing particular applications, e.g., vectorizable codes. These affinities, and the costs of exploiting them, must be factored into scheduling decisions and policies.
- **Multiple language support and interoperability**: Legion applications will be written in a variety of languages. It must therefore be possible to integrate heterogeneous source-language application components in much the same manner that heterogeneous architectures are integrated. Interoperability also means that we must be able to support legacy codes and work with emerging standards such as CORBA [32] and DCE [28].
- **Fault-tolerance**: In a system as large as Legion it is certain that at any given instant several hosts, communication links, and disks will have failed. Thus, dealing with failure and with dynamic re-configuration is a necessity for both Legion system-level objects and the applications they support.

In addition to these purely technical objectives, there are also political, sociological, and economic issues, such as encouraging the participation of resource-rich centers and discouraging the temptation to take advantage of free resources without making reciprocal contributions to the resource pool. We intend to develop mechanisms that facilitate accounting policies to encourage good community behavior.

2.1 Constraints

In addition to the goals described above, several constraints restrict our design.

- **We cannot replace host operating systems**. Organizations will not permit their machines

to be used if their operating systems must be replaced. This would require rewriting applications and retraining users. It could also make Legion resources incompatible with and unavailable to other resources in the organization. Our experience developing the Mentat system [16] indicates that building a metasystem on top of existing host operating systems is a viable approach.

- **We cannot legislate changes to the interconnection network**. We must initially assume that the network resources and the protocols in use are fixed. Much as we must accommodate operating system heterogeneity, we must live with the available network resources. However, we can layer better protocols over existing ones, and we can warn users that the performance for a particular application on a particular network will be poor unless the protocol is changed.
- **We cannot require that Legion run in privileged mode**. To protect their objects and files, most Legion users will want the Legion software to run with the fewest possible privileges. Of course, we do not prohibit Legion implementations that require root privilege; this may provide some additional benefit and may be acceptable to some sites.

3. Philosophy

Complementing our use of the object-oriented paradigm is one of our driving philosophical themes: we cannot design a system that will satisfy every user's needs. We must design Legion to allow users and programmers the greatest flexibility in their applications' semantics, resisting the temptation to dictate solutions to a wide range of system functions. Users should be able, whenever possible, to select both the *kind* and the *level* of functionality, and to make their own trade-offs between function and cost.

Neither the kind nor the level of functionality are linearly ordered, but a simplistic model

is that of a multi-dimensional space. The needs of users will dictate where they need and/or can afford to be in this space; we as the designers of the supporting conceptual system have no way of knowing what those needs are, or what they will evolve to be in the future. Indeed, if we were to dictate a single system-wide solution to almost any of the issues raised in our list of objectives, we would preclude large classes of potential users and uses.

Consider security, with respect to both kind and level of functionality. Some users are more concerned with privacy, some only need to maintain the integrity of their data, and others require both of these types of security. Banks and hospitals, for example, are likely to fall into the last category. There is a difference between the kind of security functionality and the degree, or level, of security. Some users are content with password authentication, while others feel the need for more stringent user identification, such as signature analysis, fingerprint verification, or another approach. A user might be willing to pay the higher cost (in terms of CPU, bandwidth, and time) of a more powerful cryptographic key in order to have a stronger degree of security without changing the basic nature of the type of security provided. On the other hand, an application that requires low overhead cannot afford such a policy and should not be forced to use it. Such an application might instead choose a light-weight policy that merely verifies communication integrity or perhaps one with no security at all. Users decide what trade-offs to make, whether by implementing their own policies or by using existing policies, instead of coping with an inevitably unsatisfactory fixed security mechanism.

Next, consider the issue of consistency semantics in a distributed file system. To achieve good performance, it is often desirable to replicate all or parts of a file. If updates to the file are permitted, the replicated data may begin to diverge. There are many ways to address this problem: do not replicate writable files, use a cache invalidation protocol, use lazy updates to a

master copy, and so on. Each has an associated cost and semantics. Some applications don't require all copies to be the same, others require a strict “reads deliver the last value written” semantics, others know that the file is read-only (so that consistency protocols are a waste of time), while still others may need different semantics for the file in different regions of the application. Independent of the file semantics, some users may need frequent automatic backups and archiving, while others may not. The point is that the user—not the system—should make these decisions for users; users themselves should select the kind and level of service they require.

This philosophy has been manifested in the system architecture. The Legion object model specifies the composition and functionality of Legion's core objects (the objects that cooperate to create, locate, manage, and remove objects from the Legion system). Legion specifies the functionality but not the implementation of the system's core objects. Therefore, the core consists of extensible, replaceable components. The Legion project provides default implementations of the core objects, but users are not be obligated to use them. Instead, users are encouraged to select or construct objects that implement mechanisms and policies that meet their specific requirements.

The object model provides a natural way to achieve this kind of flexibility. Files, for example, are not part of Legion itself. Anyone may define a new Legion object type whose interface and semantics are recognizable as those of a file, but whose specifics suit its intended application. The current Legion software system provides an initial collection of file objects that reflect the most common needs, but we do not have to anticipate all possible future requirements.

4. The Object Model and Key Legion Concepts

Legion is an object-oriented system comprising independent, logically address space disjoint

objects that 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).[2]

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

In the Legion object model, each Legion object belongs to a class, and each class is itself a Legion object. All Legion objects export a common set of *object-mandatory* member functions that are necessary to implement the core Legion services (such as **`deactivate()`** and **`getInterface()`**). 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 important role of Legion classes. In Legion, 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 simply provide mechanisms for user-level classes to implement the policies and

2. MPL [16] 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 4.3).

algorithms that they choose. Assuming that we have defined the operations on core objects appropriately (i.e. that they are the right set of primitive operations to enable a wide enough range of policies to be implemented), this philosophy effectively eliminates the danger of imposing inappropriate policy decisions, and opens up a much wider range of possibilities for the application developer.

4.1 Naming and Binding

Legion objects are identified using a three-level naming hierarchy, as 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 system-level, unique, location-independent binary names called *Legion object identifiers* (*LOID*s). For direct object-to-object communication LOIDs must be bound to low-level addresses that are meaningful within the context of the transport protocol that will be used for message passing. These low-level addresses are called *object addresses* and the process by which LOIDs are mapped to object addresses is called the Legion *binding process*.

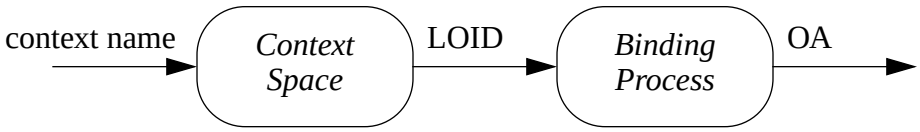

FIGURE 1. The three-level Legion naming hierarchy. Context names are convenient user-defined textual identifiers. These map to Legion object identifiers (LOIDs): system-wide unique, location-transparent object identifiers. For direct communication, LOIDs are mapped to low-level object addresses.

4.1.1 LOIDs

LOIDs are the central, system-level naming mechanism in the Legion system. Every Legion object is assigned a unique LOID that can be used to communicate with the object. The basic LOID data structure consists of a sequence of binary string *fields*; a LOID can contain up

to 2^{16}-1 fields, each of which may contain up to 2^{16}-1 bytes of arbitrary binary information. In addition to containing the information associated with each field, the LOID also encodes the number of fields it contains and the size of each field. Each LOID also contains a type identifier, a four byte string used to describe the meaning of the LOID contents (for example, to determine the semantics of certain content fields).

A LOID contains an initial four byte type identifier, followed by a two byte unsigned integer indicating the number of fields, followed by the fields themselves, as depicted in Figure 2. Each field is effectively a two byte unsigned integer indicating the field length, followed by the bytes that make up the field. Of course, the implementation of various LOID data structures may differ from this model, but the implied information content will be preserved.

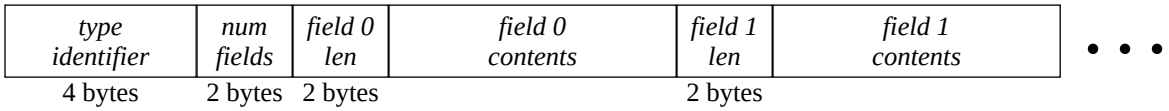

FIGURE 2. The LOID data structure contents.

Within the abstract LOID data type, four of the fields are reserved for specific system purposes. The first three reserved fields play a key role in the LOID to object address binding mechanism. Field 0 contains a Legion *domain identifier*, which can be used to support the dynamic connection of separate existing Legion systems. Field 1 is a *class identifier*, a string of bits uniquely identifying the named object's class. Field 2 is an *instance number* that distinguishes the named object from other instances of its class within the same Legion domain. LOIDs containing an instance number field of length zero are defined to refer to class objects.

The fourth field of the LOID (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.

4.1.2 Context names

Whereas LOIDs provide the basic system-level naming abstraction, users require a more natural naming mechanism, one that allows them to assign meaningful, human-readable names to their objects. Legion supports the notion of context space (directed graphs of *context objects* that cooperate to translate user-defined names into LOIDs) to fill this role. For example, an object that represents a processing resource might be assigned a context name corresponding to that host's standard DNS. An object that represents a file might be assigned a descriptive context name based on the file contents. Context space is discussed in greater detail in Section 6.6.

4.1.3 Object Addresses

Legion uses standard network protocols and communication facilities of host operating systems to support communication between Legion objects. However, LOIDs are meaningful only at the Legion level, not within existing protocols such as TCP/IP. Consequently, Legion must provide a mechanism by which LOIDs can be mapped to names that are meaningful to underlying protocols and communication facilities. These low-level names are called object addresses, or *OA*s. An OA is a list of *object address elements* and an *address semantic* field, which describes how to use the list. An OA element contains two basic parts, a 32-bit *address type* field, and the address itself. The address type field indicates the type of address that is contained in the address field, whose size and format vary depending on the address type. For example, the current Legion implementation contains an OA that consists of a single OA element. This element contains a 32-bit IP address and a 16-bit port number; every Legion object is linked with a Unix-sockets-based data delivery layer (called the Modular Message Passing System, or MMPS [17])

that communicates with the data delivery layers of other objects using these OA types.

The address semantic field is intended to encapsulate various forms of multicast and replicated communication. For example, the field could specify that all addresses in the list should be selected, that one of the addresses should be chosen at random, that k of the N addresses in the list should be used, etc. The composition and meaning of the full set of options that will be defined by Legion have not yet been identified, but provisions for extending the list with user-definable address semantics will likely be made.

4.1.4 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 (this field may also be set to some value that indicates that the binding will never become explicitly invalid). Bindings are first-class entities that can be passed around the system and cached within objects.

Note, the third field—the binding invalidation time—is strictly an optimization hint. A binding may still be used after the timeout appears to expire at a client—the binding may simply no longer be valid, leading to a communication timeout and rebinding. On the other hand, a client could use the timeout information to schedule re-binding in advance in order to avoid communication delays. Thus, the fact that there is no globally accurate notion of time does not affect correctness, just performance.

4.1.5 Object States

In a typical Legion system, the number of objects is expected to be orders of magnitude larger than the number of available processors. Thus, it would be unreasonable for our design to require an active process for every object in the system, although this would be the naive

approach to implementing the disjoint address space model.[3] To address this issue Legion objects are persistent, and alternate between two different states, *active* or *inert*. When an object is active, it is running as a process on a Legion host and it can be accessed via an object address. When an object is inert, it exists in persistent storage that is controlled by a Legion *vault object* (Section 6.3), is described by an *object persistence representation (OPR)*, and can be located using an *object persistence address (OPA*). Throughout their lifetime, objects can be moved between active and inert states by other Legion objects.

An OPR is associated with each Legion object and is used to store an object's persistent state (see Figure 3). Legion objects implement an internal **`saveState()`** method, which enables them to store persistent state into their OPR before becoming inert, and an internal **`restoreState()`** method, which is called immediately after reactivation to recover needed state from the OPR. Through the use of these object-internal mechanisms, in cooperation with system management of OPRs, objects are given the opportunity to preserver their state when they are migrated between hosts.

The OPA of an inert object is analogous to the OA of an active object. Objects use their OPA to gain direct access to their OPR. Typically, an OPA will be a file name (or a set of file names), and will necessarily only be meaningful to the Legion vault that controls the named

3. Legion does not specify that each object will necessarily have its own process. Our current implementation has one process per active object, but future alternative implementations may have the ability to multiplex objects to processes. However, even assuming multiple objects per process, we expect the number of objects to exceed the ability of the system to support active processes.

OPR and to the object with which the OPA is associated.

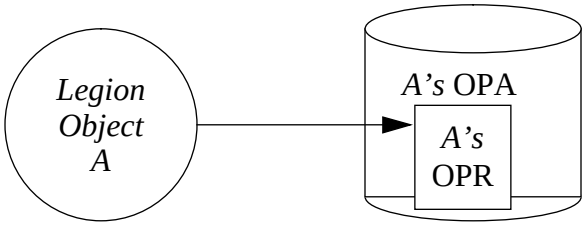

FIGURE 3. A Legion object has direct access to its persistent state, termed its object persistence representation, or OPR. The location of an object's OPR is called the object persistence address, or OPA.

4.2 Attributes

Legion *attributes* provide a general mechanism that allows an object to describe itself to other objects in the system. An attribute is a tuple that contains a *tag* and a *value*; the tag is a character string, and the value contains data that varies for different tags. Attributes reside in the address space of the object they describe, and can be accessed via object-mandatory functions (such as **`getAttributes()`**) whose parameters can be empty, can contain a specific tag or set of tags, or can contain <tag, value> pairs. The set of attribute kinds (i.e., tags) supported by an object depends on the object's type. For example, each host object contains attributes that describe the architecture and system configuration of the machine it represents (e.g., <Architecture, Sparc>, <Operating System, Solaris>). Programmers can include arbitrary attribute sets in the objects they create. During the lifetime of an object, its attributes can be dynamically updated or augmented to reflect changes in the object.

4.3 Legion Programming

Legion is designed to support and allow interoperation between multiple programming models. At its base, Legion prescribes the message format for inter-object communications, thereby enabling a wide variety of Legion object implementation strategies. However, in its most useful form, Legion presents programmers with high-level programming language interfaces to the system. These high-level interfaces are supported by Legion-targeted compilers, which in turn

use the services of a run-time library that enables Legion-compliant inter-object communication.

We have implemented a *Legion run-time library (LRTL)* [36], and we have ported the *Mentat [16] programming language compiler (MPLC)* to use the LRTL.[4] Thus, programmers can define Legion object implementations in MPL, which can be translated by MPLC to create executable objects that communicate with one another in a Legion-compliant fashion. Figure 4 depicts this typical programming model.

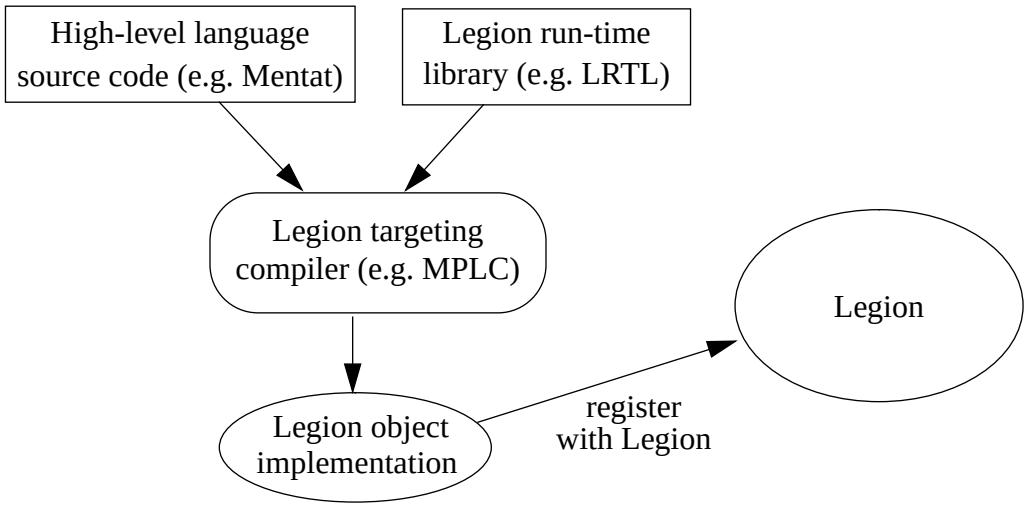

FIGURE 4. The Legion programming model. The object is defined in a high level language. Through a combination of Legion-aware compiler support and use of the LRTL, a complete Legion object implementation is produced. Based on this object implementation, new Legion objects can be instantiated in the system. These new objects can communicate with one another and with existing Legion objects.

5. An Illustrative Example

We have described the Legion object model and fundamental features such as naming and persistence, but we have not yet considered the design of system services such as object creation and LOID to OA binding. In Legion, these system level services are supported by a cooperating set of core objects, which are described in detail in Section 6. However, before examining the interfaces and designs of the individual system-level Legion objects, it would be useful to have a

4. We have also developed implementations of PVM [14] and MPI [21, 29] layered on top of the LRTL, and we currently support a Java interface to the LRTL and a specialized programming interface for Fortran called Basic Fortran Support (BFS) [11].

high-level understanding of the roles these objects play and their interrelationships.

To this end, this section describes how Legion implements a simplified RPC-style interaction between two Legion objects, *Caller* and *Callee*. The description introduces the basic functionality that the supporting Legion core objects must support, but does not describe them in intricate detail, nor does it expound on any alternative policies and implementations that are allowable under the architecture and object model. These discussions are deferred until Section 6.

Suppose Legion object Caller wishes to invoke member function **`func()`** on another Legion object, Callee (see Figure 5). The typical chain of events consists of several parts:

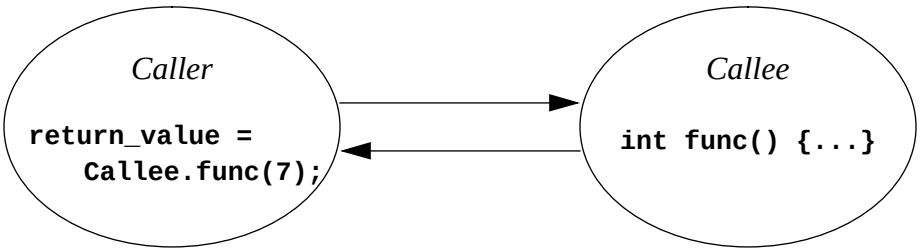

FIGURE 5. A simple RPC-style interaction between two Legion objects.

determining the LOID for Callee, obtaining a binding for this LOID, and potentially creating or activating Callee.

5.1 Determining LOIDs

The programmer, in writing the source code for object Caller, uses the appropriate programming language constructs to indicate that **`func()`** should be invoked on object Callee. For example, in MPL, the programmer would simply include the following line of code in the program:

```
return_value = Callee.func(7);
```

where **`return_value`** is a variable of the type that **`func()`** returns (in this case an integer). The compiler generates code in the translated implementation of Caller to marshall the integer

argument, to construct a simple macro data-flow program graph [36] representing the function call, to translate the graph into a Legion message, and to retrieve the return value and place it in the **`return_value`** variable.

To send a message to Callee, the compiler must associate the variable **`Callee`** with the appropriate Legion object, which entails finding Callee's LOID. One possible strategy is to statically translate the programmer-specified name for Callee (in this case **`Callee`**) into a context name that the compiler can use to identify the object. This context name can then be resolved—either statically within the compiler itself, or dynamically within the object being created (with potentially different results in each case)—to determine Callee's LOID (the LRTL provides routines for using Legion context objects to resolve context names to LOIDs).

In this example we have assumed that the program wishes to use an existing object. In an alternative scenario, Caller might create a new object to associate with the variable **`Callee`** (for example, when that variable comes into scope). In that case, the object creation mechanism would result in a LOID being returned to Caller. In any event, we now assume that the caller has determined Callee's LOID.

5.2 The Binding Mechanism

Once Callee's LOID has been determined, Caller must bind this LOID to Callee's current OA in order to carry out the desired method invocation. Finding a binding for a LOID is called the binding mechanism, and is depicted in Figure 6.

If Caller has communicated with Callee 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 6a). Legion object binding caches are populated by bindings collected during the repeated execution of the binding mechanism. These caches allow objects to

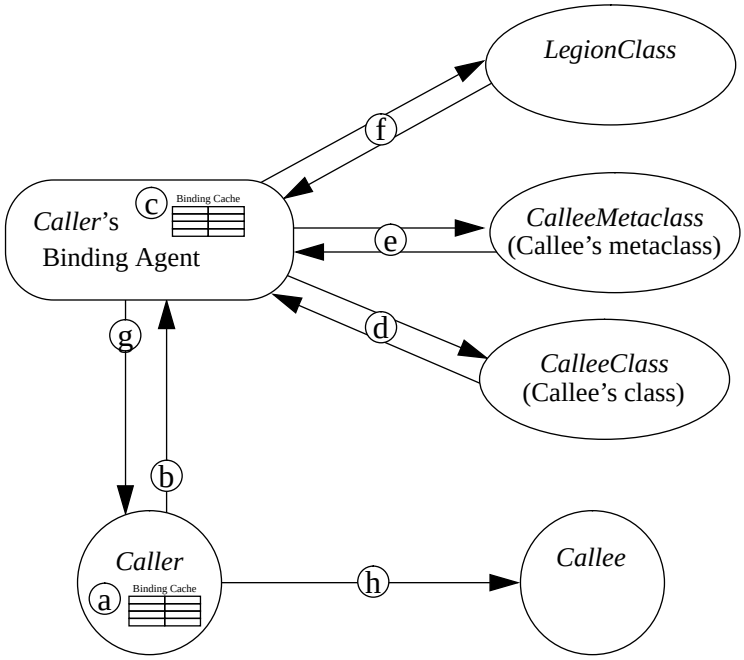

FIGURE 6. Potential steps in the Legion binding and class-of mechanisms—Caller must bind the LOID of Callee to an OA for low-level communication. Caller may already have a cached binding for Callee (a), or may need to consult a binding agent (b). The binding agent may have a cached binding for Callee (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).

take advantage of the natural temporal and spacial locality observed in method invocation; that is, if Caller invokes a method on Callee, Caller is likely to invoke other methods on Callee in the near future. If Caller does have a binding for Callee, it simply uses the OA in that binding, and the binding mechanism does not require any remote invocations on other Legion objects. Given the natural locality in method invocation, and the low cost of maintaining relatively large binding caches, we expect the use of cached bindings to be the common case. If Callee becomes inert or if it migrates to a new OA after its binding is cached in Caller, then the binding becomes *stale*, and Caller must obtain the up-to-date OA. Detecting stale bindings and obtaining current OAs is discussed in Section 5.4.

If Caller does not have a cached binding for Callee, Caller can contact its *binding agent*, whose job is to return bindings for its clients (Figure 6b). Typically, many objects will be clients

of the same binding agent, allowing shared caching of the results of costly binding requests. Thus, if any other objects using Caller's binding agent had recently invoked Callee, the binding agent may have a cached binding for Callee that it can immediately return to Caller. If the binding agent does not have a cached binding for Callee, it can contact Callee's class, CalleeClass, to obtain the desired binding (Figure 6d). Finding the LOID of a Legion object's class is called the *class-of mechanism*, and is described in Section 5.3.

Once the binding agent obtains CalleeClass's LOID, it can request Callee's binding from CalleeClass. However, to do so, 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, but the binding and class-of mechanisms are recursive, and because the class-of hierarchy is rooted at LegionClass, the mechanism is guaranteed to terminate.

5.3 The Class-Of Mechanism

The binding mechanism may need to consult an object's class; therefore it needs to be able to determine that class's identity. 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*. If Callee is not itself a class object, then objects can use the fact that CalleeClass's LOID contains the same class identifier as Callee's LOID, and contains an empty instance number (as mentioned in Section 4.1.1). Thus, the binding agent can search through its binding cache for a LOID with these characteristics, and can assume that any such LOID is that of CalleeClass.

As in the binding mechanism, a desired class-of result may not be cached. In this case, the class-of mechanism is performed through a binding agent. As they do for bindings, binding agents provide a shared caching mechanism for class-of results. If the desired class-of result is

not cached locally or in the binding agent, the class-of caller (in our running example, Caller'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 and unchanging object address. In practice, LegionClass will be distributed over multiple cooperating processes, and the class map will be highly replicated. 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 need not incur the overhead of maintaining cache coherence.

5.4 The Rebinding Mechanism

If Callee is a valid object (i.e. Callee maps to the LOID of an object that was created and has not yet been destroyed), Caller will be able to obtain a binding. However, as noted earlier, bindings (whether they come from binding caches, binding agents or class objects) can become stale. Specifically, the Callee's binding might contain an OA at which Callee no longer resides. When this happens, Caller determines that the binding is stale (typically by noticing repeated failed attempts to communicate with Callee at its old address) and 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 that contains 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. The binding agent may attempt to verify that the binding is stale, or might immediately defer to CalleeClass. In any event, assuming that Callee has just changed its OA (i.e., only stale information about the location of Callee is currently cached in its clients and their binding agents), CalleeClass will be consulted and will again serve as the ultimate authority for locating

its instances.

5.5 Object Activation

In Section 5.2 and Section 5.4, we based our discussion of the binding process on the fundamental assumption that classes could always return a valid OA for their instances. However, as described in Section 4.1.5, objects may be inert, and thus will be 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. In that case, the binding process would result in a call to CalleeClass to obtain a new binding for Callee. At this point, CalleeClass recognizes that Callee is inert, and employs the *object activation mechanism* to move Callee into the active state. Only by activating Callee can CalleeClass obtain a valid binding to return to either Caller or a binding agent operating on Caller's behalf.

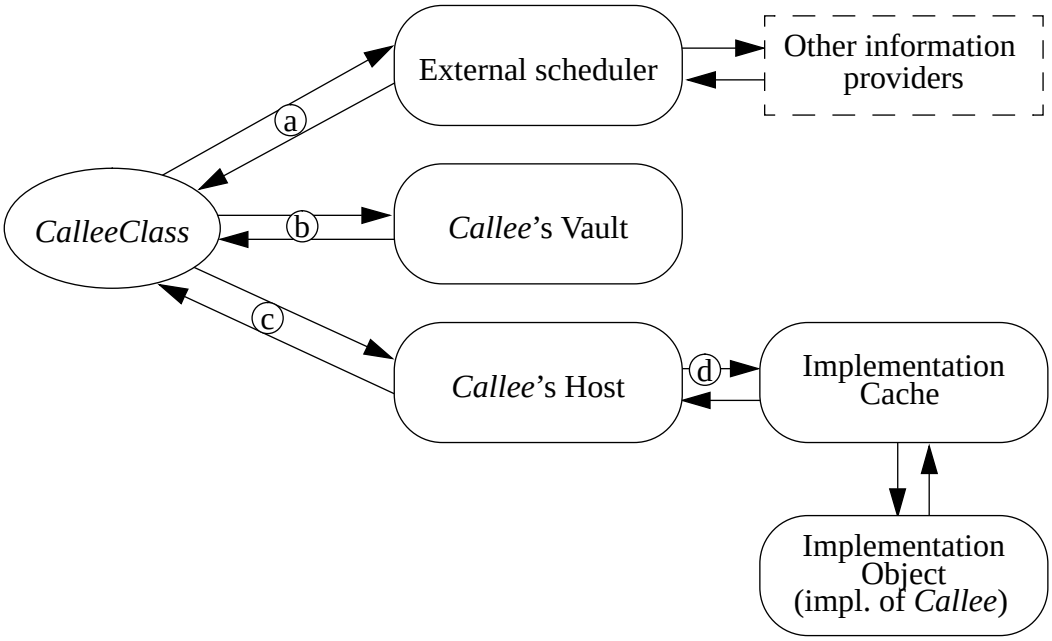

FIGURE 7. The Legion object activation mechanism—CalleeClass wishes to activate Callee. First, CalleeClass must decide on which host and vault to place Callee. To do this, CalleeClass may consult an external scheduler (a). After a placement decision is made, CalleeClass must determine the OPA for Callee by consulting Callee's vault (b). Finally, to activate Callee, CalleeClass sends an activation request to the desired host (c) specifying the object LOID, implementation to use, and OPA. To created a process for Callee, the host must obtain the implementation for Callee. To do this, the host uses a shared implementation cache object (d). After downloading the implementation, the host starts a process for Callee and returns the binding to CalleeClass.

Figure 7 depicts the object activation process. Before Callee can be activated, CalleeClass must select a host on which Callee can execute. CalleeClass has complete freedom in selecting an appropriate host for its instances. A very conservative and simple class object might place all of its instances on its own host. More typically, a class object will want to employ more elaborate and flexible placement policies. A common way to associate placement policies with classes is through external scheduling agents. This allows a simple, generic class object implementation to be combined with any number of separately defined scheduling policies. It also supports the dynamic replacement of a class object's scheduling policy.

When an external scheduling agent is used (as in Figure 7), the external scheduling agent may implement any specialized placement policy appropriate for the class. For example, specialized policies may be appropriate for a 2D finite difference class used in an ocean model,

or for a class designed to meet a particular organization's requirements that objects only execute on machines local to the organization and only use resources not currently supporting interactive sessions. Whatever the policy, the external scheduling agent will typically interact with other information providers (objects that gather and dynamically update information about which hosts are available, their type and attributes, their current load, and so on). Note, though, that the placement process is guided by a set of restrictions determined by the class. A class might maintain a list of acceptable object implementations (possibly for different architectures). If the class has no implementation available for Sun hosts running the Solaris operating system, hosts of this type must be excluded from selection. For more details on the scheduling model see Karpovich [24]. For more information on application specific scheduling agents see Berman [5].

The placement process is performed either by the class object itself, or by an external agent, and ultimately produces the LOID of the host object selected for placement. Next, the instance must be activated on that host. However, the class must first ensure that the instance will be able to access its OPR when it runs on that host. As mentioned in Section 4.1.5, vault object manage OPRs, which reside on physical storage devices. Not all storage devices are accessible from all hosts, so before a class activates an instance on a given host, it must verify that the current vault object containing the instance’s OPR is compatible with that host. Stated differently, the class must ensure that the host it selects to execute its instance has access to the storage device that contains the instance’s persistent state. Attributes (Section 4.2) of the host and vault objects are used to indicate compatibility. If a class wishes to execute an instance on a host that is not compatible with the vault currently containing the instance’s OPR, the OPR must be migrated to a new compatible vault before the object is started. Once the appropriate host has been selected, and the object’s OPR resides in a compatible vault, the class object invokes the

host object's **`startObject()`** method. Parameters to this method specify the LOID and OPA of the object to be started, and the LOID of the *implementation object* (see Section 5.5.1 and Section 6.4) to be used.

The **`startObject()`** method may not succeed, however, for a variety of reasons. The host object is free to refuse the activation request for policy or security reasons; perhaps, for example, only privileged users can use that particular resource. Or, a host might decide that its load is too high to accept new object activations. In some cases, a host may have simply crashed and will (at least temporarily) be unable to service the request. If the **`startObject()`** invocation fails, the class object must make another placement selection, possibly re-invoking the external scheduling agent.

5.5.1 Implementation Objects

Assuming, though, that the host decides to accept the instantiation request, it must start a process to represent the object. This means that the host must obtain appropriate executable code. Typically, implementation objects contain a binary executable file, although the model explicitly allows shell scripts and interpreted code such as Java bytecode or Perl. Management of object executables is based on the use of implementation objects, Legion objects that contain the executable code for other objects. Each class maintains a list of LOIDs of implementation objects that are suitable for the class's instances. Several different implementation objects might be maintained by a class to support the use of multiple platforms—a class might have implementation objects for different architectures, for different operating systems, with different memory requirements, etc. So, when **`startObject()`** is invoked on a host, the class passes the LOID of the appropriate implementation object, from which the host can retrieve appropriate executable code.

5.5.2 Implementation Caches

To service a **`startObject()`** request, the host object must find or make a local copy of the executable code contained within the specified implementation object. A simple host object could retrieve, via the member functions of implementation objects, the executable code on every **`startObject()`** invocation. However, retrieving executable code can be expensive, in both communication time and local storage space. Thus, groups of host objects typically share an external *implementation cache*, a Legion object that downloads executable code on behalf of a set of host objects and caches copies of the executables to save storage space and communication time. To use an implementation cache, the host object sends the cache object the LOID of a desired implementation object. The cache object responds with the name of a local file that contains the cached executable code—the host need not be aware of whether the cache retrieved the executable in response to this request, or used an existing local copy.

5.5.3 Running Objects

Once the implementation is locally available, the host object can execute it. How an executable is used depends on the implementation's type and host object's characteristics. For example, if the implementation consists of native executable code, the host runs the executable as a normal process; if the host represents a normal Unix workstation it uses the **`fork()`** and **`exec()`** system calls. Alternatively, if the implementation consists of Java bytecode, the host executes it within a Java Virtual Machine. In yet another case, if the host represents a workstation farm that is managed by a queueing system such as Condor [27] or LoadLeveler [23], the host starts the object through the batch system's particular interface.

Once the host activates the object, the host passes the object its LOID and its new OPA. The host object determines the activated object's local OA, and returns it to the calling class

object. The class marks the instance as active, records the instance's OA, and can once again return accurate bindings for the instance.

The binding and activation processes as described throughout this section can be time consuming. In practice, aggressive caching of bindings and object executables helps bypass much of the mechanism and its cost. The benefits of this design include flexibility and the convenient transparent binary migration, one-step system-wide binary replacement for objects, object-local policy autonomy, licensing and proxies, user-definable scheduling policies, user-definable persistent storage, and more. The following sections describe how users can realize these and other features by using and customizing core object implementations.

6. Core Object Types

6.1 Classes and Metaclasses

As described earlier, every Legion object is defined and managed by its class object. Class objects are empowered with system-level responsibility to create new instances, schedule them for execution, activate and deactivate them, and provide bindings for clients who wish to communicate with them. In this sense. Classes are *managers* and *policy makers*: Legion allows users to define and build their own class objects so that Legion programmers can determine and even change the system-level mechanisms that support their objects. Combining these two important features—class objects manage their instances and can be provided by applications programmers, not just system developers—provides considerable flexibility in determining how an application behaves, and further supports the Legion philosophy of enabling flexibility in the kind and level of functionality

Legion classes export the class-mandatory interface, a simplified subset of which is

depicted in Figure 8.

```
class ClassObject {
   LOID     createInstance(<placement info>);
   LOID     createMultipleInstances(<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 8. A subset of the Legion class-mandatory interface

The **`createInstance()`** function causes a new instance of the class to be created, and returns this new instance's LOID. **`CreateMultipleInstances()`** can be used to create several instances of the class at once. There are actually several different flavors of **`createInstance()`** and **`createMultipleInstances()`**, which allow the caller varying levels of control over the creation and placement processes. For example, the caller can specify the host object on which the new instance(s) should be created, or a list of acceptable hosts from which to choose, or even a list of characteristics of acceptable hosts (processor speeds, architectures, etc.). The same is true of the **`activateInstance()`** function. The general object placement model is that the class selects the host and vault objects when placing its instances, but includes the object placement parameters in the activation and creation functions so as to give callers a way to help the class make intelligent decisions, should the caller so choose. The **`deactivateInstance()`** function allows callers to make an active object inert, and **`deleteInstance()`** allows its caller to remove an instance from Legion.[5] The **`addImplementation()`** and **`removeImplementation()`** functions allow external

5. We should note here that an object can disallow any member function invocation requests, typically based on the identity of the caller. This is especially relevant to the system-level functions implemented in core objects, but it is true of all Legion objects.

objects (typically Legion-targeted compilers or Legion objects that manage the compilation process) to configure classes with implementation objects. The **`getBinding()`** functions support the binding mechanism as described in Section 5.2. Not shown in Figure 8 are several other functions that allow clients of class objects to retrieve information about the location and characteristics of a particular class's instances, including the instances' interface, host (if any) on which they're currently running, current state (active or inert), etc.

Although the core interface to classes is set, the implementation behind that interface can vary depending on the behavior that the class wishes to exhibit. This enables considerable flexibility of policy and mechanism. For example, a class can match its scheduling and object placement policies to the characteristics of the instances it supports. If an implementation runs much faster on a particular architecture, the class can factor in that affinity when selecting a host on which to run. If instances of a class communicate frequently with one another when they are created in relatively rapid succession (thereby possibly indicating that they are all part of the same instantiation of an application), the class can attempt to schedule these objects "close" to one another, (i.e. on hosts between which the communications links are fast, or possibly even as multiple threads within the same process).

Class objects are in the best position to be able to take advantage of their instances' special characteristics since class objects can be provided or selected by the same programmer who provides the implementation of that class object's instances. This does not mean, however, that all programmers must build a new specialized class object for each type of Legion object that they build, thereby incurring the burden of metasystem-level programming. On the contrary, we expect that a vast majority of programmers will be served adequately by the class object types that already exist in Legion.

The mechanism for taking advantage of an existing class object type is simple. Legion contains the notion of *metaclass objects*, 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 functionality behind that interface. To use one, a programmer simply calls **`createInstance()`** on the appropriate metaclass object, and configures the resulting class object, via **`addImplementation()`**, with implementation objects for the application in question. The new class object can then support the creation, migration, activation, and location of these application objects in the manner defined by its metaclass object.

One example of a class object taking advantage of knowledge about its instances' implementations is a *stateless* Mentat class. In MPL, the keyword **`stateless`** can be used to describe a class definition, as depicted in Figure 9.

```
stateless mentat class Example {
   public:
      int functionOne(int i);
      int functionTwo();
};
```

FIGURE 9. Stateless Mentat class definition

Here, the programmer has indicated that the instances of class **`Example`** do not maintain state between their member function invocations—that the instances are pure functional units. Therefore, from the client's point of view, invoking a function on one instance of a stateless class is functionally equivalent to invoking the same function on any other instance of that class.[6] In particular, two consecutive invocations from the same client need not be made on the

6. Of course, a function may return a different result if, for example, it queries its environment in some way for information. Mentat considers any environment information that may be different across different stateless objects to be state; in other words, if an object does this, it should not be declared as stateless.

same object.

The class object that supports stateless objects can take advantage of this fact when responding to class-mandatory member function invocations. For example, in response to a **`createInstance()`** call, the class object need not actually create a new instance; instead, it can instead simply return the binding for an instance that already exists. Conversely, if the load on some instance rises above some acceptable threshold, the class can create a new instance and respond to requests to bind to the heavily-loaded instance with a binding for the new instance. The point is that the class object can use its knowledge about the semantics of its instances to attempt to optimize its support for them.

6.2 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 aggregation of multiple hosts. A host object is a host's representative to Legion: it is responsible for executing objects on the host, reaping objects, and reporting object exceptions. A host object is also ultimately responsible for deciding which objects can run on the host it represents. Thus, host objects are important points of security policy encapsulation within the system.

Aside from implementing the host-mandatory interface depicted in Figure 10, host object implementations can be programmed to adapt to different environments and suit different users' needs. For example, host objects that provide interfaces to different resources must be based on the underlying resource management interface for those machines. A host object implementation suitable for use on a normal interactive workstation will use different process creation mechanisms than a host object that operates on a parallel computer whose nodes are managed by

a batch queuing system (e.g. LoadLeveler [23]) will use.

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

FIGURE 10. The Legion host object interface.

Whereas host object implementations provide a uniform interface to different resource management interfaces, they also (and more importantly) provide a means for users to enforce security and resource management policies for Legion objects. For example, the host object implementation can be customized to allow only a restricted set of users to have access to a resource. User authentication can be performed using any means desired. Alternatively, host objects can restrict access based on code characteristics. For example, a host might be configured to accept only object implementations that contain proof-carrying code [31] demonstrating certain desired security properties. A less formally restrictive host might analyze incoming object implementations for certain “restricted” system calls.

We now consider a sample host object implementation (our current default host object), and two possible alternative implementations. Our current host object has a very simple design—it implements a non-restrictive access policy and uses the Unix process management interface (i.e. **`fork()`**, **`exec()`**, **`kill()`**) for starting and stopping objects. While simple to implement, this basic host object design has a number of limiting features. In terms of performance, it places a high cost on object activation, since each object on the host executes within its own process, and new processes are created on demand to contain activating objects. In terms of security, this host object implementation is severely limited, since it executes objects owned by different Legion users under the same Unix user-id (processes executing as the same user-id can send one

another arbitrary signals, examine one another's state, and so on). Fortunately, we can address these limitations transparently by providing alternative host object implementations.

One possible implementation to address these performance problems might use threads instead of traditional processes. This design would improve the performance of object activation, and would also reduce the cost of method invocation between objects on the same host by allowing simplified shared address space communication. To support this style of host object, alternate forms of object implementations would need to be made available, particularly, object implementations in the form of dynamically loadable object files (as opposed to normal executable files). This would allow 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 implementation would appeal to users with high performance requirements, but it shares and exacerbates our existing host object's security limitations. An alternate host object implementation to support better security properties might be based on the use of multiple Unix user-ids to run different users' objects. Our current host object typically runs under a single user-id, and all objects that it starts also run as this user. If we extend the host object implementation to have the ability to start up processes under a set of different user-ids, it could ensure that different Legion users' objects run under different Unix user ids.

This host object implementation can be supported in a number of ways. For example, the host object could be given the limited amount of privilege needed to start processes as different user id's. This could take the form of a “set uid” script without write permissions for the host object so that it would not require full root permissions. Alternate approaches are also possible,

such as the use of a set of “sub-host objects”—one running as each user id—that could be used by the “primary” host object for control of low-level processes. In this design, the standard Legion authentication mechanisms could be used to ensure that only the host object is able to use these process-control daemons.

Different versions of this multi-user id host object can map different Legion users to different “anonymous” local user ids (e.g. “legion1,” “legion2,” etc.), or map Legion users to their associated local Unix user ids. The latter form extends directly to a simple scheme for limiting resource use to approved users—if a Legion user attempting to activate an object does not have a local Unix user-id, the activation request could be denied.

6.3 Vault Objects

Vault objects are responsible for managing other Legion objects’ 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. It might manage a portion of a Unix file system or a set of databases. The vault supports the creation of OPRs for new objects, controls access to the OPRs of existing objects that it manages, and supports the migration of OPRs from one storage device to another. The basic vault interface is depicted in Figure 11.

```
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 11. The Legion vault object interface.

Class objects manage the assignment of vaults to individual objects: 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 or reactivated, the selection of a new vault for the object is managed by the object's class.

If an object's class (or an external scheduling agent acting on behalf of that class) decides to move the object to another host, the migration may require moving the OPR to a new vault, whose persistent storage is accessible by objects on the new host. In this case, the class (or scheduling agent) selects a new vault, and the OPR is transferred between the vaults.

The above vault activities are supported by the basic Legion vault abstract-interface depicted in Figure 11. To enable object creation, the vault provides a **`createOPR()`** method, which constructs a new empty OPR, associates this OPR with the given LOID, and returns the address of the new OPR for use by the newly created object. To support object activation and deactivation, the vault provides a **`getOPRAddress()`** method to determine the location of the OPR associated with any of its managed objects. For use during object migration, vaults support **`giveOPR()`** and **`getOPR()`** methods, which transfer a linearized (i.e. transmissible) OPR to and from vaults, respectively. The **`deleteOPR()`** can be used to terminate a given vault's management of an OPR. The **`isManaged()`** method can be used to determine if a vault manages a given object. Finally, **`markActive()`** and **`markInactive()`** methods are provided so that the vault can be notified when an object is active or inactive, respectively. This knowledge allows the vault to store the OPRs of inactive objects in compressed or encrypted forms for efficiency and security purposes.

An important feature of the vault interface is its use of OPAs to provide access to object persistence representations. When an object wants to access its OPR, it can learn the OPA from its vault. The vault must provide an address that contains enough information embedded in it to

find and access the OPR. For example, consider an implementation of vaults and OPRs that is based on the Unix file system. In such an environment, an OPR might be implemented as a Unix directory, and an OPA might contain a Unix path name corresponding to a Unix directory. In this case the OPAs, besides containing the path name needed to locate the OPR, would also need to contain a type indicator that lets the object know that it should access the OPR in the form of a Unix subdirectory. In a sense, the OPA constitutes an agreement between a vault and a managed object about what kind of OPR will be used for the object. With this agreement, the object can directly access its OPR without consulting its vault. Clearly, not all object types or implementations need be compatible with all vaults. Just as class objects restrict the placement of objects and use of implementations to acceptable host objects, they also ensure reasonable placement of objects onto vaults.

The current Legion implementation supports two types of vaults (and hence, two types of OPR implementations): one for use in Unix file systems, and one for use with the SRB archival storage interface. These implementations are quite similar, as both systems support a file and directory interface typical of file systems. The addition of vaults for other file systems (e.g. Windows NT) and other archival file storage systems (e.g. HPSS) is straightforward. Alternative vault implementations can be built on top of database management systems. In this design vaults must manage the association between OPRs and database entries, and the mapping must be encapsulated in a suitable OPA format that can be used by managed objects to bind to their OPRs.

6.4 Implementation Objects and Caches

Implementation objects in Legion hide the storage details of object implementations. These objects can be thought of as the Legion equivalent of executable files in Unix or other traditional

operating systems. Given this similarity to files, implementation objects support an interface typical of file objects, as depicted in Figure 12. Read and write operations that assume a client-side file pointer, and a method to determine the size of the implementation data are provided.

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

FIGURE 12. The Legion implementation object interface

However, a fundamental difference between file objects and implementation objects is that implementation objects cannot be written to after being read from. After being initialized with a sequence of **`write()`** methods an implementation object's contents are constant until the object is deleted. This allows important caching optimizations to be employed by implementation object clients (i.e. host objects).

Legion object implementations typically contain executable object code for a single architecture and operating system platform, but may in general contain any information that would be necessary to instantiate an object on an appropriate type of host object. For example, the implementation might contain Java byte code, a Perl script, or even high-level source code that would require compilation by a host object upon object activation. A complete list of (possibly very different) acceptable implementation objects appropriate for use with a given class is maintained by the class object. When the class calls on a host to perform object activation, it selects an implementation object based on the attributes (see Section 4.2) of the host and the instance in question.

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 to run more quickly on hosts with abundant memory, and more slowly on hosts with less memory. To provide users with ability to customize their cost and performance trade-offs, a class might maintain slower, low-cost implementations for use with some instances, and faster, higher-cost implementations for use with other instances created by users willing to pay more.

In our discussion of object activation in Section 5.5, we described how host objects typically employ an external implementation caching object to avoid storage and communication costs. Since the contents of implementation objects do not change, a hosts can safely cache the downloaded contents of an implementation object for later use, saving a potentially significant amount of communication costs. Furthermore, if multiple host objects share access to some common storage device they can share the downloaded contents of implementation objects—that is, if one host downloads an implementation data to shared storage other hosts do not have to download that implementation themselves. Both of these performance enhancements are supported in the current Legion implementation through the use of implementation cache objects.

The interface to the implementation cache object is depicted in Figure 13—a single method is provided to return the path of a local file containing the same data as contained in a named implementation object. In our current Legion implementation, each host object is

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

FIGURE 13. The Legion implementation cache interface

associated with an implementation cache, and implementation caches can be shared among any number of hosts. Instead of performing implementation downloads, host objects invoke the

`getImplementation()` method on their local implementation cache object, which in turn downloads requested implementation data, caching the results of common requests. Thus the use of implementation caches results in object activation being approximately as inexpensive as running a program located in a file system visible on the host.

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, to begin using a new version of an implementation, a class need only replace the old implementation LOID with the new implementation LOID on its list of valid binaries. The new version will be specified with future activation requests, and the old implementation will simply time-out and be discarded from caches. Since the implementation is keyed on its LOID, there is no danger of "invalid" cached binaries being used to execute objects.

6.5 Binding Agents

Section 5 introduced the binding and class-of mechanisms and the role of binding agents, which exist in Legion to help client objects map LOIDs to OAs and to find the class of an object, given that object's LOID. The core interface to a binding agent is depicted in Figure 14.

```
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 14. The Legion binding agent interface

The **`getBinding(LOID)`** function returns a binding for a specified LOID, and **`getClassBinding(LOID)`** returns a binding for the class of a specified 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)`** support the rebinding mechanism (Section 5.4), allowing a client to pass a stale binding, and to suggest that the binding agent return a different binding in response. 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. In this way, a class object can help a binding agent better reduce the number of binding requests it makes to the class object.

Binding agents are not technically necessary for the correct execution of the binding mechanism; clients can directly contact class objects and LegionClass's class map to obtain bindings for objects and classes with which they wish to communicate. However, in a system that consists of millions of potentially migratory objects—as we envision Legion becoming—binding is a necessary and common operation. Binding agents exist to help make the binding mechanism *scalable*. For instance, in the example of Section 5.2, the binding agent runs a simple algorithm in response to the **`getBinding(LOID)`** call—it checks its local cache and (if necessary) it contacts the appropriate class object to obtain the binding. Even this simple strategy allows clients to benefit from the execution of the binding mechanism by other clients that share the same binding agent, thereby reducing the total amount of binding traffic in the Legion system.

A binding agent can implement several different strategies to improve its ability to provide bindings for its clients. For example, a binding agent might attempt to ensure that the bindings in its cache don't become stale, by either periodically “pinging”[7] the objects named in

its binding cache or contacting their classes to make sure they're still located at the same OA. Binding agents may also choose to get up-to-date values for bindings whose time-out field indicates that they are about to expire.

In addition to these strategies in which binding agents act essentially autonomously, many binding agents can be configured to cooperate with one another to serve their clients. For instance, binding agents could be organized hierarchically, as DNS name servers are, or could emulate a software combining tree [38], thereby sharing the responsibility for providing bindings and improving the mechanism for scaling to the millions of objects the system will need to support.

The above strategies attempt to improve the response time of binding requests, but they default to the full binding mechanism of contacting the appropriate class object if a binding cannot be otherwise obtained. Other binding agents may slightly change the semantics of the binding agent member functions in an attempt to optimize on different performance metrics. For instance, they may try to decrease the variance in the response times to binding requests. This could be accomplished by responding to a **`getBinding()`** call with a simple check the binding cache, avoiding the outcall to classes, even if the binding is not in the cache. To better respond to future requests, the binding agent could contact the class during the binding agent's idle time to get the binding, rather than while the client waits. Unlike the example of Section 5.2, a client of this kind of binding agent should not assume that a binding does not exist just because the binding agent doesn't return it; the client may be able to expect more timely responses to its requests.

---

7. **`ping()`** is an object-mandatory member function that returns the LOID of the called object.

6.6 Context Objects and Context Spaces

As described in Section 4.1, Legion objects are identified LOIDs. A LOID contains a set of fields including those that identify the class of the named object, a class-unique instance number for the named object, and a public key for the named object. 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. An example of a typical LOID is depicted in Figure 15.

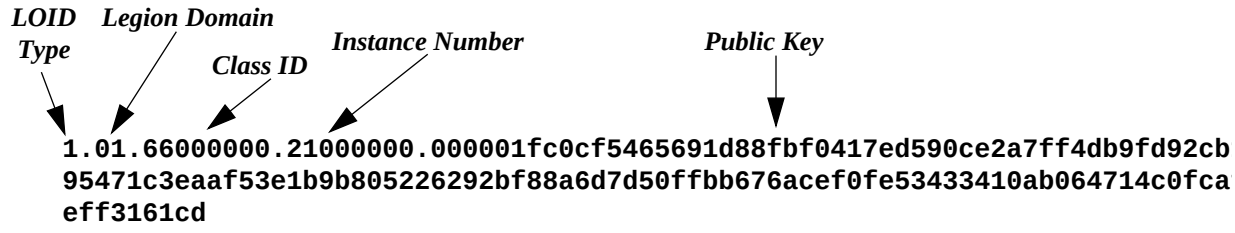

FIGURE 15. Example of a LOID

The LOID naming scheme is central to a number of Legion design features, but as Figure 15 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 in Legion, we define the interface to a user-level naming service called *context spaces*.

Context spaces consist of 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, as depicted in Figure 16. Operations are provided to insert a <name,LOID> tuple, to remove a string name, and to find the LOID associated with a given user-level string name. Also, a method is provided to return a list of <name,LOID> pairs, elements of which match a specified regular expression. At most one tuple containing any string

name may be contained within a context, although any number of different string names may map to the same LOID.

```
class Context {
   int      insert(String name, LOID loid);
   int      remove(String name);
   LOID     lookup(String name);
   List<String,LOID> multilookup(String regexp);
};
```

FIGURE 16. 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*,[8] and library routines are provided for traversing context space to map context paths to LOIDs.

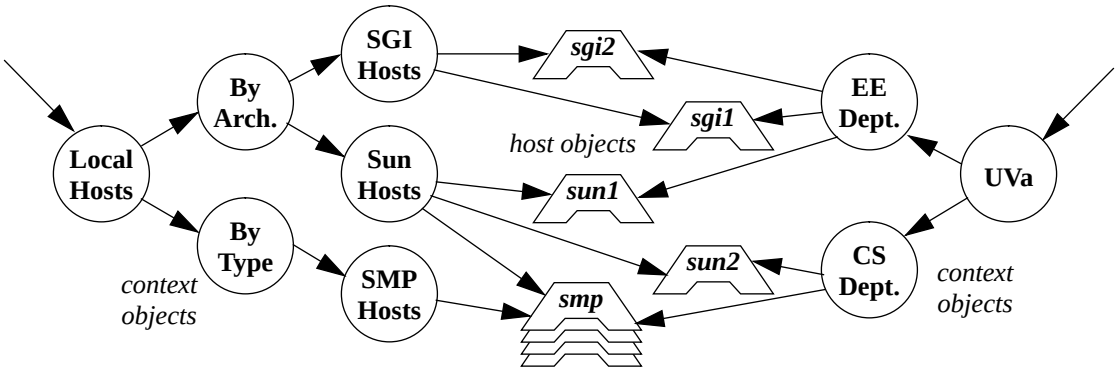

FIGURE 17. Context space used to organize information about host objects

On the surface, context space appears to provide a basic directory service. However,

8. 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.

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. For example, Figure 17 depicts a simple portion of a context space used to organize information about host objects. One view of context space provides hints about how to find hosts of given types, whereas a different view provides information about resource ownership. Similar organizational schemes are equally useful for objects of other types.

7. Project Status

In June 1996, after a year of design work, we began code development for Legion, and in December of 1997 we released VaL 1.0 (VaL stands for Virginia Legion—to denote that others may release different versions of Legion). VaL 1.0 is a complete implementation; it includes an implementation of 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 range of platforms (see Table 1).

Table 1: VaL 1.0 Supported Platforms

Platform	Operating System	Comments
x86	Linux	Linux is our development platform.
Sparc	Solaris	
RS/6000	AIX	Includes the SP-2. Does not currently use the SP-2 native message passing.
SGI	IRIX	Both workstations as well as the PCA and Origin 2000.
Alpha	Linux/DEC Unix	
Cray T90		IEEE FP.

From the user's point of view, the implementation consists of four parts: a set of "client" utilities, the core objects described earlier (hosts, vaults, etc.) with a set of default scheduling and placement policies, a set of tools to develop new Legion applications, and the Legion run-time library. Below we briefly expand on the utilities and application development tools. The core system has already been described in detail, and the run-time library is described completely in [10].

7.1 Client utilities

Examples of "client" utilities include

- Utilities such as **`legion_ls`**, **`legion_cat`**, **`legion_cp`**, etc., that manipulate context space and are similar to the Unix utilities of the same suffix.
- Utilities that manipulate classes to activate and deactivate instances, destroy instances, move instances, and set the acceptable hosts and vaults
- utilities to manipulate hosts and vaults, listing active objects on the host or in the vault, destroying running instances, or querying the host or vault about its status, such as the type of host, the amount of memory or disk space, the load, etc.
- security utilities to create new users, manipulate access control lists, authenticate a user to a login object, etc.

The list is quite extensive and is beyond the scope of this paper. It suffices to say that there are dozens of utilities. A full description can be found in the Legion users manual [25].

7.2 Program development tools

Without programming tools Legion would be of little use beyond a shared file space. We have concentrated on tools for parallel programming and I/O support. Our strategy focuses on the popular parallel computing tools PVM [14] and MPI [21, 29]. MPI is the most popular of the

two—and most parallel applications today are written in MPI. Legion provides a complete emulation of both PVM and MPI with user libraries for C, C++, and Fortran. Applications and benchmarks, such as the NAS benchmark, have been ported to Legion. Besides PVM and MPI, the Mentat Programming Language (MPL) [16], Basic Fortran Support (BFS), and Java are supported in Legion.

MPL, as noted earlier, 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 run-time system take over and construct parallel computation graphs of the program, then execute the methods in parallel on different processors. Legion is written in MPL. BFS is a set of pseudo-comments for Fortran and a pre-processor that gives the Fortran programmer access to Legion objects. It also allows parallel execution via remote asynchronous procedure calls and the construction of program graphs.

I/O support in applications programs is provided via a set of library functions with a unix-like file and stream operations to read, write, and seek. These functions provide complete, location independent, secure, access to context space and to “files” in the system.

8. 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 [13] and Globe [35]. However, it is worth noting that these projects, Legion, and other metacomputing projects such as MOL [33], Ice-T [15], and Harness [9], are all outgrowths of the significant existing work in first-generation network parallel computing systems, such as PVM [14] and MPI [21], and in modern transparent distributed computing systems, such as the Berkeley NOW project [1] and DCE [28].

8.1 Globus

The Globus project [13] at Argonne National Laboratory and the University of Southern California shares with Legion a common base of target environments, technical objectives, and target end users. Beyond a basic similarity in goals, Globus and Legion also share 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) [12]. Legion abstracts access to processing resources via the host object interface, and Globus abstracts access to processing resources through the Globus Resource Allocation Manager (GRAM) interface [8]. Both systems support a range of programming interfaces, including popular packages such as MPI.

Despite an underlying commonality of goals and basic approaches, 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). Globus defines a set of metacomputing-related data structures that are contained within the LDAP (e.g. information about hosts, users, networks), and allows LDAP to run over the Globus message passing implementation.

The Globus approach of adding value to existing high-performance computing services by rendering them interoperable and extending their implementations to operate well in a wide-area distributed environment has a number of advantages. For example, this approach takes great advantage of code reuse, with its many attendant advantages, and allows the user to retain

familiar tools and work environments. However, this sum-of-services approach has a number of drawbacks: as the amount of provided 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. The short-term advantages of patching existing parallel and distributed computing services together to render them interoperable and usable in a wide-area environment do not outweigh the long-term *necessity* of basing a metacomputing software system on an extensible design consisting of orthogonal building blocks. The challenges of metacomputing are great; finding scalable, efficient, and robust solutions demands fundamental architectural design that can not be achieved within the framework of most existing parallel and distributed systems.

8.2 Globe

The Globe [35] 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 inter-object 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.

8.3 CORBA

The Common Object Request Broker Architecture (CORBA) standard developed by the Object Management Group (OMG) [32] 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 the large degree of similarity in basic concepts between CORBA and Legion, 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 their basic object models 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.

9. 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; it requires that the metasystem software be extensible to meet unanticipated needs, and that it 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.

References

[1] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team, "A Case for NOW (networks of Workstations)" *IEEE Micro*, vol. 15, no. 1, pp. 54-64, February, 1995.

[2] Bal, H.E., Steiner, J.G., and Tanenbaum, A.S., "Programming languages for distributed computing systems," *ACM Computing Surveys*, vol. 21, no. 3, September 1989.

[3] Betz, M., "Interoperable objects," *Dr. Dobb's Journal*, pp. 18-39, October 1994.

[4] Brockschmidt, K., "What OLE is really about," Microsoft Corporation, July 1996.

[5] Berman, F., Wolski, R., Figueira, S. Schopf, J., and Shao, G. “Application-Level Scheduling on Distributed Heterogeneous Networks”, *Proceedings of Supercomputing '96*, November 1996.

[6] Cardelli, L. “A language with distributed scope,” Digital Equipment Corporation, May 1995.

[7] Chin, R.S. and Chanson, S.T., “Distributed object-based programming systems,” *ACM Computing Surveys*, vol. 23, no. 1, pp. 91-124, March 1991.

[8] Czajkowski, K., Foster, I., Kesselman, C., Martin, S., Smith, W., and Tuecke, S., “A Resource Management Architecture for Metacomputing Systems,” available at: **`http://www.globus.org/globus/papers.htm`**

[9] Dongarra, J., Geist, A., Kohl, J., Papadopoulos, P., Sunderam, V., “HARNESS: Heterogeneous Adaptable Reconfigurable Networked Systems,” available at: **`http://www.epm.ornl.gov/harness/`**

[10] Ferrari, A.J., Lewis, M.J., Viles, C.L., Nguyen-Tuong, A., and Grimshaw, A.S., “Implementation of the Legion library,” University of Virginia Computer Science Technical Report CS-96-16, November 1996.

[11] Ferrari, A.J. and Grimshaw, A.S. “Basic Fortran Support In Legion”, University of Virginia Department of Computer Science, Technical Report CS-98-11, March 4, 1998.

[12] Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W., and Tuecke, S., “A Directory Service for Configuring High-Performance Distributed Computations,” *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, 1997.

[13] Foster, I. and Kesselman, C., “Globus: A metacomputing infrastructure toolkit,” *International Journal of Supercomputer Applications* (to appear).

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

[15] Gray, P., and Sunderam, V.S. “IceT: Distributed Computing and Java,” *Concurrency, Practice and Experience*, vol. 9, no. 11, pp. 1161-1168, Nov. 1997.

[16] Grimshaw, A.S., “Easy-to-use object-oriented parallel processing with Mentat,” *IEEE Computer*, pp. 39-51, May 1993.

[17] Grimshaw, A.S., Weissman, J.B., and Strayer, W.T. “Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,” to appear *ACM Transactions on Computer Systems*.

[18] Grimshaw, A.S., Weissman, J.B., West, E.A., and Loyot, E., “Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems,” *Journal of Parallel and Distributed Computing*, vol. 21, no. 3, pp. 257-69, June 1994.

[19]Grimshaw, A.S., Wulf, W.A., the Legion team, “The Legion vision of a worldwide virtual computer,” *Communications of the ACM*, vol. 40, no. 1, January 1997.

[20]Grimshaw, A.S. and Wulf, W.A., “Legion—a view from 50,000 feet,” *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.

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

[22]Homburg, P., van Doorn, L., van Seen, M., Tanenbaum, A.S., and de Jonge, W., “An object model for flexible distributed systems,” Vrije Universiteit.

[23]IBM, “IBM LoadLeveler: User’s Guide (SH26-7226-02),” IBM Publication number ST00-9696, October 1994.

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

[25]The Legion Research Group, “Legion 1.0 User Manual,” University of Virginia Computer Science, November 1997. available from **`http://legion.virginia.edu/Documentation.html`**.

[26]Lewis, M., and Grimshaw, A.S., “The core Legion object model,” *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.

[27]Litzkow, M.J., Livny, M., and Mutka, M.W., “Condor—A Hunter of Idle Workstations,” *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 1988, pp. 104-111.

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

[29]Message Passing Interface Forum, “MPI: A message-passing interface standard,” May 1994.

[30]Microsoft Corporation, “The Component Object Model specification,” Version 0.9, Microsoft Corporation, October 24, 1995.

[31]Necula, G.C., “Proof-Carrying Code,” *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 106-119, Jan 15-17, 1997.

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

[33] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Rvmke, and J. Simon, “The MOL Project: An Open Extensible Metacomputer,” in *Proceedings of the Heterogenous Computing Workshop, HCW97*, IEEE Computer Society Press, pp. 17-31, 1997.

[34] SunSoft, SunOS 5.3 Linker and Libraries Manual, Sun Microsystems, Inc., Mountainview, California, 1993.

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

[36] Viles, C.L., Lewis, M.J., Ferrari, A.J., Nguyen-Tuong, A., and Grimshaw, A.S., “Enabling flexiblity in the Legion run-time library,” *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’97)*, pp. 265-274. Las Vegas, Nevada, June 30 — July 2, 1997.

[37] Wulf, W.A., Wang, C., and Kienzle, D., “A new model of security for distributed systems,” University of Virginia Computer Science Technical Report CS-95-34, August 1995.

[38] Yew, P.-C., Tzeng, N.-F., and Lawrie, D.H., “Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,” *IEEE Transactions on Computers*, Vol. C-36(4), April 1987.