The Core Legion Object Model[1]

Mike Lewis
Andrew Grimshaw

{mlewis,grimshaw}@Virginia.edu
Department of Computer Science
University of Virginia
August 1995

Abstract

This document 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 persistent 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.

1. This is University of Virginia Computer Science Technical Report CS-95-35. It is located on the Web at ftp://ftp.cs.virginia.edu/pub/techreports/CS-95-35.ps.Z.

1 Objectives

The Legion project[1] at the University of Virginia is an attempt to design and build system services that provide the illusion of a single virtual machine to users. Legion targets wide-area assemblies of workstations, supercomputers, and parallel supercomputers. Legion tackles problems not solved by existing workstation-based parallel processing tools. It aims to provide shared object and shared name spaces, application adjustable fault-tolerance, improved response time, greater throughput, wide area network support, management and exploitation of heterogeneity, protection, security, efficient scheduling, comprehensive resource management, parallel processing, and object inter-operability.

This document 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 persistent 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 the system's 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.

A single persistent name space unites the objects in the Legion system. This makes remote files and data more easily accessible, thereby facilitating the construction of applications that span multiple sites.

Site autonomy is provided in the object model by logically partitioning Legion resources into possibly non-disjoint sets, called JURISDICTIONS, and by distributing control of these resources among an extensible set of Legion objects, called MAGISTRATES. 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 is 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.

1. For more information about Legion, see our Web page at http://www.cs.virginia.edu/~legion/

Security and fault tolerance have had significant impact on the design of the object model, which enables the realization of both objectives. The Legion security model is described in detail in [8], and fault tolerance is addressed in [6].

The remainder of this document describes the core Legion object model, characterizes its main components, presents an implementation model for its realization, describes the mechanism it is intended to support, and argues that the model scales well. We describe an evolving model that includes several aspects that have yet to be addressed in detail, not a model that is set in stone.

2 Overview

Legion is an object-oriented system comprised of independent, address space disjoint objects that communicate with one another via method invocation. 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 inherited from its class. Legion class interfaces can be described in an Interface Description Language (IDL).[1] An instance of Legion is partitioned into autonomous Jurisdictions, each of which consists of a set of hosts and associated storage. Jurisdictions manipulate object representations and state directly.

2.1 Object model

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 *MayI()*, *SaveState()*, and *RestoreState()*. Class objects export an additional set of CLASS-MANDATORY member functions, including *Create()*, *Derive()*, and *InheritFrom()*. A class object is responsible for creating and locating its instances (non-class objects) and subclasses (other class objects). A new class object is created by calling

1. At least two different IDL's will be supported by Legion: the CORBA IDL Interface Definition Language[2], and the Mentat Programming Language (MPL)[5].

the Derive() member function on an existing class object. Similarly, a new non-class object is created by calling the Create() member function on an existing class object.

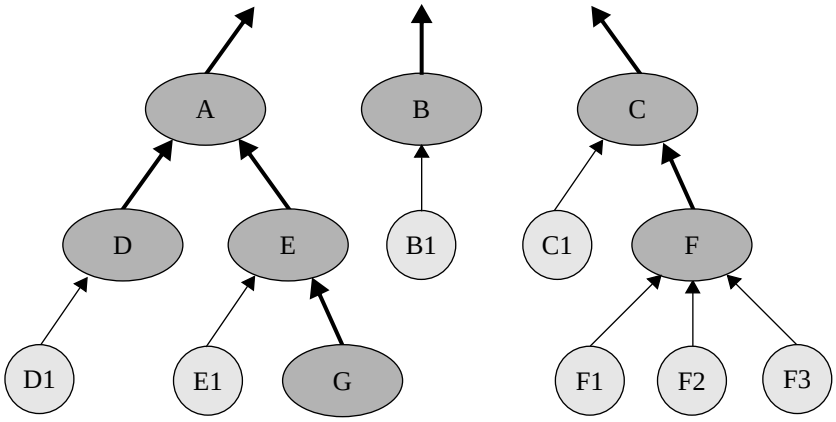

FIGURE 1. Objects A-G are class objects. D and E are derived from A, F is derived from C, and G is derived from E. Each of the other objects is a non-class object—an instance of its class. This figure does not depict multiple inheritance, which is supported in Legion.

A class that is derived from another class inherits the superclass's member functions and variables.[1] Multiple inheritance is fully supported in Legion via the class-mandatory member function InheritFrom(). Invoking InheritFrom() on an existing class object A, and passing the name of an existing class object B, causes A to inherit from B. Thus, inheritance in Legion is an active process that is carried out at run-time.

2.1.1 Relationships between Legion objects

The class-mandatory member functions Create(), Derive(), and InheritFrom(), define three different relations that can exist between Legion objects: the IS-A relation, the KIND-OF relation, and the INHERITS-FROM relation, respectively. Figure 2 introduces the Legion conventions for graphically depicting these three types of relationships.

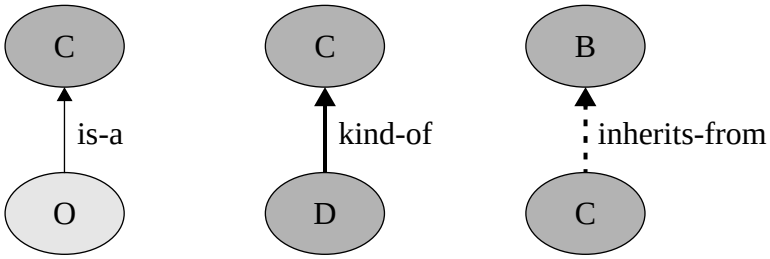

FIGURE 2. Conventions for depicting relationships between objects in Legion. A thin solid arrow from non-class object O to class object C indicates that O is-a C. A thick solid arrow from class object D to class object C indicates that D is a kind-of C. A thick dotted arrow from class object C to class object B indicates that C inherits-from B.

Each type of relation is described below:

1. Legion may allow a class to select the components that it wishes to inherit from its superclass.

- is-a: The is-a relationship can exist between a non-class object and a class object. The is-a relationship results from instantiating new Legion non-class objects. This happens as a result of invoking the Create() member function on an existing class object. Suppose C is the class object whose Create() function is invoked. C creates an instance of itself as non-class object O, which then exports the interface that C defines. We say that O is an instance of class C, or equivalently, that O is-a C. Non-class object O inherits the Legion object-mandatory member functions, but not the Legion class-mandatory member functions (since it is not a class). As O's creator, C is responsible for being able to locate O. Classes typically instantiate many objects, but an object belongs to exactly one class.

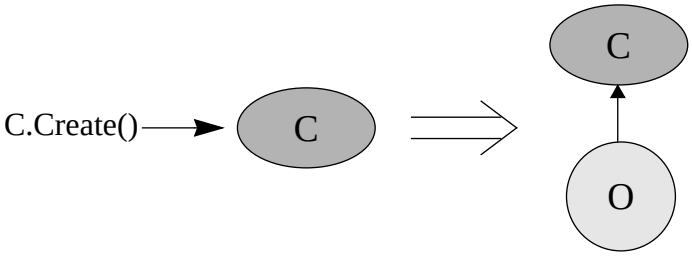

FIGURE 3. Invoking the Create() member function on an existing class object C causes an instance, O, to be created. We say that O is-a C.

- kind-of: The kind-of relationship can exist between two class objects. This happens when the Derive() member function is invoked on an existing class object. Suppose C is the class object whose Derive() function is invoked. C creates a new class object D that inherits all of the Legion object- and class-mandatory member functions, and some or all of the member functions and data structures particular to C. As D's creator, C is responsible for being able to locate D. We say that D is a subclass of C, that C is the superclass of D, and equivalently, that D is a kind-of C. A class can be the superclass for any number of different subclasses, but it is the subclass of exactly one superclass.

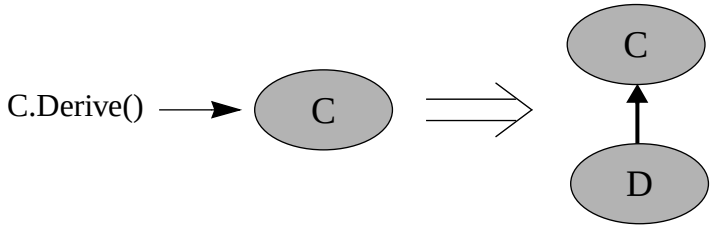

FIGURE 4. Invoking the Derive() member function on an existing class object C causes a new class object, D, to be created. We say that D is a kind-of C.

- inherits-from: The inherits-from relationship can also exist between two class objects. This happens when the InheritFrom() member function is invoked on an existing class object. This function does not cause any new objects to be created; instead, it serves to alter the composition of future instances of the class. Suppose C is the class object whose InheritFrom() function is invoked with the name of class object B as an argument. This causes B's member functions to be added to C's interface. Unlike with the

is-a and kind-of relations, B has no responsibility for locating C. We say that B is a base class for C, or equivalently, that C inherits-from B. A class can inherit from, and be a base class for, any number of other classes.

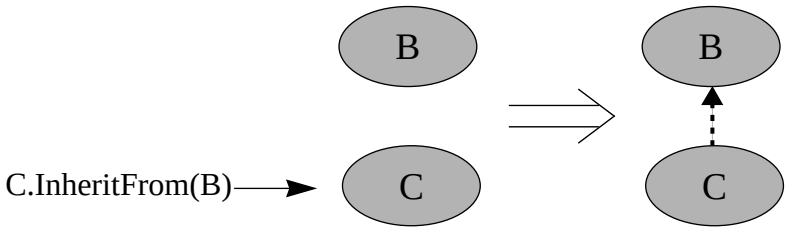

FIGURE 5. Invoking the InheritFrom() member function on an existing class object C, and passing the name of another existing class B, causes C to inherit from B.

Thus, multiple inheritance in Legion is a two step process. First, the class is created by calling Derive() on an existing class object. Second, the composition of future instances of the class is set via calls to the InheritFrom() method in the new class object. When the instances of the class are created via the Create() method, their composition reflects the way the class was defined in the inheritance process.

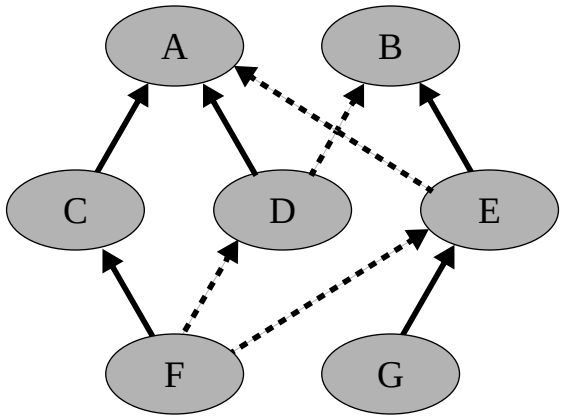

FIGURE 6. Legion classes can be multiply derived from other Legion classes. In Legion terminology, each class object has exactly one superclass, but can have multiple base classes. For example, C is F's superclass, while D and E serve as F's base classes.

The Legion inheritance mechanism is still in the process of being designed. Thus, many concepts are intentionally not addressed, or are discussed only in vague terms in this document.

2.1.2 Legion class types

The creators of a Legion class may overload or redefine any of Create(), Derive(), and InheritFrom() to be possibly empty member functions. This leads to three special types of Legion classes. A class object whose Create() function is empty is said to be ABSTRACT; no direct instances of an Abstract class can exist. A class object whose Derive() function is empty is said to be PRIVATE; Private class objects can have no derived classes, just

instances. A class object whose InheritFrom() function is empty is said to be FIXED; a Fixed class inherits member functions and variables only from its superclass.

2.1.3 Legion's core Abstract class objects

Legion defines the interface and functionality of several core Abstract class objects, including *LegionObject*, *LegionClass*, *LegionHost*, *LegionMagistrate*, and *LegionBindingAgent*. Each of these is introduced below, and discussed in more detail in subsequent sections.

- LegionObject: LegionObject provides the full set of object-mandatory member functions. The class object for LegionObject is the only sink in the graph that is implied by the union of the kind-of and is-a relations. That is, 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: 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.

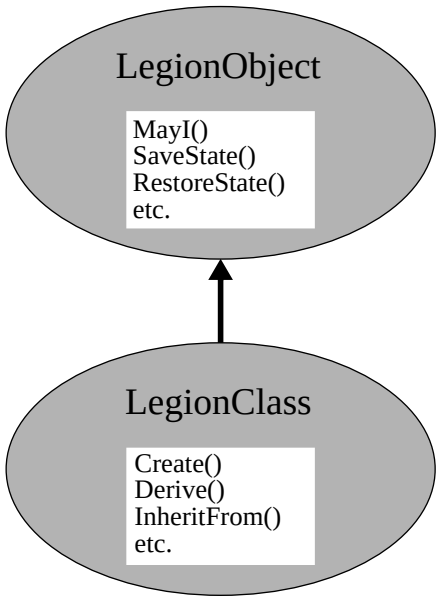

FIGURE 7. Legion defines two Abstract base classes—LegionObject and LegionClass—that define the object- and class-mandatory member functions respectively. LegionClass is derived from LegionObject; thus, classes are objects in Legion.

- LegionHost: LegionHost models Legion hosts. To be included in Legion[1], a host must run a Legion HOST OBJECT, which is an instance of some class that is eventually derived from LegionHost. For example, UnixHost and SPMDHost might be two different Legion classes derived directly from class LegionHost. More specific Host classes

might be derived from each of these, as shown in Figure 8. A Sun workstation might run an instance of class UnixHost, whereas a Silicon Graphics Power Challenge might run an instance of UnixSMMP, a class derived from UnixHost.

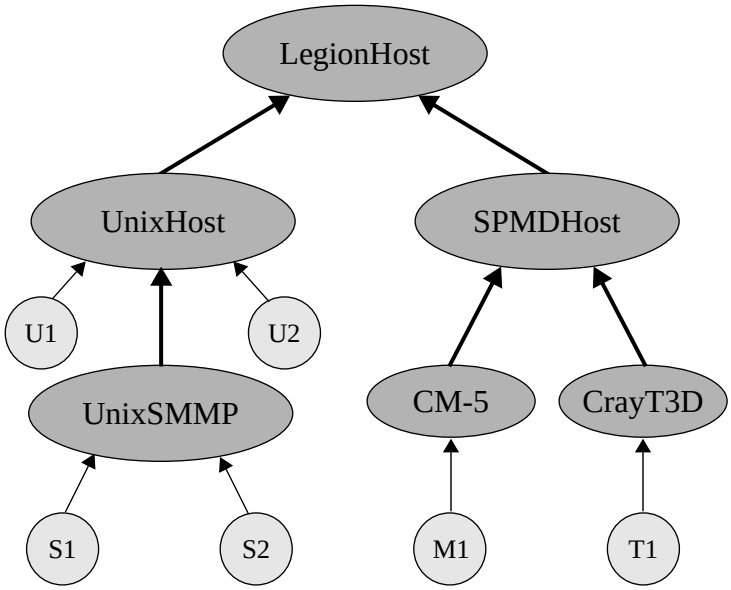

FIGURE 8. The Legion class LegionHost is the root of all classes whose instances are Legion Host Objects. In this figure, UnixHost and SPMDHost are derived directly from LegionHost. UnixSMMP is derived from UnixHost, and CM-5 and CrayT3D are derived from SPMDHost. The figure shows six different Host Objects: two instances of both UnixHost and UnixSMMP, and one instance of both CM-5 and CrayT3D.

- LegionMagistrate: LegionMagistrate models Legion Magistrates—those objects that manage the activation, deactivation, and migration of Legion objects in a particular Jurisdiction. All Legion Magistrates are eventually derived from class LegionMagistrate. Resource providers can build Magistrates that meet their own security and resource access requirements. For example, suppose the Department of Energy (DOE) does not trust university graduate students to write a Magistrate class that adequately protects its objects. The DOE can write its own Magistrate, and insist via the class mechanism that all objects that the DOE owns execute only on Magistrates that it trusts. Further, it can ensure that their Magistrates only use Host Objects that have been certified by the DOE not to leak information. An additional benefit of this mechanism is that users can choose their favorite service providers, potentially creating a market in ser-

1. A host being “included in Legion” in this sense refers to Legion’s ability to execute objects on the host. Legion also includes the notion of “client” hosts that can access Legion resources without themselves being Legion resources.

vice provision. For example, national laboratories, who also may not trust university graduate students, may choose to trust the DOE, and use the DOE implementations. Alternatively, a commercial provider may be used.

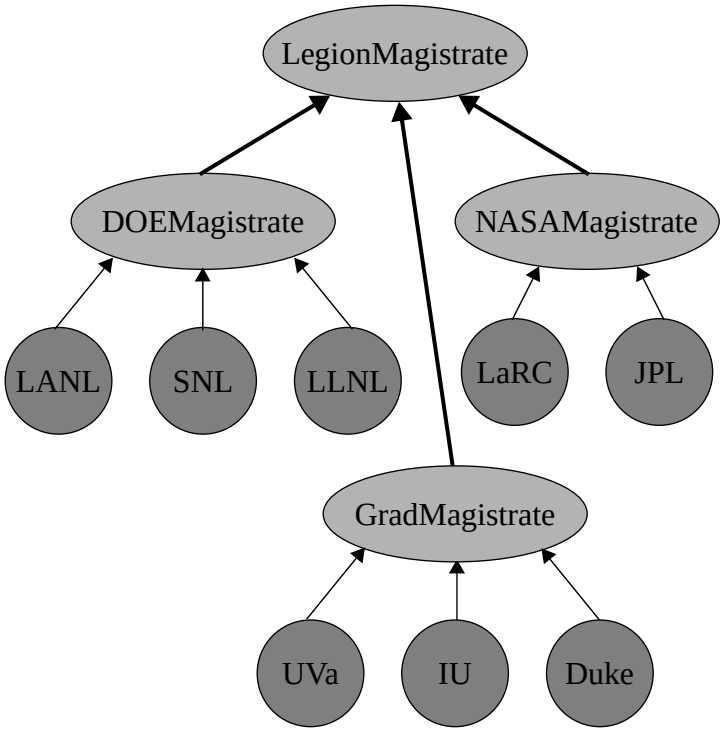

FIGURE 9. DOEMagistrate, NASAMagistrate, and GradMagistrate are class objects derived directly from LegionMagistrate. They will likely implement different security and resource access policies.

- LegionBindingAgent: LegionBindingAgent models Legion BINDING AGENTS—those objects that map Legion names[1] to physical addresses in order to enable inter-object communication. The binding problem is discussed extensively later in the document.

2.1.4 Trust and security

The fact that Legion's core objects can be reimplemented by users reflects the Legion philosophy on trust. In wide-area systems, trust is an important issue that contains several different aspects. Who do users trust? Who do physical resource owners trust? Who do software resource (databases and applications) owners trust? Indeed, what exactly does it mean for A to trust B? Is trust absolute, or is it relative?

There can be no single answer to the "who do you trust?" questions. In particular, we cannot mandate that users and resource owners must trust "Legion." If we do, then many will not participate because we will be unable to convince them that Legion is trustworthy, that there are no Trojan horses, that they have a legitimate copy of Legion, that we won't leak their persistent object state to other users, etc. Instead, Legion provides mechanism that

1. Legion "names" are described in Section 3.2.

allows each entity to choose who or what it trusts; in particular, resource owners can provide their own, trusted by them, implementations of Legion functions and objects. Thus, users and resource owners may provide their own implementations of Magistrates, Host Objects, and Binding Agents by deriving off of the Abstract classes described above.

2.2 Jurisdictions and Magistrates

Legion is divided into Jurisdictions. A Jurisdiction consists of some aggregate persistent storage space and a set of Legion hosts. Jurisdictions are potentially non-disjoint; both hosts and persistent storage may be contained in two or more Jurisdictions, and Jurisdictions can be organized to form hierarchies. The union of all Jurisdictions comprises the full Legion system.

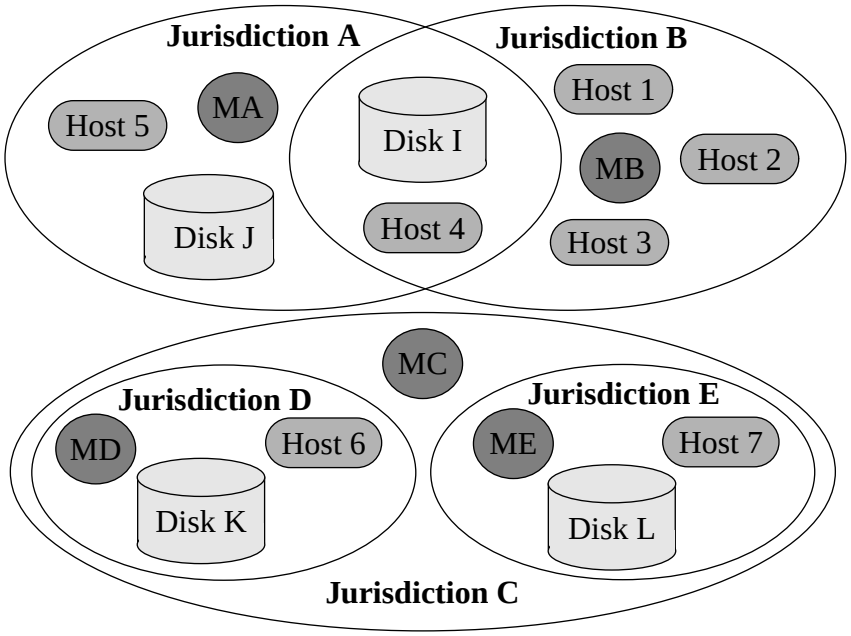

FIGURE 10. Jurisdictions, which contain hosts and persistent storage, may overlap (e.g. A and B), may be disjoint (e.g. A and C), or may form hierarchies (e.g. C, D, and E). Magistrates, one per Jurisdiction, manage objects and resources; MA is the Magistrate for Jurisdiction A, MB is the Magistrate for Jurisdiction B, etc. Typically, Jurisdictions will contain many more hosts and disks than shown in this figure.

Jurisdictions are the mechanism by which Legion provides site autonomy to participating organizations. Sites can offer their resources to Legion, and can insist that they be managed only by objects that the sites trust. In particular, an organization may choose to implement its own Magistrate to manage its Jurisdiction. Alternatively, it may trust another, possibly commercial, organization's Magistrate class, and choose an instance of that class to manage its resources. The organization could also simply put its resources under the control of another Magistrate. By providing each of these alternatives, Legion allows an organization to ensure that its resources are protected to its satisfaction.

Jurisdictions also make the Legion system scalable and extensible. No single Magistrate is responsible for managing the entire Legion system. Instead, control is completely decentralized. Further, if a Jurisdiction's resources impose a substantial load on its Magistrate,

the Jurisdiction can be split, and a new Magistrate can be created to take over responsibility for some of the resources and objects.

2.3 Host Objects

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. Thus, the Host Object for a host is ultimately responsible for deciding which objects can run on the host it represents. Since Host Objects 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. Further details about Host Objects are included in Section 3.9.

2.4 Security

Legion does not attempt to guarantee security to its users. Instead, we (1) are as precise as possible about the degree of confidence that a user can have, (2) make that confidence "good enough" and "cheap enough" for an interestingly large selection of users, and (3) provide a context that allows the user to gain the additional confidence that she requires with a cost that is intuitively proportional to the added confidence she gets.

Our security model is based on three principles: (1) as in the Hippocratic Oath, *do no harm*, (2) *caveat emptor*, let the buyer beware, and (3) *small is beautiful*. In the final analysis, users are responsible for their own security. Legion provides a model and mechanism that make it feasible, conceptually simple, and inexpensive in the default case. But in the end, the user has the ultimate responsibility to determine what policy is to be enforced and how vigorous that enforcement will be.

In the Legion security model, every object provides certain security-related member functions, including MayI() and *Iam()*. These functions may default to empty for the case of no security. User-defined objects play two security related roles—Responsible Agent (RA) and Security Agent (SA). To play these roles, they provide additional member functions that are known to Legion. Every method invocation is performed in an environment consisting of a triple of object names—those of the operative Responsible Agent, the Security Agent, and the Calling Agent. The general approach is that Legion will invoke the known member functions to define and enforce security, thus giving objects the responsibility of defining and ensuring the policy they choose.

The Legion security model is described in detail in [8].

3 Implementation model

3.1 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 one or more of the hosts in a Jurisdiction, and is described by an OBJECT ADDRESS (Section 3.4). When an object is Inert, it exists in persistent storage somewhere in a Jurisdiction, is described by an OBJECT PERSISTENT REPRESENTATION, and can be located using an OBJECT PERSISTENT ADDRESS. Because of the way that objects are suspended and restarted by Legion[3], all of a Jurisdiction's persistent storage space must be visible from each of its hosts. Magistrates are responsible for moving objects between Active and Inert states, and for migrating objects between Jurisdictions.

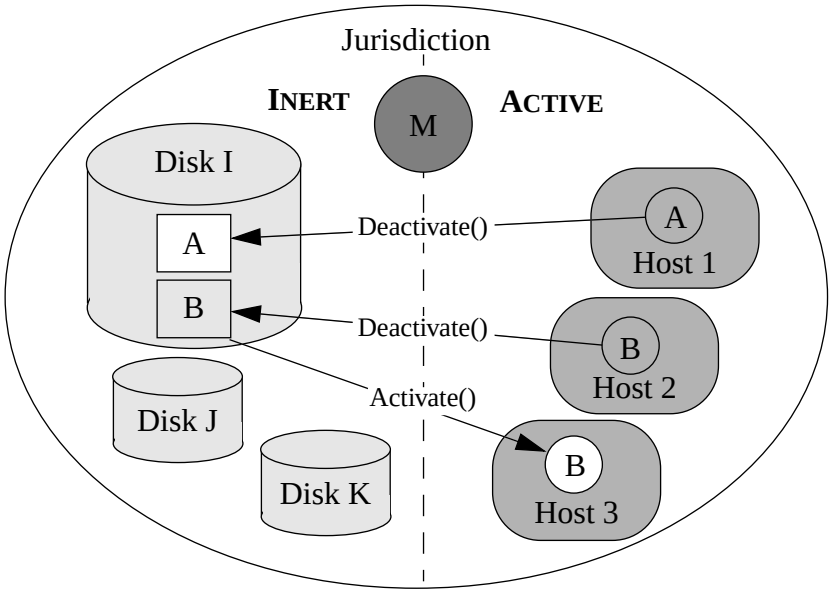

FIGURE 11. A sample Jurisdiction comprised of three disks (I, J, and K) and three hosts (1, 2, and 3); all three hosts can access all three disks directly. Objects A and B belong to the Jurisdiction and are moved between Active and Inert states by the Magistrate. Object A has been deactivated into an Object Persistent Representation on Disk I, and B has been migrated from Host 2 to Host 3 through Disk I.

3.1.1 Object Persistent Representations and Addresses

An Object Persistent Representation is a sequential set of bytes that represents an Inert object, and that can be used by a Magistrate to activate the object. An executable file could be an Object Persistent Representation for an object that has yet to become Active. However, once an object is activated, it may acquire state information that would need to be stored as part of the Object Persistent Representation. Every Legion object will export functions to save and restore its state, and Magistrates will call these functions to create and interpret an Object Persistent Representation of the object. The mechanism for saving and restoring state is described in [3].

The Object Persistent Address of an Inert object is analogous to the Object Address of an Active object. An Object Persistent Addresses will typically be a file name, and will only be meaningful within the Jurisdiction in which it resides.

3.2 Legion Object Identifiers

Every Legion object is named by a LEGION OBJECT IDENTIFIER (LOID). The 128 high order bits are separated into CLASS IDENTIFIER (64 bits) and CLASS SPECIFIC (64 bits) parts. The P low order bits comprise the PUBLIC KEY of the object and will be used for security purposes.[1]

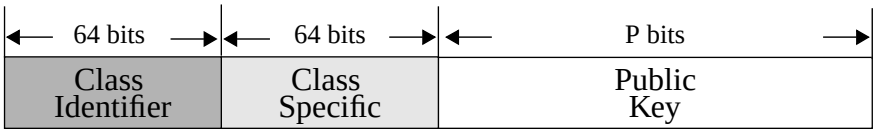

FIGURE 12. An LOID is comprised of a 64 bit Class Identifier, a 64 bit Class Specific field, and a P bit Public Key.

LegionClass is responsible for handing out unique Class Identifiers to each new class. The Class Specific portion is set to zero for all class objects, and can be used by classes to provide a unique LOID to each instance of the class. While it is likely that the Class Specific field will often be used by classes as a sequence number to guarantee the generation of unique LOID's, Legion does not restrict how any particular class sets this portion of an LOID that it generates.

3.3 The binding problem

Legion uses standard protocols and the communication facilities of host operating systems to support communication between Legion objects. However, Legion object names—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 the underlying protocols and communication facilities. The general problem, depicted in Figure 13, 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.

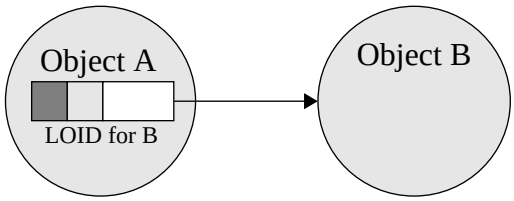

FIGURE 13. Object A has the LOID for object B, and must bind it to an Object Address to communicate directly with B.

1. P is a constant whose size has yet to be determined.

To solve the binding problem, Legion defines Object Addresses and bindings, and specifies the core functionality of Binding Agents, Magistrates, Host Objects, and class objects, which combine to locate and manage the objects in the system. These components are described in the sections below.

3.4 Object Addresses

An OBJECT ADDRESS ELEMENT contains, at the highest level, two basic parts: a 32 bit address type field, and 256 bits of address specific information. The address type field names the type of address (e.g. IP, XTP, etc.) that is contained in the other 256 bits. We envision that the first and most common type of address will be IP. For a normal IP address, 48 of the 256 bits will be utilized: 32 bits for the IP address, and 16 bits for a port number. On multiprocessors, a 32 bit platform-specific internal node number may be used to distinguish each particular processor.

An Object Address is a list of Object Address Elements, along with semantic information that describes how to utilize the list. The address semantic is intended to encapsulate various forms of multicast communication. For example, the semantic 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 options will likely be made.

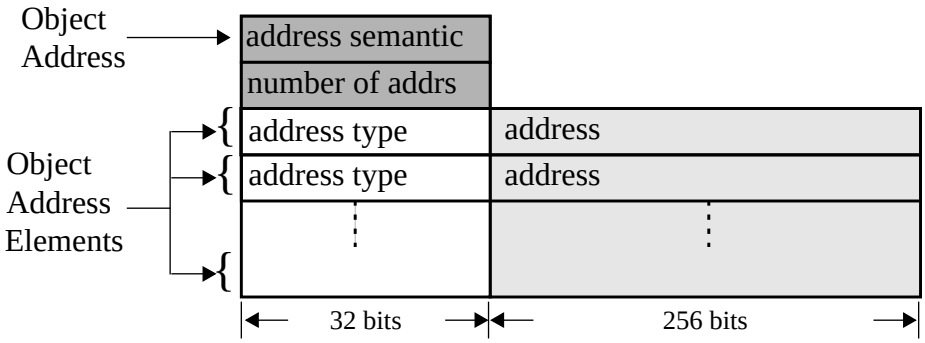

FIGURE 14. An Object Address is comprised of a list of physical addresses along with semantic information that describes how the list is to be used.

3.5 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.

3.6 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. But 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[1] 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 4.1.

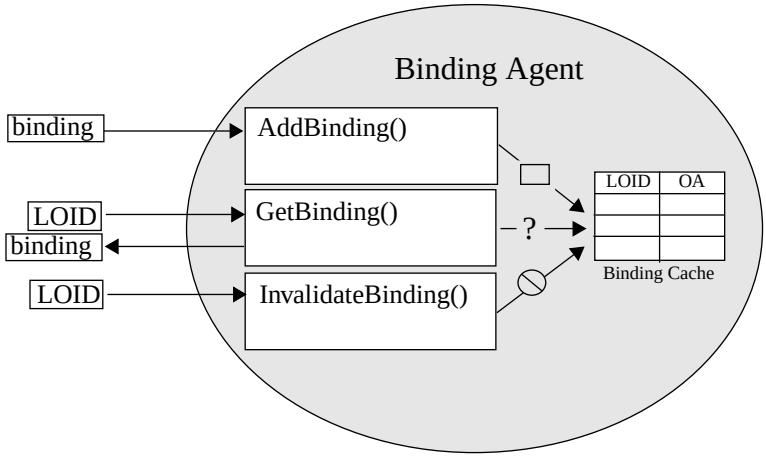

FIGURE 15. A typical Binding Agent maintains a cache of bindings, and responds to member function calls to add, return, and invalidate bindings.

LegionBindingAgent has the following member functions:

- binding GetBinding(LOID), binding GetBinding(binding) : The o verloaded method *GetBinding()* 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. Further details about the binding process are included in Section 4.1.

1. For details about how the appropriate class object can be found, refer to Section 4.1.3.

- InvalidateBinding(LOID), InvalidateBinding(binding): The overloaded method *InvalidateBinding()* 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.
- AddBinding(binding): AddBinding() 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.

3.7 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 (GetBinding()). 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 Class Specific 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 Class Specific 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 the table depicted in Figure 16. 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.".

LOID	Object Address	Current Magistrate List	Scheduling Agent	Candidate Magistrate List

FIGURE 16. The logical table that Legion class objects maintain to keep appropriate information about instances and subclasses of the class.

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, or if the object is Active but the class doesn't know the Object Address). This field is used to respond to GetBinding() requests from Binding Agents and other Legion objects.

- Current Magistrate List: The Current Magistrate List field contains a list of Magistrates that currently have Object Persistent Representations for an object. Typically, only one Magistrate will have a copy of the Object Persistent Representation of an object. If the class object receives a GetBinding() request and the Object Address field is empty, the class object can consult a Magistrate in this list to get a binding for the object in question.
- Scheduling Agent: The Scheduling Agent field contains the LOID of the object that is responsible for scheduling the object entered in the table. Scheduling is intentionally left out of the core object model, except for a few "hooks" (including this one) that allow other Legion objects to suggest scheduling policies to Magistrates. It is expected that each class will have a default Scheduling Agent that is inherited by each of its objects unless a different Scheduling Agent is explicitly specified.
- Candidate Magistrate List: The Candidate Magistrate List field indicates the Magistrates 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 Magistrates with a given security policy." Therefore, a language mechanism that names or restricts sets of Magistrates might be appropriate; details of this mechanism have not been formulated.

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.

The full set of class-mandatory member functions has yet to be formulated. However, it will include at least Create(), Derive(), InheritFrom(), Delete(), GetBinding(), and *GetInterface()*.

3.8 Magistrates

A Magistrate is in charge of a Jurisdiction. Thus, a Magistrate manages a set of hosts and some aggregate persistent storage. The purpose of a Magistrate is to perform the activation, deactivation, and migration of the Legion objects under its control.

Magistrates have member functions that allow other objects to suggest how to schedule the objects in the Jurisdiction, and when and how to move objects between Active and Inert states. Magistrates are not intended to be complex decision making entities. Instead, they should act as mechanisms by which other Legion objects implement policies and algorithms. As a likely security boundary for the objects it manages, a Magistrate has the authority to reject requests.

The member functions exported by a Magistrate include (but will certainly not be limited to) the following:

- binding Activate(LOID), binding Activate(LOID,LOID): The overloaded *Activate()* function takes the LOID of the object to activate, and causes it to become a running process on one of the hosts in the Jurisdiction if the object isn't already Active. The LOID of a Host Object in the Jurisdiction of the Magistrate can be passed as a parame-

ter to allow a Scheduling Agent (or any other Legion object) to provide suggestions about where to run the object. The Activate() function returns a binding that contains the Object Address of the object once it has been activated.

- Deactivate(LOID): The *Deactivate()* function takes the LOID of the object to deactivate, and causes it to be removed from the host on which it is running, and to be placed on persistent storage in an Object Persistent Representation. The method by which this is done is described in [3].
- Delete(LOID): The Delete() function removes the object with the given LOID from existence. Both Active and Inert copies of the object are removed from the system. After a Delete() function is successfully executed, future attempts to bind the LOID to an Object Address will be unsuccessful. Stale bindings may exist, but will be eventually removed as objects unsuccessfully try to use them.
- Copy(LOID,LOID): The *Copy()* function can be used to tell the Magistrate to copy the object with the given LOID to another Magistrate which is named by the second parameter. This function causes the Magistrate to deactivate the object, creating an Object Persistent Representation, and to send the Object Persistent Representation to the other Magistrate. This function, along with *Move()*, is used to migrate objects between Jurisdictions.
- Move(LOID,LOID): The Move() function is equivalent to Copy() then Delete(). It serves to change the Magistrate that manages a given object. The first parameter names the object to move, and the second names the Magistrate to which to move it.

There will likely be many other member functions, especially having to do with scheduling, that all Magistrates will export. Magistrates will have some default scheduling behavior, but complex scheduling policies are intended to be implemented outside of the Magistrate in Scheduling Agents. The Scheduling Agents will implement their policies by making calls on the primitive scheduling functions exported by the Magistrates. The right set of functions has yet to be identified and defined.

Since a Magistrate is a likely security boundary for the objects it manages, it may choose to refuse to service any of the requests it is issued, depending on the security policy that it enforces. In this sense, member function calls on Magistrates should be thought of as requests rather than commands.

3.9 Host Objects

As described in Figure 2.3, a Host Object runs on each host that is included in the Legion system. It is likely that a Host Object will implement a security mechanism that will attempt to ensure that its member functions will be invoked only by its Magistrate. In a Unix-like implementation, all Legion objects that execute on a host will execute with the same privilege as the Host Object. Therefore, Host Objects will typically execute with minimal privilege. Individual sites may choose to grant Host Objects a higher privilege if they desire.

Host Objects are started from outside Legion, for example from a command line or shell script in the host operating system. Host Objects are responsible for contacting LegionHost to notify it of the Host Object's existence and address. Host Objects are started external to Legion because they are the mechanism by which objects are started; there is no Legion object to start Host Objects (see Section 4.2.1).

Host Objects export member functions that start or restart processes, that suspend processes that are currently running, and that restrict access to the host. The full set of member functions that a Host Object exports will include at least the following, Activate(), Deactivate(), *SetCPULoad()*, *SetMemoryUsage()*, and *GetState()*.

4 Mechanism

4.1 Binding

This section describes a typical process by which a Legion Object Identifier gets bound to an Object Address. Recall from Section 3.3 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. At run-time, the run-time system interprets the LOID's and binds them to Object Addresses as described below.

4.1.1 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, as in Figure 13, 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, LegionClass, B's Magistrate, and the Host Object for the host on which B is currently Active. The role of each of these objects is described below.

4.1.2 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 may implement 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 GetBinding() member function on its Binding Agent, for which it already has 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 Section 4.1.3, and assume for now that it can be done. A's Binding Agent invokes the GetBinding() member function on C, which in turn consults its logical table (Section 3.7). If the Object Address field for the appropriate entry in the logical table is not empty, then C can construct and return a binding. If the field is empty, then C can consult B's Magistrate, whose LOID is contained in the Current Magistrate List field. C invokes the Activate() method on the Magistrate, which returns a binding for B. Thus, referring to the LOID of

an Inert object can cause the object to be activated. The returned binding is passed back through the objects, each of which may cache it.

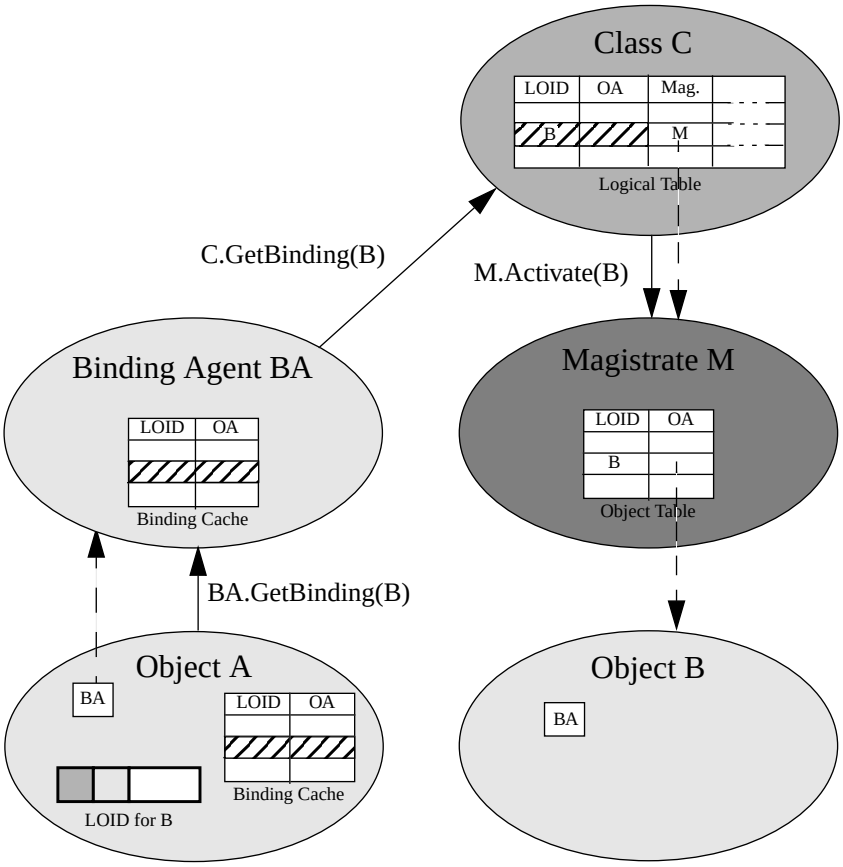

FIGURE 17. A typical binding process. Object A generates a reference to B, and contacts its Binding Agent for a binding. The Binding Agent checks its local cache, and then consults C, the class that created B. C may have to contact B's Magistrate M via the Activate() member function, and M may in turn have to activate B. Table entries that are filled with diagonal lines show the places where a binding for B may be cached.

4.1.3 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 Class Specific 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 5.

4.1.4 Stale bindings

Legion expects the presence of stale bindings—cached bindings containing Object Addresses that are no longer valid. Object Addresses can become invalid as a result of an object migrating, being deactivated, or being removed from the system. When an object attempts to communicate with an invalid Object Address, the Legion communication layer of the object is expected to detect that it has become invalid. When it does, it will likely request that the binding be refreshed. Some classes may even attempt to reduce the number of stale bindings by explicitly propagating news of an object's migration or removal.

4.2 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, most likely 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 Magistrate for the Jurisdiction in which the new object will initially reside, and of the Host Object 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 Scheduling Agent. Some classes may allow the creating object to suggest a Magistrate, a Host Object, or both. At any rate, the actual creation of the object is carried out by the Magistrate and Host Object, which are given enough information by the class to allow them to create the new object. This information may take the form of an executable program, the name of an executable, a list of steps to follow, etc. Details about exactly how this will be done have yet to be completely formulated.

4.2.1 Bootstrapping: bringing up core objects

Legion contains a set of core objects and object types that implement the mechanism by which Legion objects are created and activated. For this reason, the creation and activation of this set of objects must be carried out by mechanisms different from those used for normal Legion objects. For instance, all objects are activated on a host by the representative Host Object; however, the Host Object obviously cannot activate itself before it exists. The core objects, including the core Abstract classes (LegionObject, LegionClass, etc.), Host Objects, and Magistrates, are intended to be started from the command line or shell script in the host operating system.

When Host Objects come alive, they contact the existing class object named LegionHost to tell it of their existence. Thus, once they are alive, Host Objects can be located via the same mechanism that is used to locate other types of Legion objects; that is, the class of the object can be consulted. Magistrates also get started "outside" of Legion, and they too contact their class, LegionMagistrate. New Host Objects and Magistrates will be added as the Legion system expands to include new hosts and Jurisdictions. The Abstract class objects are started exactly once—when the Legion system comes alive.

4.3 Object replication

Section 3.4 describes the composition of a Legion Object Address, which is a list of physical addresses (Object Address Elements) along with semantic information that describes how the list is to be utilized. This design enables object replication at the Legion system level. That is, a Legion object—an entity named by a single LOID—can be implemented as a set of processes without changing the application-level semantics for communicating with the object. Replicating an object at the Legion level is a matter of creating an Object Address with multiple physical addresses in its list, assigning the address semantic appropriately, and binding the LOID of the object to this Object Address.

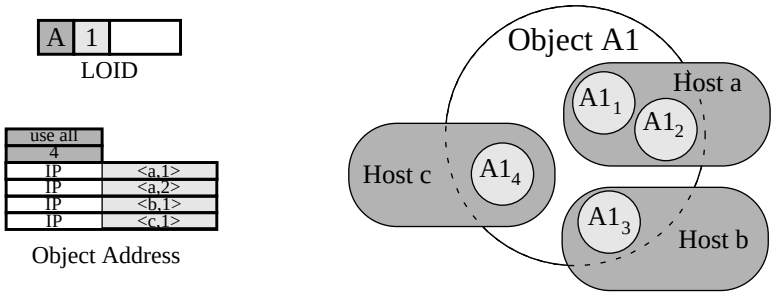

FIGURE 1. An LOID names Legion Object A1, which is implemented as a replicated object consisting of four processes, A1$_1$—A1$_4$, residing at four different physical addresses, <a,1>, <a,2>, <b,1>, and <c,1>. The Object Address for A1 includes each of the address elements.

This design does not preclude the replication of objects at the Legion application level instead of, or in congruence with, replication at the system level. In other words, multiple Legion objects, each with its own LOID, can work together to perform a single logical function, but in this case the management of the "object group" and the semantics of communication with the group is left to the application programmer.

5 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.

5.1 What is “scalability”?

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[7] 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.

5.2 Scalability in Legion

In the distributed systems realm, 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 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 NII 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 inter user-level object communication that occurs 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.

5.2.1 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.

5.2.2 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[9].

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.

6 Conclusions

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, Host Objects, Magistrates, 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. The system scales to millions of sites and trillions of objects.

References

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

2) 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.

3) Adam J. Ferrari, Andrew Grimshaw, *Persistent Object State Management in Legion*, University of Virginia Computer Science Technical Report CS-95-36, in progress.

4) Adele Goldberg, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

5) The Mentat Research Group, *Mentat 2.8 Programming Language Reference Manual*, Department of Computer Science, University of Virginia, 1995.

6) Anh Nguyen-Tuong, Andrew Grimshaw, *[Fault tolerance in Legion]*, University of Virginia Computer Science Technical Report CS-??-??, in progress.

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

8) 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.

9) 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.