The Core Legion Object Model

Mike Lewis and Andrew Grimshaw

{mlewis, grimshaw}@Virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The Legion project at the University of Virginia is an architecture for designing and building system services that provide the illusion of a single virtual machine to users, a virtual machine that provides secure shared object and shared name spaces, application adjustable fault-tolerance, improved response time, and greater throughput. Legion targets wide area assemblies of workstations, supercomputers, and parallel supercomputers. Legion tackles problems not solved by existing workstation based parallel processing tools; the system will enable fault-tolerance, wide area parallel processing, inter-operability, heterogeneity, a single global name space, protection, security, efficient scheduling, and comprehensive resource management.

This paper describes the core Legion object model, which specifies the composition and functionality of Legion's core objects—those objects that cooperate to create, locate, manage, and remove objects in the Legion system. The object model facilitates a flexible extensible implementation, provides a single global name space, grants site autonomy to participating organizations, and scales to millions of sites and trillions of objects.

1. Introduction

The next several years will see the widespread introduction and use of gigabit wide area and local area networks. These networks have the potential to transform the way people compute, and more importantly, the way they interact and collaborate with one another. The increase in bandwidth will enable the construction of wide area virtual computers, or metasystems. However, just connecting computers together is insufficient. Without easy-to-use and robust software to simplify the environment, the network will be too complex for most users.

The Legion project at the University of Virginia is an architecture for designing and building system services that provide the illusion of a single virtual machine to users, a virtual machine that 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; the system will enable fault-tolerance, wide area parallel processing, inter-operability, heterogeneity, a global name space, protection, security, efficient scheduling, and comprehensive resource management.

Legion will consist of workstations, vector supercomputers, and parallel supercomputers connected by local area and wide area networks. The total computation power of such an assembly of machines approaches a petaflop, but so far this enormous potential is unrealized. The machines are currently tied together in a loose confederation of shared communication resources used primarily to support electronic mail, file transfer, and remote login. However, the resources could be used to provide far more than just communication services; they have the potential to provide a single seamless computational environment in which processor cycles, communication channels, and data are all shared, and in which the workstation across the continent is no less a resource than the one down the hall.

- This work is partially funded by NSF grants ASC-9201822 and CDA-8922545-01, and ARPA grant J-FBI-93-116.
- Many people have contributed to the design of the Legion core. The other current members of the Legion team include professors William A. Wulf, James C. French, Paul F. Reynolds, Jr., and Alfred C. Weaver, research associate Charles Viles, research scientist Mark Hyett, and graduate students Adam Ferrari, John Karpovich, Darrell Kienzle, Anh Nguyen-Tuong, and Chenxi Wang.

We envision a system in which a user sits at a Legion workstation and has the illusion of 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 will be invisible.

2. Objectives

From our Legion vision we have distilled ten design objectives that are key to the success of the project—site autonomy, an extensible core, a scalable architecture, an easy-to-use and seamless computational environment, high performance via parallelism, a single global name space, security for both users and resource providers, management and exploitation of resource heterogeneity, multi-language support and inter-operability, and fault tolerance. These ten objectives are discussed at length in a companion paper in these proceedings [8]; four of the objectives are particularly relevent to the design of the core and are described below.

Site autonomy: Legion will not be a monolithic system. Legion will comprise diverse resources owned and controlled by an array of different organizations. These organizations, quite properly, insist on having control over their own resources by specifying how much of a resource can be used, when it can be used, and who may and may not use the resource.

Extensible core: We cannot know the future or all of the many and varied needs of users. Therefore, mechanism and policy must be realized via extensible, replaceable, components. This will permit Legion to evolve over time and will allow users to construct their own mechanisms and policies to meet specific needs.

Scalable architecture: Because Legion will consist of millions of hosts, it must use a scalable software architecture. This implies that there must be no centralized structures, and that the system must be totally distributed.

Single global name space: One of the most significant obstacles to wide area parallel processing is the lack of a single name space for data and resource access. The existing multitude of disjoint name spaces makes writing applications that span sites extremely difficult.

In addition to the ten objectives, three constraints influence our design—we cannot replace host operating systems, we cannot require privileged access to participating hosts, and we cannot legislate changes to the interconnection networks. Operating system replacement would require organizations to rewrite many of their applications and to retrain many of their users, possibly resulting in incompatibilities with other systems in the organization. Requiring privileged access to participating hosts would be asking for more trust than most organizations will be willing to give us, especially during the initial deployment stages of Legion. 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 state that performance for a particular application on a particular network will be poor unless the protocol is changed. Our experience with Mentat [8] indicates that it is sufficient to layer a system on top of an existing host operating system.

In addition to the purely technical issues, there are also political, sociological, and economic ones. These include encouraging the participation of resource-rich centers and discouraging the human tendency to free-ride. We intend to develop mechanisms that facilitate accounting policies that encourage good community behavior.

2.1. The core object model

This paper describes the core Legion object model. The model specifies the composition and functionality of Legion's core objects—those objects that cooperate to create, locate, manage, and remove objects from the Legion system. The model reflects the underlying philosophy and objectives of the Legion project. In particular, the object model facilitates a flexible extensible implementation, provides a single global name space, grants site autonomy to participating organizations, and scales to millions of sites and trillions of objects. Further, it offers a framework that is well suited to providing mechanisms for high performance, security, fault tolerance, and commerce.

Legion specifies the functionality, not the implementation, of its core objects. The Legion system designers cannot predict the many and varied needs of users. Therefore, the object core will consist of extensible, replaceable components. The Legion project will provide implementations of the objects that comprise the core, but users will not be obligated to use them. Instead, Legion users will be encouraged to select or construct objects that implement mechanisms and policies that meet the users' own specific requirements.

To facilitate the development of applications that span multiple sites, a single global name space will unite the objects in the Legion system; this will make remote files and data more accessible. Site autonomy will be provided by distributing control of Legion resources among an extensible set of core user-level Legion objects. Controlling a resource includes making decisions about which Legion objects can access it, and to what extent. Placing this responsibility in the hands of objects that users can build themselves gives sites the autonomy that they properly require.

The Legion system will be fully scalable. Although the object model includes, and relies on, a few single logical Legion objects, access to these objects will be limited due to heavy caching and hierarchical organization of lower level objects. Legion objects can be replicated to further reduce contention. Thus, the system will be configured such that an increase in the number of Legion computing resources will not impact contention for the few "centralized" Legion objects.

The initial design phase of Legion has been completed and is presented in this paper, which describes the core Legion object model, characterizes its main components, describes the mechanism it is intended to support, and addresses the issue of scalability. The model is still evolving and includes several aspects that have yet to be addressed in detail, or that are addressed in other documents [18][11].

Implementation of the core model has just begun and is expected to take approximately one year. In the interim, application and tool developers at the University of Virginia use our existing prototype system, the Campus Wide Virtual Computer (CWVC) [10]. The CWVC is based on the Mentat object-oriented parallel processing system [8]; it contains over one hundred hosts of five different architecture types, spanning several different file systems and university departments. Applications developed using the CWVC will be source code compatible with the eventual fully developed Legion system.

3. Related work

The vision of a seamless metacomputer such as Legion is not novel; worldwide computers have been the vision of science fiction authors and distributed systems researchers for decades. However, to our knowledge no other project has the same broad scope and ambitious goals of Legion. Fortunately, it is not necessary to develop all of the required technology from scratch. A large body of relevant research in distributed systems, parallel computing, management of workstation farms, and pioneering wide area parallel processing projects, will provide a strong foundation on which to build.

OSF/DCE [13] is rapidly becoming an industry standard. Legion and DCE share many of the same objectives, and draw upon the same heterogeneous distributed computing literature for inspiration. Consequently, both projects use many of the same techniques, including an object-based architecture and model, IDL's to describe object behavior, and wrappers to support legacy code. However, Legion and DCE differ in several fundamental ways. First, DCE does not target high-performance computing; its underlying computation model is based on blocking RPC between objects. Further, DCE does not support parallel computing; instead, the emphasis is on client-server based distributed computing. Legion, on the other hand, is based upon a parallel computing model, and one of our primary objectives is high performance via parallel computation. Another important difference is that Legion specifies very little about the implementation. Users and resources owners are permitted—even encouraged—to provide their own implementations of "system" services. Our core model is completely extensible and provides choice at every opportunity—from security to scheduling to fault-tolerance. Despite these differences, we recognize that DCE is here to stay for the foreseeable future. Our intention, therefore, is to support DCE-like services and DCE compatibility modules so that applications developed in a DCE environment can be easily ported to Legion.

Several other projects have also begun to address some of the same issues that Legion does. For example, Nexus [6] provides communication and resource management facilities for parallel language compilers. Castle [3] is a set of related projects that aims to support scientific applications, parallel languages and libraries, and low-level communications issues. The NOW [1] project provides a somewhat more unified strategy for managing networks of workstations, but is intended to scale only to hundreds of machines instead of millions. Globe [16] is an architecture for supporting wide area distributed systems, but does not yet seem to address important issues such as security and site autonomy. Finally, CORBA [4] defines an object-oriented model for accessing distributed objects. CORBA includes an Interface Description Language, and a specification for the functionality of run-time systems that enable access to objects. But like DCE, CORBA is based on a client-server model rather than a parallel computing model, and less emphasis is placed on issues such as object persistence, placement, and migration. Thus, while each of these projects—and others like them—addresses some, or even many, of the issues that are necessary for the realization of wide area parallel computing, none aspires to deal as comprehensively with all of the issues that Legion does.

4. The Legion core

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. 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 value, 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 can be described in an Interface Description Language (IDL). Initially, two different IDL's will be supported by Legion: the CORBA IDL [4], and the Mentat Programming Language (MPL) [14].

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, including may_I(), save_state(), and restore_state(). Class objects export an additional set of *class-mandatory* member functions, including create(), derive(), and inherit_from().

The power of the Legion object model 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 subclasses, 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 algorithms that they choose. Assuming that we define 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. Security

Legion is intended to be used by a wide variety of users with a correspondingly wide variety of security concerns. Thus, it is not possible to dictate a single standard policy by which all users must abide. Any compromise position would most likely either degrade performance to a degree unacceptable to many users, or be too insecure for an equally large number.

To appeal to the widest range of users, Legion must allow users to define whatever degree of security they deem necessary in a manner that is simple to implement and that does not penalize other users unnecessarily. One alternative is to have a number of existing approaches from which users can select. Another—the one we have chosen—is to provide users with the tools necessary to build robust security measures that provide the required degree of security. A number of approaches will be provided so that users can customize as much or as little of their security mechanism as they wish.

Legion security is divided into three components. The message layer is responsible for inter-object communication and authentication. The discretionary layer allows the user to provide a function, called may_I(), that acts as an access control predicate. Before any other object can invoke any method on an object it must gain approval from the may_I() function. This approach permits any discretionary policy to be defined, concentrates all discretionary security in one location, and frees the implementation of the object's functionality from being cluttered with security concerns. Finally, the mandatory layer enables Legion objects acting as Security Agents to monitor other Legion objects in order to enforce security policies that the objects themselves cannot be trusted to abide by. A Security Agent can restrict the forms of communications that objects under its scrutiny can undertake. It can also implement dynamic information flow policies by tracking the flow of secure information to other objects.

The Legion security model is elegant and powerful. Using its basic mechanisms, it will be possible to implement CORBA, DCE, Kerberos, MLS, NFS, and other existing systems. Further, the flexibility of the mechanism allows entirely new forms of security policies to be developed and enforced. The Legion security model is described in detail in [18].

4.2. Core class objects

Legion defines the interface and functionality of several core class objects, including *LegionObject*, *LegionClass*, *LegionHost*, *LegionVault*, and *LegionBindingAgent*. LegionObject provides the full set of object-mandatory member functions. All Legion objects are instances of classes that are eventually derived from the class LegionObject, and thus they inherit all of the member functions defined in LegionObject. LegionClass provides the full set of class-mandatory member functions. All Legion classes are eventually derived from LegionClass, and thus they inherit all of the member functions defined in LegionClass. LegionClass is derived from LegionObject; thus, classes are objects in Legion. Classes may alter the functionality of object- or class-mandatory member functions by overloading them, by redefining them, or by explicitly "re-inheriting" their implementation from class objects other than LegionObject and LegionClass.

LegionHost, LegionVault, and LegionBindingAgent are base classes for Legion's core class types—Hosts, Vaults, and Binding Agents. The core classes set the minimal interface that the core objects should export. Every core object is an instance of some class that is eventually derived from one of the class objects above. For example, as shown in Figure 1, UnixHost and SPMDHost will be two different Legion classes derived directly from class LegionHost. More specific host classes will be derived from each of these. A Sun workstation would run an instance of class UnixHost, whereas a Silicon Graphics Power Challenge would run an instance of UnixSMMP, a class derived from UnixHost. Similar class hierarchies will develop for Vaults and Binding Agents. The roles of the core objects are described later in the paper.

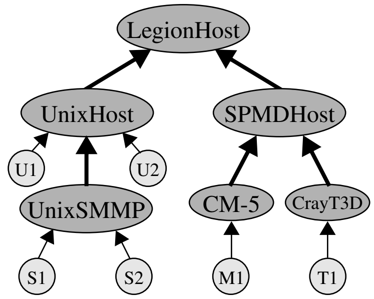

FIGURE 1. The Legion class LegionHost is the root of all classes whose instances are Legion Hosts. In this figure, UnixHost and SPMDHost are derived directly from LegionHost. UnixSMMP is derived from UnixHost, and CM-5 and CrayT3D are derived from SPMDHost. The figure shows six different Legion Hosts: two instances of both UnixHost and UnixSMMP, and one instance of both CM-5 and CrayT3D.

4.3. Naming and binding

LOID's: Every Legion object is named by a Legion Object Identifier (LOID). An LOID comprises four different fields—a Format, a Class Identifier, an Instance Number, and a Public Key extension. The Format field makes up the first 32 bits of the LOID, and although the other three fields are expected to be contained in all LOID's, their size, format and use can vary with different formats. The Class

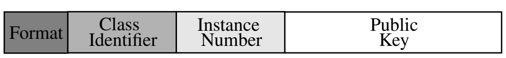

FIGURE 2. An LOID comprises a Format, a Class Identifier, an Instance Number, and a Public Key extension.

Identifier indicates the class of the object that the LOID names. LegionClass is ultimately responsible for handing out unique Class Identifiers to each new class. The Instance Number field can be used by classes to provide a unique LOID to each instance of the class, but this use is not mandated by Legion—a class object can assign Instance Numbers in any way it chooses. The Public Key extension field allows the entire LOID to be a public key for the object, and is used for security purposes [17][18]. Our initial implementation will support fixed size LOID's that contain 64-bit Class Identifiers, 64-bit Instance Numbers, and 128-bit Public Key extensions. Other LOID formats will emerge.

Legion will use standard protocols and the communication facilities of host operating systems to support communication between Legion objects. However, LOID's have meaning only at the Legion level. Consequently, Legion must provide a mechanism by which LOID's can be bound to names that have meaning to these underlying protocols and communication facilities. The general problem is that one object, A, has the LOID of another object, B, and A wishes to invoke member functions on B. A physical *Object Address* for B must be obtained before the communication can take place.

Object Addresses: An Object Address is a list of *Object Address Elements*, along with semantic information that describes how to utilize the list. An Object Address Element contains two basic parts—a 32-bit Address Type field, and the address itself. The Address Type field indicates the type of address (e.g. IP, XTP, etc.) that is contained in the address specific field, whose size and format will vary depending on the Address Type. The first (and probably the most common) type of Object Address will consist of a single Object Address Element that comprises a 32-bit IP address and a 16-bit port number.

The Address Semantic field is intended to encapsulate various forms of multicast communication. For example, the field could specify that all addresses should be sent to, 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.

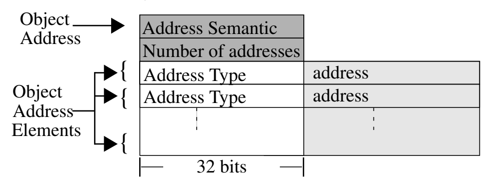

FIGURE 3. An Object Address comprises a list of physical addresses and semantic information that describes how the list is to be used.

Bindings: Bindings from LOID's to Object Addresses in Legion are implemented as simple triples. A binding consists of an LOID, an Object Address, and a field that specifies the time that the binding becomes invalid. This field may 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.

Binding Agents: Binding Agents are derived from the Abstract class LegionBindingAgent. A Binding Agent acts on behalf of other Legion objects to bind LOID's to Object Addresses. That is, given an LOID for an object, a Binding Agent is responsible for returning a binding to an Object Address for the object that the LOID names. The persistent state of each Legion object contains the Object Address of its Binding Agent.

Legion does not mandate how any particular Binding Agent performs its duty. Typically, however, a Binding Agent will maintain a cache of bindings that it will consult in response to binding requests from other objects; LegionBindingAgent's member functions reflect this fact. Any particular Binding Agent may also consult other Binding Agents, and may employ any other means to locate a binding for a given LOID. If all else fails, the Binding Agent can consult the class of the object, which must be able to return a binding if one exists. A more in-depth discussion of a typical binding procedure is included in Section 5.1. LegionBindingAgent has the following member functions:

- *binding get_binding(LOID),*
  binding get_binding(binding): The overloaded method get_binding() is passed an LOID or a binding, and returns a binding. Passing an LOID as the parameter requests that the Binding Agent bind it to an Object Address. Passing a binding requests that the Binding Agent return a different binding than the one passed as a parameter. For instance, if the Object Address in the binding parameter matches the one in the Binding Agent's local cache, the Binding Agent might contact the class object for an updated binding. Thus, the object employing the Binding Agent can explicitly request that a binding be refreshed; it will typically do so when the binding that it has doesn't work.
- *invalidate_binding(LOID),*
  invalidate_binding(binding): The overloaded method invalidate_binding() tells the Binding Agent to remove bindings from its cache. The first form requests that the Binding Agent remove an LOID's binding, if any exists, from its cache. The second form requests that it remove a binding if it matches exactly the binding that is passed as an argument.
- *add_binding(binding):* add_binding() is used to add a binding to the cache of bindings that the Binding Agent maintains. It can be used by Binding Agents, or any other Legion objects, to explicitly propagate binding information for performance purposes.

4.4. Object states

The full set of Legion hosts will be unable to simultaneously provide each Legion object with a process to implement the disjoint address space model. Therefore, a Legion object can be in one of two different states, *Active* or *Inert*. When an object is Active, it is running as a process (or set of processes) 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, it is described by an *Object Persistent Representation (OPR)*, and it can be located using an *Object Persistent Address (OPA)*. Throughout their lifetime, objects can be moved between Active and Inert states by other Legion objects.

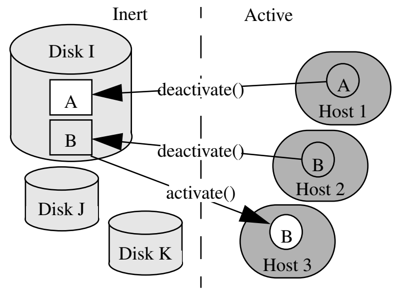

FIGURE 4. A sample subset of Legion comprising three disks (I, J, and K) and three hosts (1, 2, and 3). Objects A and B are moved between Active and Inert states. Object A has been deactivated into an Object Persistent Representation on Disk I, and B has been migrated from Host 2 to Host 3 through Disk I

Object Persistent Representations and Addresses: An OPR is associated with an Inert object and can be used to restore the state of the object. Every Legion object will export save_state() and restore_state() member functions. Save_state() will be called just before an object is deactivated, and restore_state() will be called as the first member function after the object is activated. Thus, objects are given the opportunity to carry their state information with them when they are migrated between hosts [5]. The OPA of an Inert object is analogous to the Object Address of an Active object. An OPA gives an object a handle on the OPR that the object reads and writes to save and restore its state information. Typically, an OPA will be a file name, and will only necessarily be meaningful to the Legion Vault that controls it, and to the object with which it is associated.

Legion Hosts: A Legion Host is a host's representative to Legion. It is responsible for executing objects on the host, reaping objects, and reporting object exceptions. Thus, the Legion Host is ultimately responsible for deciding which objects can run on the host it represents. Since Legion Hosts can be implemented by the users who offer their resources to Legion, and since our security model is one in which security is built into the object by its implementor, Legion users can select the policy and mechanism that restrict access to their own hosts. In a Unix-like implementation, all Legion objects that execute on a host will execute with the same privilege as the Legion Host. Therefore, Legion Hosts will typically execute with minimal privilege. Individual sites may choose to grant Legion Hosts a higher privilege if they desire.

A Legion Host is associated with a single logical machine, not necessarily a single physical machine. This allows resource providers to aggregate multiple machines into a single logical resource, which can be helpful in formulating a cohesive security or scheduling policy for the set of machines. It can also reduce the complexity of the environment for other objects in Legion.

Legion Hosts are started from outside Legion, for example from a command line or shell script in the host operating system. This is because Hosts are the mechanism by which objects are started; there is no Legion object to start Hosts. When new Legion Hosts are activated, they are responsible for registering themselves with the LegionHost class object. Hosts export member functions that start or restart processes, that suspend processes that are currently running, and that restrict access to the host on which they run.

Legion Vaults: Vaults hold OPR's for other Legion objects. In response to save_state() and restore_state() member functions, an object reads and writes its state from and to its OPR, which typically exists on persistent storage. A Vault has access to the OPR's it holds via mechanisms outside of Legion (e.g., a shared Unix file or directory). When an object is created, a Vault for the object is chosen by the object's class, and the Vault supplies an empty OPR. The object's save_state() method can then write its state into this OPR before being deactivated, and its restore_state() method can read it out upon being reactivated. Sometimes an object's class (or the Placement Mapper on behalf of that class) might want to migrate an object to another Legion Host. This could require moving the OPR to a Vault that can share an OPR with an object running on the new host. It is up to the class (or Placement Mapper) to select such a Vault, and up to the Vaults to transfer the OPR. The mechanism for saving and restoring state in Legion is described in more detail in [5].

4.5. Class objects

Each class object exports class-mandatory member functions to create new instances (create()) and subclasses (derive()), to delete instances and subclasses (delete()), and to find instances and subclasses (get_binding()). A class object is responsible for assigning LOID's to its instances and subclasses upon their creation. For its instances (non-class objects), the class object can construct the LOID completely locally; it assigns the Class Identifier portion to match its own Class Identifier, and uses the Instance Number field in any way it sees fit, most likely as a sequence number to guarantee that all LOID's are unique. To assign an LOID to a new subclass, the class object contacts LegionClass to obtain a new Class Identifier. This allows LegionClass to be an authority for finding class objects. Conventionally, the Instance Number portion of a class object's LOID is set to zero.

To perform the functions for which it is responsible, each class object must *logically* maintain a table whose entries contain fields for an LOID, an Object Address, a Placement Mapper, a Current Vault Set, and a Candidate Vault Set. In practice, the class object may employ other Legion objects, such as database servers, to maintain some or all of the information that class objects are required to maintain in what we refer to as the "logical table." Each row in the table corresponds to an object that the class object created—an instance or a subclass. The intended uses of each field are described below:

- *LOID*: The LOID names the object for which the entry contains information.
- *Object Address*: The Object Address field contains either the Object Address of the object (if the object is currently Active and the class knows its Object Address), or NIL (if the object is currently Inert). This field is used to respond to get_binding() requests from Binding Agents and other Legion objects.
- *Placement Mapper*: The Placement Mapper field contains the LOID of the object that is responsible for assigning the object entered in the table to a Legion Host when it is about to be activated. This mapping decision is intentionally left out of the core object model, except for a few "hooks" (including this one) that allow other Legion objects to implement scheduling policies. It is expected that each class will have a default Placement Mapper that is inherited by each of its objects unless a different Mapper is explicitly specified.
- *Current Vault Set*: The Current Vault Set field contains a list of Vaults that currently have Object Persistent Representations for an object. Typically, only one Legion Vault will have a copy of the Object Persistent Representation of an object.
- *Candidate Vault Set:* The Candidate Vault Set field indicates the Vaults that may be given responsibility for the object. This field could be implemented as a simple list, but more likely it will need to encapsulate more sophisticated information, such as "no restriction" or "all Vaults with a given security policy."

Objects may be given the opportunity by their class to directly manipulate these fields. In this way, the Legion class mechanism is reminiscent of reflective architectures.

5. Mechanism

The components described in the previous sections are intended to support the operations that are necessary for wide area parallel processing. Two of the most common and important of these operations—binding and object creation—are described in detail below.

5.1. Binding

This section describes a typical process by which a Legion Object Identifier gets bound to an Object Address. Recall that LOID's are meaningful only at the Legion level, and that the underlying communication facilities upon which Legion relies must be given lower level names in order to allow objects to communicate. Thus, LOID's must be bound to Object Addresses, which can in fact encapsulate names that are meaningful to underlying facilities.

The binding process is intended to be completely hidden from the vast majority of Legion users. Thus, it will typically be carried out by the various compilers and run-time systems that comprise Legion. A user will write a Legion application program in her favorite language, and will typically name Legion objects with string names. The program is compiled within a particular "context" by a Legion-aware compiler. The compiler uses the context to map string names to LOID's, which then become embedded within Legion executable programs. As the object executes, the run-time system interprets the LOID's and binds them to Object Addresses as described below. The reader should keep in mind that the binding model is key to the scalability of Legion—a poor design would seriously limit scalability.

The model: A class is ultimately responsible for providing bindings to its instances and subclasses. But to make the binding process scalable, and to distribute functionality, control, and responsibility appropriately, the object model introduces other objects to the binding process. Suppose that object A wishes to bind the LOID for object B, which is an instance or subclass of class C. The following Legion objects are potentially involved in the binding process: A, A's Binding Agent, C, and LegionClass. The role of each of these objects is described below.

Details: Object A begins the binding process by generating a reference to the LOID of B. Since A is a Legion object, it contains a Legion-aware communication layer which implements a binding cache. Therefore, A will often have a cached binding for B, and external objects will be unnecessary. If A does not contain a cached binding, it invokes the get_binding() member function on its Binding Agent, for which it is guaranteed to have an Object Address as part of its persistent state. The Binding Agent may have a binding for B's LOID in its cache, in which case it simply responds to A with a binding for B. If the Binding Agent does not have a cached binding, it may undertake any process it wishes in order to generate or locate a binding for B's LOID. In particular, the Binding Agent may consult other Binding Agents, which may be organized in a hierarchy to allow the binding process to scale.

Sometimes, a Binding Agent will be unable to locate a binding for B by any means other than contacting class object C. Recall that B is an instance or subclass of C, which is therefore responsible for finding B. We delay the discussion of how to find C until the next section, and assume for now that it can be done. A's Binding Agent invokes the get_binding() member function on C, which in turn consults its logical table (Section 4.5). If the Object Address field for the appropriate entry in the logical table is not empty, then C can construct and return a binding. The returned binding is passed back through the objects, each of which may cache it.

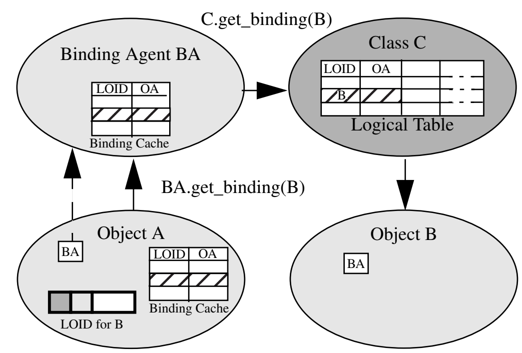

FIGURE 5. A typical binding process. Object A generates a reference to B, and contacts its Binding Agent for a binding. The Binding Agent checks its local cache, and then consults C, the class that created B. Table entries that are filled with diagonal lines show the places where a binding for B may be cached.

Finding the responsible class object: Omitted from the above discussion is an explanation of how C, the class responsible for locating B, is itself located. At first glance, this would seem to be as difficult as finding B. However, several characteristics of the object creation process and of Legion classes combine to make it a different problem—one that can be solved in a efficient and scalable fashion.

Recall that B is either an instance or a subclass of C. Therefore, C is a class object with an associated unique Class Identifier, which was assigned by LegionClass. Thus, LegionClass can be the authority for locating class objects. LegionClass does not directly maintain the bindings; instead, it delegates that responsibility to other class objects. To do so, LegionClass maintains a mapping of LOID pairs. The existence of pair <X,Y> indicates that X is responsible for locating Y. When a new class object D is created, the creating class C contacts LegionClass for a new Class Identifier to assign to the class. At this time, LegionClass can record that C is responsible for locating D by constructing and maintaining the pair <C,D>. When objects are trying to locate class object D, LegionClass can point them toward C. When objects are trying to locate a non-class object N, the process is even simpler; the LOID of the responsible class can be determined by setting the Class Identifier field to match that of N, and by setting the Instance Number field to zero.

Notice that we now have the LOID of the responsible class C. Thus, the binding process may need to be repeated in order to locate C, and again to locate C's superclass, and so on. Since all classes are eventually derived from LegionClass, the process can end when the responsible class is LegionClass itself. In this case, LegionClass simply hands out the appropriate binding which, as a class object, it is responsible for maintaining.

While this process may seem to scale poorly, extensive caching of both bindings and "responsibility pairs" ensures that the vast majority of accesses occurs locally. A more extensive argument for the scalability of the binding process is included in Section 6.

5.2. Object creation

As with the binding process described above, the creation of Legion objects is intended to be initiated by normal Legion programs via the mechanisms that the programs' implementation languages support. In C++, for instance, the creation of a non-class object might be triggered by the use of the keyword "new." The creation of a new class object might result from using the C++ inheritance mechanism to derive a new class. The Legion-aware compiler for the language creates code to call the create() or derive() member function on the appropriate class object, using the local context to map a string name to the intended Legion LOID.

When a class object receives a request to create a new instance or subclass, it must do so with the cooperation of the Legion Host for the host on which the new object will initially run. Selecting these two objects is a scheduling decision that is left up to the class, which may choose to employ the services of a Placement Mapper [11]. Some classes may allow the creating object to suggest a Legion Vault, a Legion Host, or both. At any rate, the actual creation of the object is carried out by the Legion Host, which is given enough information by the class to allow it to create the new object. This information could take the form of an executable program, the name of an executable, a list of steps to follow, etc.

Once the Legion Host has physically created the new object, using information provided by the class, the Legion Host returns the Object Address of the newly created object. The class object then stores the information for future use and returns the new LOID to the object that issued the request to instantiate. Alternatively, the class object could choose a different semantics in which, rather than creating a new instance in response to a request to instantiate, it "reuses" an existing object and either returns the existing object's LOID or a new LOID that maps to the same physical object. Another option is to multiplex multiple LOID's to the same object address to conserve address spaces or to improve inter-object communication.

The point is that the class object can implement whatever semantics it desires for either intantiating objects or binding LOID's to Object Addresses. We will provide default implementations that will be good enough for most classes, but the ability to reimplement core functionality provides a tremendous amount of flexibility to the class designer.

6. Scalability

Scalability is an important challenge to a system that is intended to contain millions of sites and trillions of objects. Before a system can be described as scalable, a precise definition of exactly what it means to be scalable must be formulated—scalability is a term that is used in different ways by different people. Typically, a scalable architecture refers to one that has the property that as the number of processors increases, the granularity of computation does not need to increase to keep the machine balanced. Thus, the machine can be scaled up to an arbitrary number of processors. Architectural scalability is claimed by many different architectures, including hypercubes, meshes, tori, and rings. But as Reed [15] points out, scalability of an architecture must be claimed with respect to a particular application and the communication patterns that the application exhibits. For example, a two dimensional torus or mesh is scalable with respect to 2-D nearest-neighbor stencil applications such as computational fluid dynamics. However, the architectures are not scalable with respect to applications that exhibit random communication patterns. The hypercube, however, is scalable with respect to random communication.

In distributed systems, scalability is best summed up by the "distributed systems principle"—that is, the number of requests to any particular system component must not be an increasing function of the number of hosts or objects in the system. Our claim is that as the number of Legion hosts and objects increases, no component will become a bottleneck that limits performance and restricts growth.

We make two assumptions about the Legion system. First, we assume that most accesses will be local. By local, we mean within the same organization, for instance within a department or university campus. If this assumption does not hold, then the scalability of Legion will depend on the scalability of the underlying interconnect. We do not expect the underlying wide area network to be scalable in the parallel architecture sense. The second assumption is that class objects will not migrate frequently, and further, that they will tend to stay active for long periods of time relative to instance objects.

With these assumptions in mind, let us examine where communication and interaction in Legion occur. First, consider communication that occurs between user level objects inside of an application. This communication may or may not contain a bottleneck. The user may have chosen an implementation with a centralized object that acts as shared memory for a large number of workers. The object could very easily become a bottleneck and limit application performance. This does not mean that Legion is not scalable; it simply means that the *application* is not scalable. Legion does not guarantee that all applications written using Legion as the underlying fabric will be scalable.

Instead, our claim to scalability refers to communication traffic that is required as a part of the Legion implementation model. This traffic is concentrated in two areas—LOID binding lookups from objects to Binding Agents, and Binding Agent traffic required to satisfy object binding requests. We consider each separately below.

Object to Binding Agent traffic: Each Legion object will maintain a cache of bindings. Therefore, an object's Binding Agent will only be consulted on a local cache miss, or when a stale binding is encountered. The Legion system will include many Binding Agents, and each object may select its Binding Agent based on its charge rate, its performance, or other criteria. As the load on a particular Binding Agent increases, or as the domain serviced by a particular agent enlarges, more Binding Agents may be created. Thus, each Binding Agent can be set up to service a bounded number of clients, ensuring that object to Binding Agent traffic is scalable.

Traffic induced by Binding Agents: Recall that on a cache miss, a Binding Agent must find a binding. If all requests went to a single "master" Binding Agent, the system would not scale. Instead the Binding Agent consults the class object of the object for which it needs a binding. Thus, the load is distributed to the class objects. This raised two concerns: (1) Given the way that class objects are located, won't LegionClass become a bottleneck, and (2) Won't commonly used classes—for instance file classes—also become a bottleneck?

The Binding Agent can acquire the binding for a class object by consulting LegionClass, or by consulting another Binding Agent. Under the assumptions that class bindings change very slowly and Binding Agents cache class object bindings, the traffic to LegionClass will be reduced. Further, by constructing a k-ary tree of Binding Agents, eliminating traffic from "leaf" Binding Agents to LegionClass, we can arbitrarily reduce the load placed on LegionClass. In essence, Binding Agents could be organized to implement a software combining tree [19].

The problem of popular class objects becoming bottlenecks can be alleviated by "cloning" class objects when they become heavily used. The cloned class is derived from the heavily used class without changing the interface in any way. New instantiation and derivation requests are passed to the cloned object, making it responsible for the new objects. Further, several clones can exist simultaneously, with the different clones residing in different domains.

Thus, Legion is scalable in the sense that the underlying mechanisms mandated by the system model have implementations that will scale to an arbitrary number of hosts and objects. However, it does not promise scalability for all applications—no architecture can do that.

7. Summary

This document has described the core Legion object model. The model places system-level responsibility in the hands of classes and objects that users can create and define themselves. Legion specifies the intended functionality of the core objects—LegionObject, LegionClass, Legion Hosts, Legion Vaults, and Binding Agents—which cooperate to create, locate, and manage the objects in the system. But Legion encourages users to implement and select replacements that meet the users' own particular requirements. This policy, in concert with the Legion security model, enables site autonomy by allowing resource providers to control their own resources. The Legion naming system—comprised of LOID's, Object Addresses, and bindings—unites the objects in the system, thereby facilitating access to remote files and data.

8. References

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team, "A Case for NOW (Networks of Workstations)," December 9, 1994, to appear IEEE Micro.

[2] Grady Booch, Object Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[3] The Castle Project, University of California, Berkeley, http://http.cs.berkeley.edu/projects/parallel/castle/castle.html.

[4] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., SunSoft, Inc., *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 93.xx.yy, Revision 1.2, Draft 29, December 1993.

[5] Adam J. Ferrari, Andrew Grimshaw, "Persistent Object State Management in Legion," University of Virginia Computer Science Technical Report CS-95-36, in progress.

[6] Ian Foster, Carl Kesselman, Steven Tuecke, "Nexus: Runtime Support for Task-Parallel Programming Languages," Argonne National Laboratories, http://www.mcs.anl.gov/nexus/paper/.

[7] Adele Goldberg, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

[8] Andrew Grimshaw, William A. Wulf, "Legion—A View from 50,000 Feet," *High Performance Distributed Computing-5*, August 1996.

[9] Andrew Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May 1993.

[10] Andrew Grimshaw, Anh Nguyen-Tuong, William A. Wulf, "Campus-Wide Computing: Results Using Legion at the University of Virginia," University of Virginia Computer Science Technical Report CS-95-19, March 27, 1995.

[11] John F. Karpovich, "Support for Object Placement in Wide Area Heterogeneous Distributed Systems," University of Virginia Computer Science Technical Report CS-96-03, January 16, 1996.

[12] Mike Lewis, Andrew Grimshaw, "The Core Legion Object Model," University of Virginia Computer Science Technical Report CS-95-35, August 1995.

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

[14] The Mentat Research Group, *Mentat 2.8 Programming Language Reference Manual*, Department of Computer Science, University of Virginia, 1995.

[15] Daniel A. Reed, Richard M. Fujimoto, Multicomputer Networks: Message-Based Parallel Processing, The MIT Press, Cambridge, Massachusetts, 1985.

[16] M. van Steen, P. Homburg, L. van Doorn, A.S. Tanenbaum, and W. de Jonge. "Towards Object-based Wide Area Distributed Systems". In L.-F. Carbrera and M. Theimer, (eds.), *Proceedings International Workshop on Object Orientation in Operating Systems*, pp. 224-227, Lund, Sweden, August 1995.

[17] Chenxi Wang, William A. Wulf, "A Distributed Key Generation Technique," University of Virginia Computer Science Technical Report CS-96-08, March 1996.

[18] William A. Wulf, Chenxi Wang, Darrell Kienzle, "A New Model of Security for Distributed Systems," University of Virginia Computer Science Technical Report CS-95-34, August 1995.

[19] Pen-Chung Yew, Nian-Feng Tzeng, Duncan H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, Vol. C-36(4), April 1987.