Legion

Flexible Support for Wide-Area Computing[1]

Andrew S. Grimshaw, William A. Wulf[2]

1 The Opportunity

Recent technological changes have made the realization of nationwide, and indeed worldwide, virtual computers possible. A nationwide virtual computer would consist of workstations, personal computers, graphics rendering engines, supercomputers, and possibly non-traditional computing and rendering devices such as TV's, all connected via the NII and an integrated software infrastructure. Such a system, if constructed, may well revolutionize how we work, how we play, and in general, how we interact with one another. The staggering success of the World Wide Web only hints at the potential of such a system.

The technological changes that enable the realization of a worldwide virtual computer are (1) the introduction of gigabit/second fiber-optic backbones, which effectively "shrink" the distance between geographically separate computers, (2) the maturation of parallel compilers that support parallel execution on distributed memory machines, (3) advances in distributed systems software over twenty years which have increased our understanding of how to manage complex distributed environments, (4) the widespread acceptance in industry of the object-oriented paradigm, and (5) advances in cryptography and authentication protocols.

These changes challenge the computer science community to provide a solid integrated software foundation on which to build applications, and with which to unleash the potential of so many diverse, powerful resources. The foundation must at least hide the underlying physical infrastructure from users and from the vast majority of programmers, support access, location, and fault transparency, enable inter-operability of components, support construction of larger integrated components using existing components, and provide a secure environment for both resource owners and users. Further, the services must be provided in the context of tens of thousands to millions of autonomous hosts; thus, the software infrastructure must be scalable or it will fail to meet users' performance expectations. It is this software foundation that will make the collection of hosts and fiber connections a virtual computer. We call such software *metasystems software*.

Whether or not metasystems software is explicitly designed, the nation (and the world) will eventually end up with a metasystem. The reason is simple—both individual and organizational users will be required to deal with increasingly obvious shortcomings of the nationwide computer system. The World Wide Web and gopher show that tools will be developed in an *ad hoc* fashion to paper over the gaps between local systems. The issue is not whether metasystems will be developed; clearly they will. It is, rather, whether they will come about by design in a coherent, seamless system, or painfully and in an *ad hoc* manner by patching together independently developed systems, each with different objectives, design philosophies, and computation models.

2 Legion

Legion is a metasystems software project at the University of Virginia. Our goal is a highly usable, efficient and scalable system based on solid principles. We have been guided by our own work in object-oriented parallel processing, distributed computing, and security, as well as by decades of research in distributed computing systems. When complete, Legion will provide a single, coherent virtual machine that addresses scalability, programming ease, fault tolerance, security, site autonomy, etc. In short, we believe Legion is a conceptual base for the sort of metasystem we seek.

We envision Legion consisting of millions of hosts and trillions of objects co-existing in a loose confederation tied together with high-speed links. The user will have the illusion of a very powerful computer on her desk. She will sit at her terminal and manipulate objects. "Terminals" include workstations, immersive environments such as head-mounted displays or the CAVE™, and portable Personal Digital Assistants. The objects she manipulates will represent data resources such as digital libraries or video streams, applications such as teleconferencing or physical simulations, and physical devices such as cameras, telescopes, and linear accelerators. Naturally the objects being manipulated may be shared with other users, allowing the construction of shared virtual workspaces.

---

1. This work partially supported by DOE grant DE-FG02-96ER25290, DOE contract Sandia #LD-9391, and NSF grant ASC-9201822.

2. Current members of the Legion team include professors James C. French, Paul F. Reynolds, Jr., and Alfred C. Weaver, research scientist Mark Hyett, post-docs Charlie Viles and David Cooper, and graduate students Adam Ferrari, John Karpovich, Darrell Kienzle, Mike Lewis, Anh Nguyen-Tuong, and Chenxi Wang.

It is Legion's responsibility to support the abstraction presented to the user, to transparently schedule application components on processors, manage data migration, caching, transfer, and coercion, detect and manage faults, and ensure that the user's data and physical resources are adequately protected.

In this paper we present an overview of Legion, its vision and objectives, and a brief sketch of the class system and how it supports some of our extensibility objectives. Unfortunately, because of the scope of the project we cannot hope to present the details needed to satisfy the serious reader in just eight pages. A series of technical reports are available at http://www.cs.virginia.edu/~legion. The site includes technical reports on experiences with the prototype system [6], the security model [14], resource management [7], persistence, and a large number of slides that have been used for various presentations. The current implementation and all design documents and interfaces are also available at the web-site.

2.1 Objectives

From our Legion vision we have distilled ten design objectives that are central to the success of the project: site autonomy; an extensible core; scalability; an easy-to-use, seamless computational environment; high performance via parallelism; a single persistent name space; security for users and resource providers; management and exploitation of resource heterogeneity; multi-language support and inter-operability; and fault tolerance.

- *Site autonomy*: Legion will not be a monolithic system. Legion will be composed of diverse resources owned and controlled by an array of different organizations. These organizations, quite properly, insist on having control over their own resources, e.g., specifying how much resource can be used, when it can be used, and who can and cannot use the resource.
- *Extensible core*: We cannot know the future or all of the many and varied needs of users. Therefore, mechanism and policy must be realized via extensible, replaceable, components. This will permit Legion to evolve over time and will allow users to construct their own mechanisms and policies to meet specific needs.
- *Scalable architecture*: Because Legion will consist of thousands of hosts, it must use a scalable software architecture. Thus there must be no centralized structures and that the system must be totally distributed.
- *Easy-to-use, seamless computational environment*: Legion must mask the complexity of the hardware environment and of communication and synchronization of parallel processing. Machine boundaries should be invisible to users. As much as possible, compilers, acting in concert with run-time facilities, must manage the environment for the user.
- *High performance via parallelism*: Legion must support easy-to-use parallel processing with large degrees of parallelism. This includes task and data parallelism and their combinations.
- *Single, persistent name space*: One of the most significant obstacles to wide area parallel processing is the lack of a single name space for file and data access. The existing multitude of disjoint name spaces makes writing applications that span sites extremely difficult.
- *Security for users and resource owners*: Because we cannot replace existing host operating systems, we cannot significantly strengthen existing operating system protection and security mechanisms. However, we must ensure that existing mechanisms are not weakened by Legion. Further, we must allow users to manage their own security needs; Legion should not define the security policy or require a "trusted" Legion.
- *Management and exploitation of resource heterogeneity*: Clearly, Legion must support inter-operability between heterogeneous hardware and software components. In addition, some architectures are better than others at executing particular kinds of code, e.g., vectorizable codes. These affinities, and the costs of exploiting them, must be factored into scheduling decisions and policies.
- *Multiple language support and inter-operability*: Legion applications will be written in a variety of languages. It must be possible to integrate heterogeneous source language application components in much the same manner that heterogeneous architectures are integrated. Inter-operability also means that we must be able to support legacy codes written before the advent of Legion.
- *Fault-tolerance*: In a system as large as Legion, it is certain that at any given instant, several hosts, communication links, and disks will have failed. Thus, dealing with failure and dynamic re-configuration is a necessity. Current parallel processing systems do not address fault-tolerance.

In addition to the goals above, two constraints restrict our design—we cannot replace host operating systems, and we cannot legislate changes to the interconnection network.

- *We cannot replace host operating systems*: Organizations will not permit their machines to be used if their operating systems must be replaced. Operating system replacement would require them to rewrite many of their applications, retrain many of their users, and possibly make them incompatible with other systems in their organization. Our experience with Mentat [5] indicates that it is sufficient to layer a system on top of an existing host operating system.
- *We cannot legislate changes to the interconnection network*: We must initially assume that the network resources, and the protocols in use, are a given. Much as we must accommodate operating system heterogeneity, we must live with the available network resources. However, we can layer better protocols over existing ones, and we can state that performance for a particular application on a particular network will be poor unless the protocol is changed.
- *We cannot require that Legion run as "root" (or the equivalent)*. Indeed, quite the contrary — to protect themselves, most Legion users will want it to run with the least possible privileges. Of course we do not prohibit Legion implementations that require root privilege.

In addition to the purely technical issues, there are also political, sociological, and economic ones. These include encouraging the participation of resource-rich centers and discouraging the human tendency to free-ride.

The common framework that enables a coherent solution to these problems is object-orientation. In Legion all components of interest to the system are objects, and all objects are instances of defined classes. Thus users, data, applications and even class definitions are objects. Use of an object-oriented foundation, including the paradigm's encapsulation and inheritance properties, will make accessible a variety of the benefits often associated with the paradigm, including, software reuse, fault containment, and reduction in complexity. The need for the paradigm is particularly acute in a system as large and complex as Legion.

Objects, written in either an object-oriented language or other languages such as HPF Fortran, will encapsulate their implementation, data structures, and parallelism, and will interact with other objects via well-defined interfaces. In addition they may have associated inherited timing, fault, persistence, priority, and protection characteristics. Naturally these characteristics may be overloaded to provide different functionality on a class by class basis. Similarly, a class may have multiple implementations with the same interface.

2.2 System philosophy

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

Neither the "kind" nor the "level" of functionality are linearly ordered, but a simplistic model is that of a multidimensional space. The needs of users will dictate where they need to be and/or can afford to be in this space; we, the designers of the supporting conceptual system have no way of knowing what those needs are, or what they will evolve to be in the future. Indeed, if we were to dictate a system-wide "solution" to almost any of the issues raised in our list of objectives we would preclude large classes of potential users and uses.

Consider security with respect to both kind and level of functionality. Some users are mostly concerned with privacy, while others are more concerned with the integrity of their data — both banks and hospitals are in the later category for example. Some users are content with password authentication, while others might feel the need for stronger user identification — signature analysis, fingerprint verification, etc. Both of these are examples of differences in the *kind* of security functionality. The size of cryptographic key, on the other hand, is an issue of the degree, or level, of security. Without changing the basic nature of the security provided, users can get a greater degree of security by paying the higher cost of using a longer key or a stronger algorithm.

In the Legion approach, rather than providing a fixed security mechanism, with the result that no one is completely satisfied, users may choose their own trade-offs by implementing their own policies or by using existing policies via inheritance [14]. Some users may select a policy that requires every method invocation to have all of its parameters encrypted, that the caller be separately authenticated, and that the user on whose behalf the call is being made be fully authenticated as well. Such a policy will be expensive (CPU, bandwidth, time). Alternatively, an application that requires low overhead cannot afford such a policy and should not be forced to use it. Such an application could instead choose a light-weight policy that simply checks if the caller is its parent (creator) without any authentication or encryption, or perhaps does not check anything at all.

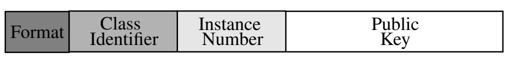
The philosophy has been extended into the system itself. The Legion object model specifies the composition and functionality of Legion's core objects—those objects that cooperate to create, locate, manage, and remove objects from the Legion system. Legion specifies the functionality, not the implementation, of the system's core objects. Therefore, the 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.

3 Legion Objects

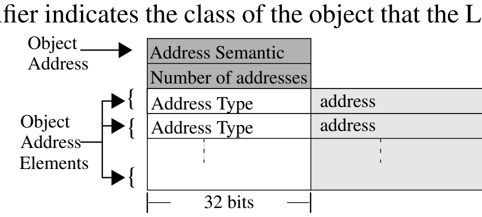
Legion objects are independent, logically address space disjoint objects that communicate with one another via method invocation and return values. Method calls are non-blocking and may be accepted in any order by the called object. Parameters may be *in*, *out*, or *in/out*. Each method has a signature that describes the parameters and return value, if any, of the method. The complete set of method signatures for an object fully describes that object's interface, which is determined by its class. Legion class interfaces can be described in an Interface Description Language (IDL). Initially, two different IDL's will be supported by Legion: the CORBA IDL [2], and the Mentat Programming Language (MPL) [5].

All Legion objects export a common set of *object-mandatory* member functions, including may_I(), save_state(), and restore_state(). These member functions relate to security [14], fault-tolerance, and persistence. These functions are invoked by the Legion run-time system as needed. Class objects export an additional set of *class-mandatory* member functions, including create(), derive(), and get_binding().

Each Legion object has a name, which we call a *LOID* (Legion Object IDentifier). A LOID has at least four different fields—a domain, a class identifier, an instance number, and a public key extension. the format field makes up the first 32 bits of the LOID, and although the other three fields are expected to be contained in all LOID's, their size, format and use can vary with different formats. The class identifier indicates the class of the object that the LOID names.



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



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

The instance number field can be used by classes to provide a unique LOID to each instance of the class, but this use is not mandated by Legion—a class object can assign instance numbers in any way it chooses. the public key extension field allows the entire LOID to be a public key for the object, and is used for security purposes [13,14]. Our initial implementation will support fixed size LOID's that contain 64-bit class identifiers, 64-bit instance numbers, and 128-bit public key extensions. Other LOID formats will emerge.

Legion objects may (and usually will) have persistent state. Because there will be far more objects in Legion than processes to hold them an object is not guaranteed to always be in-memory. A Legion object may be either *active* or *inert*. All Legion objects have the object-mandatory member functions save_state and restore_state that are used to move objects between active and inert states.

When a Legion object is active it is in-memory somewhere, has at least one thread of control, can accept and process method invocations, and has an *object address* (OA). An object address is a list of *Object Address Elements*, along with semantic information that describes how to utilize the list. An object address element contains two basic parts—a 32-bit address type field, and the address itself. the address type field indicates the type of address (e.g. IP, XTP, etc.) that is contained in the address specific field, whose size and format will vary depending on the address type. the first (and probably the most common) type of object address will consist of a single object address element that comprises a 32-bit IP address and a 16-bit port number.

When an object is inert, it exists on persistent storage that is controlled by a LegionVault, is described by an *Object Persistent Representation (OPR)*, and it can be located using an *Object Persistent Address (OPA)*. Throughout their lifetime, objects can be moved between active and inert states many times.

Before discussing of the role of classes we must digress for a moment to introduce two core Legion classes, LegionHost, and LegionVault.

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

LegionVaults hold OPR's for other Legion objects. In response to save_state() and restore_state() member functions, an object reads and writes its state from and to its OPR, which typically exists on persistent storage. A Vault has access to the OPR's it holds via mechanisms outside of Legion (e.g., a shared Unix file or directory). When an object is created, a Vault for the object is chosen by the object's class, and the Vault supplies an empty OPR. The object's save_state() method can then write its state into this OPR before being deactivated, and its restore_state() method can read it out upon being reactivated.

4 Legion Classes

One of the distinguishing characteristics of Legion is the role of classes. As in many other systems, all objects are instances of a class. Legion classes are also objects - and thus instances of a class. The power of our object model comes from the important role of classes. Much of what is usually considered system-level responsibility is delegated to user-level class objects. For instance, classes are responsible for creating and locating their instances and subclasses, and for selecting appropriate security and object placement policies. The core Legion objects simply provide mechanisms for user-level classes to implement the policies and algorithms that they choose.

Legion class objects are responsible for all aspects of their instances' implementations. This includes knowing the state (active or inert) of all instances, mapping instance LOID's to OPA's and OA's, choosing which LegionHosts and LegionVaults to trust, and choosing where to place objects and their OPR's. Class objects are themselves instances of meta classes that define their implementations.

Each class object at a minimum exports class-mandatory member functions to create new instances (create()) and subclasses (derive()), to delete instances and subclasses (delete()), and to find instances and subclasses (get_binding()). Class objects may export other functions as desired. In particular some class objects have functions that allow their instances to reflexively modify scheduling and replication policies.

To perform its functions each class object must *logically* maintain a table indexed by LOID whose entries contain fields for an LOID, an Object Address, a Placement Mapper, a Current Vault Set, a Candidate Vault Set, and a Candidate Host Set. In practice, the class object may employ other Legion objects, such as database servers, to maintain some or all of the information.

- *Object Address*: The Object Address field contains either the Object Address of the object (if the object is currently Active and the class knows its Object Address), or NIL (if the object is currently Inert). This field is used to respond to get_binding() requests from Binding Agents and other Legion objects.
- *Placement Mapper*: The Placement Mapper field contains the LOID of the object that is responsible for assigning the object entered in the table to a Legion Host when it is about to be activated. Thus, scheduling is a user-level function.
- *Current Vault Set*: The Current Vault Set field contains a list of Vaults that currently have Object Persistent Representations for an object. Typically, only one Legion Vault will have a copy of the Object Persistent Representation of an object.
- *Candidate Vault Set:* The Candidate Vault Set field indicates the Vaults that may be given responsibility for the object, i.e., the Vaults the class trusts.
- *Candidate Host Set*: The set of LegionHosts on which the instance is permitted to execute.

As the fields above suggest, classes are involved in all aspects of what are usually considered system-level responsibilities, naming, binding, protection, and replication.

Object creation and naming: Each class object is responsible for assigning LOID's to its instances and subclasses upon their creation. For its instances the class object can construct the LOID completely locally; it assigns the class identifier portion to match its own class identifier, and uses the instance number field in any way it sees fit. Usually this will be a sequence number to guarantee that all LOID's are unique. However that need not be the case. Different sequence number ranges could refer to different implementations, or the class object may re-use sequence numbers and multiplex multiple create requests onto the same object.

Binding: Class objects are ultimately responsible for binding the LOID's of their instances to Object Addresses. Normally this will imply a lookup to find the OA if the instance is active. If the instance is not active then it must be scheduled onto a LegionHost that can access a LegionVault that holds a copy of the OPR. Typically the process named by the Object Address will contain one Legion object. Other bindings are possible. Consider three possible other bindings.

1) The class may multiplex multiple instances into a single address space - reducing the need for a large number of processes and reducing the overhead of inter object communication between instances in the same address space. This type of binding will be used frequently for dataparallel objects.

2) The class may hand out the Object Address of a proxy object instead of the actual address. All invocations on the instance must first pass through the proxy. This prevents others (besides the proxy) from knowing the location of the instance, and allows the proxy to enforce security or fault-tolerance policies without the instance's knowledge.

3) The class may bind one LOID to multiple OAs, e.g., as a mechanism for distributing the load. Thus instance number 1 might be given out in response to create requests, or as a generic instance. Each binding request to instance 1 would be mapped to a different instance in a round-robin, or other, fashion.

Scheduling: Class objects control where their instances execute. They can either directly make that decision, or allow a user level scheduling object to make the decision on their behalf [7].

The ability to define the behavior of class objects provides the programmer with a flexible implementation tool. Requiring all programmers to define their own class object implementations would impose a heavy burden on the majority of programmers. Therefore several base implementations will be developed at Virginia that can either be used directly, or can be selectively modified for local circumstances.

5 The Legion Stack

Another mechanism for achieving extensibility and flexibility is the ability to modify the implementation of the server-side code. First some background. Recall that communication with a Legion object is realized via member function invocation. An obvious implementation involves a server loop at the object (callee) receiving messages in order, decoding the function identifier, invoking the function, and then receiving another message. Other alternatives are imaginable, indeed required, to support some programming language abstractions. For example we might want to allow the programmer to select which functions to accept via a select/accept, or we may want to support path expressions, or perhaps an active messages interface.

To achieve this flexibility we use an event driven model where the activities to be carried out on each event type are under the control of the programmer. The "programmer" in this case is usually the language interface implementor, but in principle the application programmer as well. When an event occurs, e.g., a message arrival event, the runtime libraries execute the event list (usually in a new thread) associated with the event. An event-list is a list of integer functions that will be called in order on an event structure until the end of the list is reached or one of the functions returns a 0. During the course of execution an event may spawn other events which are then handled asynchronously.

For example, to realize a simple server structure where the object thread polls a queue of matched[1], ready to run, invocations we would construct an event list such as:

```
message-arrival event_list
   add message to un-matched database,
       return 1 if all parameters matched, point to completed work_unit
       else return 0 (and stop processing this event)
   add matched work_unit to the ready queue
```

To provide a select/accept capability we define a work_unit database that can be queried for different ready function invocations and change the event list to:

```
message-arrival event_list
   add message to un-matched database,
       return 1 if all parameters matched, point to completed work_unit
       else return 0 (and stop processing this event)
   add matched work_unit to the work_unit database
```

To realize an active-messages like semantics (note that a server loop is no longer required)

```
message-arrival event_list
   add message to un-matched database,
       return 1 if all parameters matched, point to completed work_unit
       else return 0 (and stop processing this event)
   lookup the function ID in a jump table and call the function
```

1. We say "matched" invocations because the different parameters to an invocation may arrive in separate messages, and may require matching and assembly.

To add data decryption

```
message-arrival event_list
   decrypt using senders public key (found in its LOID)
   add message to un-matched database,
       return 1 if all parameters matched, point to completed work_unit
       else return 0 (and stop processing this event)
   lookup the function ID in a jump table and call the function
```

To catch special messages used by a special implementation of some feature

```
message-arrival event_list
   call "my_filter". If it is a special message set up a new event
       to handle it and return 0, else return 1
   ...
```

To support multiple Legion objects per address space

```
message-arrival event_list
   decrypt using senders public key (found in its LOID)
   call de-multiplexer, forwards messages to specific objects by
       setting up the event structure to point to the appropriate
       objects database
   ...
```

The list goes on and on. The point is that the mechanism can be used to support a wide range of semantics.

6 Related work

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

Related efforts such as OSF/DCE [9] and CORBA [2] are rapidly becoming industry standards. Legion and DCE share many of the same objectives, and draw upon the same heterogeneous distributed computing literature for inspiration. Consequently, both projects use many of the same techniques, e.g., an object-based architecture and model, IDL's to describe object behavior, and wrappers to support legacy code. However, Legion and DCE differ in several fundamental ways. First, DCE does not target high-performance computing; its underlying computation model is based on blocking RPC between objects. Further, DCE does not support parallel computing; instead, the emphasis is on client-server based distributed computing. Legion, on the other hand, is based upon a parallel computing model, and one of our primary objectives is high performance via parallel computation. Another important difference is that Legion specifies very little about the implementation. Users and resources owners are permitted—even encouraged—to provide their own implementations of "system" services. Our core model is completely extensible and provides choice at every opportunity—from security to scheduling to fault-tolerance. Similarly, CORBA[2] defines an object-oriented model for accessing distributed objects. CORBA includes an Interface Description Language, and a specification for the functionality of run-time systems that enable access to objects (ORB's). But like DCE, CORBA is based on a client-server model rather than a parallel computing model, and less emphasis is placed on issues such as object persistence, placement, and migration.

Other projects share many of the same objectives but not the scope of Legion. Nexus[4] provides communication and resource management facilities for parallel language compilers. Castle[3] is a set of related projects that aims to support scientific applications, parallel languages and libraries, and low-level communications issues. The NOW[1] project provides a somewhat more unified strategy for managing networks of workstations, but is intended to scale only to hundreds of machines instead of millions. Globe[12] is an architecture for supporting wide area distributed systems, but does not yet seem to address important issues such as security and site autonomy.

In its intended application for distributed collaboration and information systems, Legion might be compared to the World Wide Web. In particular, the object-oriented, secure, platform independent remote execution model afforded by the Java language[11] has added more Legion-like capabilities to the Web. The most significant differences between Java and Legion lie in Java's lack of a remote method invocation facility, lack of support for distributed memory parallelism, and its interpreted nature, which even in the presence of "just-in-time" compilation leads to significantly lower performance than can be achieved using compilation. Furthermore, the security and object placement models provided by Java are rigid and are a poor fit for many applications.

7 Summary

Legion is an ambitious middleware project that will provide a solid, integrated, conceptual foundation on which to build applications. One could argue that Legion is perhaps too ambitious, that there are just too many different complex issues to address. Tackling too many different issues is certainly a risk. On the other hand, eventually there will be Legion-like metasystem software; it is a necessary condition for a large scale digital society. The real issue is whether it will come about by design, in an organized and coherent fashion, or by pasting together different solutions. Legion's object model was *designed* for the intended environment and for extensibility. We feel that these attributes will permit Legion to readily adapt to an ever changing world.

Legion, as defined by our objectives, is not yet a reality. We have a prototype that demonstrates the feasibility of constructing a wide-area system and permits application and tool development to occur concurrently with system implementation. It is not designed to evolve directly into a complete Legion implementation. The prototype has already been used in wide-area computations. During Supercomputing '95 in San Diego we demonstrated Legion applications running over four I-way connected sites using eight parallel machines from three vendors.

In March of 1996 we began our implementation of the core Legion object model. Unlike the existing prototype the new implementation incorporates mechanism for security, fault-tolerance, application directed scheduling, autonomy, scalable binding, etc. This "full blown" implementation re-uses many components of the prototype (e.g., the compiler and debugger) but for the most part is being written from the ground up. We expect to have a usable documented system available for public use in mid 1997. The system, and sources, will be publicly available.

The new implementation will be used in several projects. Legion is the metasystem software being used in the DARPA funded Distributed Object Computation Testbed project, a collaborative effort between SDSC, NCSA, SAIC, and the University of Virginia. We are also collaborating with researchers at Sandia National Labs and Los Alamos National Labs on Legion support for the ASCI (Accelerated Strategic Computing Initiative), as well as with several sites in the NSF Partnerships for Advanced Computational Infrastructure competition.

8 References

[1] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team, "A Case for NOW (Networks of Workstations)," to appear in *IEEE Micro*.

[2] Ron Ben-Naten, *CORBA: A Guide to the Common Object Request Broker Architecture*, McGraw-Hill, 1995.

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

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

[5] A. S. Grimshaw, A. J. Ferrari, and E. A. West, "Mentat" *Parallel Programming Using C++*, Editor: Greg Wilson, MIT Press, 1996.

[6] A.S. Grimshaw, A. Nguyen-Tuong, and W.A. Wulf, "Campus-Wide Computing: Early Results Using Legion at the University of Virginia," University of Virginia Computer Science Technical Report CS-95-19, March 1995.

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

[8] M.J. Lewis, A.S. Grimshaw, "The Core Legion Object Model," *Proceedings of High Performance Distributed Computing Conference*, Syracuse, NY, August 1995.

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

[10] S. Mullender ed., *Distributed Systems*, ACM Press, 1989.

[11] Sun Microsystems, "The Java Language Specification," Version 1.0 Beta,Oct. 30, 1995

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

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

[14] W.A. Wulf, C. Wang, D. Kienzle, *A New Model of Security for Distributed Systems*, University of Virginia Computer Science Technical Report CS-95-34, August 1995.