

Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems¹

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

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

Abstract

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

This paper was submitted to *IEEE Transactions on Parallel and Distributed Systems* on November 2, 1998.

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

1. Introduction

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

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

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

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

implementation of these services or policies for their use.

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

2. Legion Objectives, Constraints and Philosophy

2.1 Objectives

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

- **Site autonomy:** Legion will be composed of resources owned by many organizations, which properly insist on retaining control over their resources. For each resource the owner must be able to limit or deny use by particular users, specify when it can be used, etc. An important aspect of autonomy is implementation selection. Sites must be able to choose or re-write the implementation of each Legion component to best suit their needs. For example, a site may trust the security mechanisms of one particular implementation over those of another.
- **Extensible core:** Legion must be flexible enough to suit the wide variety of current user demands and capable of evolving to meet unanticipated future needs. Therefore, we feel that mechanism and policy must be realized via replaceable and extensible components, including and especially those of our core system components. This model facilitates development of improved implementations that provide value-added services or site-specific policies, while enabling Legion to adapt over time to a changing hardware and user environment.
- **Scalable architecture:** Because Legion's goal is to construct metasystems with millions of hosts and objects, it must have a scalable software architecture. That is, the system must be fully distributed with no centralized structures or servers.
- **Easy-to-use, seamless computational environment:** Legion must mask the complexity of its hardware environment and simplify the communication and synchronization involved in parallel

processing. Machine boundaries, for example, should be invisible to users.

- **High-performance via parallelism:** We believe that metasytems must support high-performance parallel applications with large degrees of parallelism. Therefore, Legion must keep overhead low and support a wide variety of parallel processing models, including arbitrary combinations of task and data parallelism.
- **Single, persistent object space:** The lack of a single name space for accessing data and resources is one of the most significant obstacles to wide-area distributed and parallel processing. The current multitude of disjoint name spaces greatly impedes developing applications that span sites. All Legion objects must be able to transparently access (subject to security constraints) any other Legion object without regard to location or replication.
- **Security for users and resource owners:** Attempting to patch security on as an afterthought (as some systems are attempting today) is a fundamentally flawed approach. We believe that security must be built firmly into the foundation of a metacomputing system. We also believe that no single security policy is perfect for all users. Therefore, we must provide mechanisms that allow users and resource owners to select policies that fit their security and performance needs and meet their local administrative requirements.
- **Management/exploitation of resource heterogeneity:** Legion must support interoperability between heterogeneous hardware and software platforms. It should also exploit heterogeneity when possible by matching applications to the best suited resources (e.g., vector codes).
- **Multiple language support and interoperability:** Legion must be able to support the integration and interoperability of application components written in a variety of source languages. We feel that interoperability also dictates that we support legacy codes and work with emerging standards such as CORBA [31] and DCE [26], wherever possible.
- **Fault-tolerance:** In a large-scale metasytem, resource failures (hosts, communication links, disks, etc.) will be commonplace. Therefore, the Legion system itself must deal with failures, through system object fault-tolerance and dynamic system reconfiguration, as well as provide mechanisms to support a wide range of user application fault-tolerance needs.

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

2.2 Constraints

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

- **Cannot change host operating systems.** Organizations will not permit their machines to be used if

their operating systems must be replaced. Our experience with Mentat [15] indicates, though, that building a metasystem on top of host operating systems is a viable approach.

- **Cannot change network interface.** Just as we must accommodate existing operating systems, we assume that we cannot change the network resources or the protocols in use.
- **Cannot require Legion to run in privileged mode.** To protect their objects and resources, Legion users and sites will require Legion software to run with the lowest possible privileges.

2.3 Philosophy

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

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

3. Legion Object Model and Key Legion Concepts

Legion is an object-oriented system comprising independent, logically address space disjoint objects, which communicate with one another via method invocation. The fact that Legion is object-oriented does not preclude the use of non-object-oriented languages or non-object-oriented implementations of objects. In fact, Legion supports objects written in traditional procedural languages such as C and Fortran in addition to object-oriented languages such as C++, Java, and the Mentat Programming Language (MPL) [15].¹

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

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

MPL.

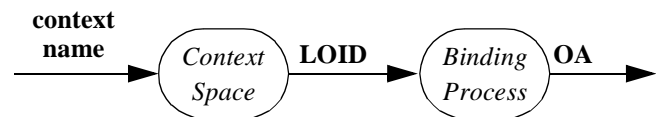
Each Legion object belongs to a class and each class is itself a Legion object. All objects export a common set of *object-mandatory* member functions (such as **deactivate()** and **getInterface()**) that are necessary to implement the core Legion services. Class objects export an additional set of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

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

3.1 Naming and Binding

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

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



LOIDs: Every Legion object is assigned a unique and immutable LOID upon creation. The LOID identifies an object to various services (e.g. method invocation). The basic LOID data structure consists of a sequence of variable length binary string fields. Four of these fields are reserved by the system. The first three play a key role in the LOID-to-object address binding mechanism: Field 0 is the *domain identifier*, used in the dynamic connection of separate Legion systems; Field 1 is the *class identifier*, a bit string uniquely identifying the object's class within its domain; Field 2 is an *instance number* that distinguishes the object from other instances of its class. LOIDs with an instance number field of length zero are

defined to refer to class objects. Field 3 is reserved for security purposes. Specifically, this field contains a public key for encrypted communication with the named object. The format of the LOID is left unspecified beyond these four reserved fields. New LOID types can be constructed to contain additional security information, location hints, and other information in the additional available fields.

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

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

Object States: In a typical Legion system the number of objects may be orders of magnitude larger than the number of processors. Since it is unreasonable to require an active process for every object in the system,² Legion objects are persistent and alternate between two states, *active* or *inert*. When active, an object runs as a process (controlled by a Legion *host object*, Section 5.2) and can be communicated with via its OA. When inert, an object exists only in persistent storage (controlled by a Legion *vault object*, Section 5.3), is described by an *object persistence representation (OPR)*, and is located using an *object persistence address (OPA)*. Throughout their lifetime, objects can be moved between active and inert states.

Each Legion object is associated with its own OPR in which persistent state is stored. Each Legion object implements an internal **saveState()** method that is called to store the object's state into its OPR (prior to becoming inert). Similarly, each object defines an internal **restoreState()** method, which is called immediately after reactivation to recover state from the OPR. The OPA of an inert object is analogous

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

to the OA of an active object and is used to gain direct access to an OPR. Typically, an OPA is a file name or a set of file names, meaningful only to the controlling Legion vault object and to the associated object.

3.2 Attributes

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

3.3 Legion Programming

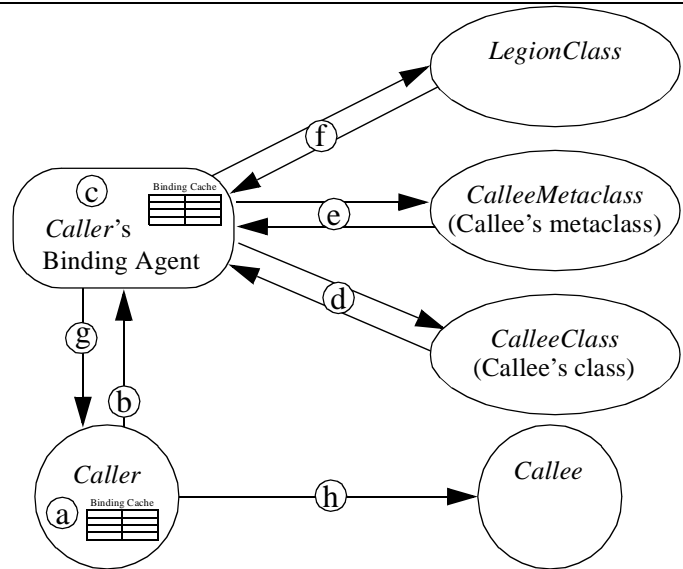
At its lowest level, Legion defines a message format for interobject communication and a set of services and protocols for managing, naming, and manipulating objects. Because Legion is essentially a specification, many implementation strategies are possible for its basic services. We have employed our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

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

4. An Illustrative Example

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

FIGURE 2. Potential steps in the Legion binding and class-of mechanisms—Caller must bind the LOID of Callee to an OA for low-level communication. Caller may already have a cached binding for Callee (a), or may need to consult a binding agent (b). The binding agent may have a cached binding for Callee (c), or may need to consult Callee’s class, *CalleeClass*, for the binding (d). In order to communicate with *CalleeClass*, the binding agent needs a binding for *CalleeClass*. If the binding agent does not have *CalleeClass*’s binding, it may need to consult *CalleeClass*’s metaclass (e). If the binding agent does not know the binding for this metaclass, the process repeats itself. The recursion is guaranteed to terminate at the root of the binding tree, *LegionClass* (f). Eventually, the binding agent returns Callee’s binding (g) and Caller can send messages directly to Callee (h).



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

4.1 Legion Binding Mechanism

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

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

Once the binding agent obtains *CalleeClass*’s LOID, it can request Callee’s binding from

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

4.2 Class-Of Mechanism

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

4.3 Stale Bindings

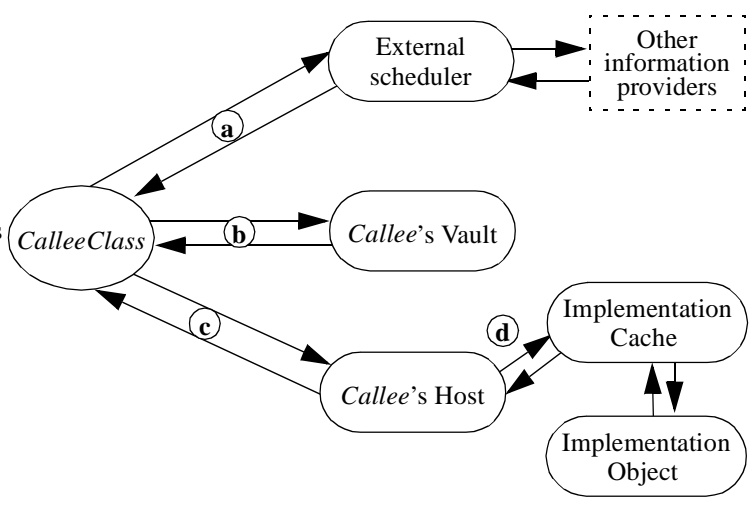
Bindings can become stale as the objects to which they refer deactivate or migrate. For example, if Caller has a binding for Callee, Caller may find that this binding is stale (e.g., by repeated failed attempts to communicate), in which case Caller invokes the *re-binding mechanism*. The re-binding mechanism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

4.4 Object Activation

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

activation mechanism to activate Callee (Figure 3) before returning Callee’s new OA.

FIGURE 3. The Legion object activation mechanism—CalleeClass wishes to activate Callee. First, CalleeClass must decide on which host and vault to place Callee. To do this, CalleeClass may consult an external scheduler (a). After a placement decision is made, CalleeClass must determine the OPA for Callee by consulting Callee’s vault (b). Finally, to activate Callee, CalleeClass sends an activation request to the desired host (c) specifying the object LOID, implementation to use, and OPA. To create a process for Callee, the host must obtain the implementation for Callee. To do this, the host uses a shared implementation cache object (d). After downloading the implementation, the host starts a process for Callee and returns the binding to CalleeClass.



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

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

The **startObject()** method requires the instance’s LOID and OPA, as well as the appropriate *implementation object* LOID for the instance as parameters (implementation objects are Legion objects that

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

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

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

5. Core Object Types

5.1 Classes and Metaclasses

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

Legion classes must implement all of the class-mandatory interface (Figure 4).³ This interface includes **createInstance()**, which creates a new instance of the class and returns the new LOID; **createMultipleInstances()**, which creates several instances of the class at once; and **activateInstance()**, which migrates an existing instance to a new location or re-starts an instance that has become inert. Each of these three functions actually has several versions, allowing the caller to tailor the creation and placement processes. For example, the caller can indicate the host object on which the instance(s) should be placed, or specify the characteristics of acceptable hosts (processor speeds, architectures, etc.). The **addImplementation()** and **removeImplementation()** functions configure which implementations the class object will use for its instances (these functions are typically used by Legion-targeted compilers). The **getBinding()** functions support binding as described in Section 4.1. There are several other functions (not shown in Figure 4) that allow clients to retrieve information about the location and characteristics of the class's instances, such as the instances' interface, their current host, their current state (active or inert), etc.

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

```

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

```

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

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

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

5.2 Host Objects

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

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

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

FIGURE 5. Basic Legion host object interface.

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

We now consider our current default host object, and two possible alternative implementations. Our default design is very simple—it implements a non-restrictive access policy and uses the Unix process management interface (i.e. **fork()**, **exec()**, **kill()**) for starting and stopping objects. Although simple to implement, this design has many flaws. It places a high cost on object activation, executing one process per object and creating new processes on demand for each activated object. This implementation is

severely limited in terms of security—it executes all objects under a single Unix user id, allowing objects from different Legion users to interfere with or examine one another’s state. Below we briefly present some ideas to address these limitations transparently using alternative host object implementations.

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

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

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

5.3 Vault Objects

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

As previously noted, class objects manage the assignment of vaults to instances: when an object is created, its vault is chosen by the object’s class. The selected vault creates a new, empty, OPR for the object,

and supplies the object with its OPA. Similarly, when an object migrates, its class selects a new target vault for its OPR. These vault activities are supported by the basic Legion vault abstract interface (Figure 6). The **createOPR()** method constructs a new empty OPR, associates this OPR with the given LOID, and returns the address of the new OPR to be used by the newly created object. The **getOPRAddress()**

```

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

```

FIGURE 6. The Legion vault object interface.

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

5.4 Implementation Objects

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

```

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

```

FIGURE 7. The Legion implementation object interface

Implementation objects typically contain executable code for a single platform, but may in general contain any information necessary to instantiate an object on a particular host. For example, implementations might contain Java byte code, Perl scripts, or high-level source code that requires compilation by a host. Like all other Legion objects, implementation objects describe themselves by maintaining a set of attributes (Section 3.2). In their attributes implementation objects specify their execution requirements and

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

FIGURE 8. The Legion implementation cache interface

characteristics which may then be exploited during the scheduling process. For example, an implementation object may record the type of executable it contains, its minimum target machine requirements, performance characteristics of the code, etc.

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

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

5.5 Implementation Caches

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

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

5.6 Binding Agents

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

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

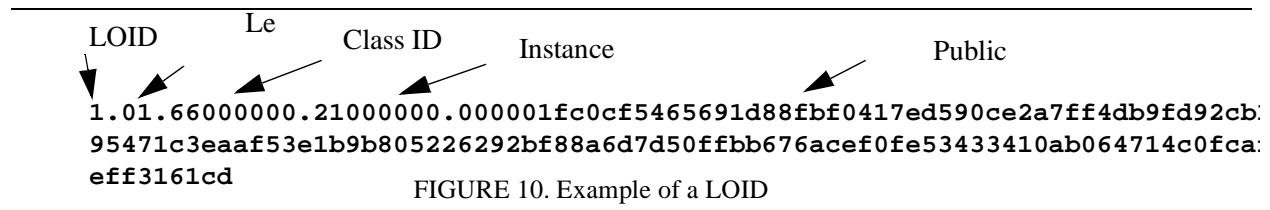
FIGURE 9. The Legion binding agent interface

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

5.7 Context Objects and Context Spaces

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

organizing information we define the interface to a user-level naming service called *context spaces*.



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

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

FIGURE 11. The Legion context object interface

In isolation, a context object may be used to provide a simple, convenient user-level naming service for a user's objects. However, the names inserted into a context can map to other context objects' LOIDs, providing a natural mechanism for constructing a directory service. Connected graphs of context objects are a basic mechanism for organizing information in Legion, and are referred to as context spaces. Every Legion object contains the LOID of a *current working context* and a *root context*,⁴ and library routines are provided for traversing context space to map context paths to LOIDs.

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

6. Related Work

Legion is one of a number of projects developing software to support metacomputing. This section discusses some of the current major metacomputing projects such as Globus [12] and Globe [33]. However, it is worth noting that these projects, Legion, and other metacomputing projects such as MOL

4. Note that there is no notion of a global “root” context for the system. The root is a user-definable starting point for resolving fully qualified context paths.

[32], Ice-T [14], and Harness [8], are all outgrowths of the significant existing work in first-generation network parallel computing systems, such as PVM [13] and MPI [19], and in modern transparent distributed computing systems, such as the Berkeley NOW project [1] and DCE [26].

6.1 Globus

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

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

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

6.2 Globe

The Globe [33] project, which is being developed at Vrije Universiteit, also shares many common goals and attributes with Legion. Both are middleware metasystems that run on top of existing host operating

systems and networks, both support implementation flexibility, both have a single uniform object model and architecture, both use class objects to abstract implementation details, and so on.

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

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

6.3 CORBA

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

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

7. Summary

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

Legion meets these requirements by using replaceable system components that encapsulate both

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

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

References

- [1] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team, "A Case for NOW (Networks of Workstations)" *IEEE Micro*, vol. 15, no. 1, pp. 54-64, February, 1995.
- [2] Brockschmidt, K., "What OLE is really about," Microsoft Corporation, July 1996.
- [3] A. Black, N. Hutchinson, E. Jul, and H. Levy, "Distribution and Abstract Types in Emerald," University of Washington, TR 85-08-05, August, 1985.
- [4] Berman, F., Wolski, R., Figueira, S. Schopf, J., and Shao, G. "Application-Level Scheduling on Distributed Heterogeneous Networks", *Proceedings of Supercomputing '96*, November 1996.
- [5] Cardelli, L. "A language with distributed scope," Digital Equipment Corporation, May 1995.
- [6] Chin, R.S. and Chanson, S.T., "Distributed object-based programming systems," *ACM Computing Surveys*, vol. 23, no. 1, pp. 91-124, March 1991.
- [7] Czajkowski, K., Foster, I., Kesselman, C., Martin, S., Smith, W., and Tuecke, S., "A Resource Management Architecture for Metacomputing Systems," available at:
<http://www.globus.org/globus/papers.htm>
- [8] Dongarra, J., Geist, A., Kohl, J., Papadopoulos, P., Sunderam, V., "HARNESS: Heterogeneous Adaptable Reconfigurable Networked Systems," available at:
<http://www.epm.ornl.gov/harness/>
- [9] Ferrari, A.J., Lewis, M.J., Viles, C.L., Nguyen-Tuong, A., and Grimshaw, A.S., "Implementation of the Legion library," University of Virginia Computer Science Technical Report CS-96-16, November 1996.
- [10] Ferrari, A.J. and Grimshaw, A.S. "Basic Fortran Support In Legion", University of Virginia Department of Computer Science, Technical Report CS-98-11, March 4, 1998.
- [11] Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W., and Tuecke, S., "A Directory Service for Configuring High-Performance Distributed Computations," *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, 1997.

- [12] Foster, I. and Kesselman, C., "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications* (to appear).
- [13] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V.S., PVM: Parallel Virtual Machine, MIT Press, 1994.
- [14] Gray, P., and Sunderam, V.S. "IceT: Distributed Computing and Java," *Concurrency, Practice and Experience*, vol. 9, no. 11, pp. 1161-1168, Nov. 1997.
- [15] Grimshaw, A.S., "Easy-to-use object-oriented parallel processing with Mentat," *IEEE Computer*, pp. 39-51, May 1993.
- [16] Grimshaw, A.S., Weissman, J.B., and Strayer, W.T. "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," to appear *ACM Transactions on Computer Systems*.
- [17] Grimshaw, A.S., Weissman, J.B., West, E.A., and Loyot, E., "Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 21, no. 3, pp. 257-69, June 1994.
- [18] Grimshaw, A.S., Wulf, W.A., the Legion team, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, January 1997.
- [19] Gropp, W., Lusk, E., and Skjellum, A., Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1994.
- [20] Homburg, P., van Doorn, L., van Seen, M., Tanenbaum, A.S., and de Jonge, W., "An object model for flexible distributed systems," Vrije Universiteit.
- [21] IBM, "IBM LoadLeveler: User's Guide (SH26-7226-02)," IBM Publication number ST00-9696, October 1994.
- [22] Karpovich, J., "Support for object placement in wide-area heterogeneous distributed systems," University of Virginia Computer Science Technical Report CS-96-03, January 1996.
- [23] The Legion Research Group, "Legion 1.0 User Manual," University of Virginia Computer Science, November 1997. available from <http://legion.virginia.edu/Documentation.html>.
- [24] Lewis, M., and Grimshaw, A.S., "The core Legion object model," *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.
- [25] Litzkow, M.J., Livny, M., and Mutka, M.W., "Condor—A Hunter of Idle Workstations," *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 1988, pp. 104-111.
- [26] Lockhart, Jr., H.W., *OSF DCE Guide to Developing Distributed Applications*, McGraw-Hill, Inc. New York 1994.
- [27] Message Passing Interface Forum, "MPI: A message-passing interface standard," May 1994.
- [28] Microsoft Corporation, "The Component Object Model specification," Version 0.9, Microsoft Corporation, October 24, 1995.
- [29] Necula, G.C., "Proof-Carrying Code," *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 106-119, Jan 15-17, 1997.

- [30] Nguyen-Tuong, A, Grimshaw, A.S., and Hyett, M., "Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System," *Proceedings of the 15th International Symposium on Reliable and Distributed Systems*, pp. 1-11, October 1996.
- [31] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Revision 2.0, July 1995 (updated July 1996).
- [32] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Rvmke, and J. Simon, "The MOL Project: An Open Extensible Metacomputer," in *Proceedings of the Heterogeneous Computing Workshop, HCW97*, IEEE Computer Society Press, pp. 17-31, 1997.
- [33] van Steen, M., Homburg, P., and Tanenbaum, A.S., "The architectural design of Globe: a wide-area distributed system," Internal report IR-422, Vrije Universiteit, March 1997.
- [34] Viles, C.L., Lewis, M.J., Ferrari, A.J., Nguyen-Tuong, A., and Grimshaw, A.S., "Enabling flexibility in the Legion run-time library," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pp. 265-274. Las Vegas, Nevada, June 30 — July 2, 1997.
- [35] Wulf, W.A., Wang, C., and Kienzle, D., "A new model of security for distributed systems," University of Virginia Computer Science Technical Report CS-95-34, August 1995.
- [36] Yew, P.-C., Tzeng, N.-F., and Lawrie, D.H., "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, Vol. C-36(4), April 1987.