Legion — A View From 50,000 Feet

Andrew S. Grimshaw, Wm. A. Wulf
and the whole Legion team
Department of Computer Science, University of Virginia
{grimshaw|wulf}@cs.virginia.edu

Abstract[1]

The coming of giga-bit networks makes possible the realization of a single nationwide virtual computer comprised of a variety of geographically distributed high-performance machines and workstations. To realize the potential that the physical infrastructure provides, software must be developed that is easy to use, supports large degrees of parallelism in applications code, and manages the complexity of the underlying physical system for the user. Legion is a metasystem project at the University of Virginia designed to provide users with a transparent interface to the available resources, both at the programming interface level as well as at the user level. Legion addresses issues such as parallelism, fault-tolerance, security, autonomy, heterogeneity, resource management, and access transparency in a multi-language environment. In this paper we present a high-level overview of Legion, its vision, objectives, a brief sketch of how some of those objectives will be met, and the current status of the project.[2]

1 The Opportunity

The dramatic increase in ubiquitously available network bandwidth will qualitatively change how the world computes, communicates, and collaborates. The rapid expansion of the world-wide web, and the changes that it has wrought are just the beginning. As high bandwidth connections become available they "shrink" distance and change our modes of computation, storage and interaction. Inevitably, users will operate in a wide-area environment that transparently consists of workstations, personal computers, graphics rendering engines, supercomputers, and non-traditional devices: e.g., TVs, toasters, etc. The relative physical location of the users and their resources will become increasingly irrelevant.

The realization of such an environment, called a "metasystem", is not without problems. Today's experimental high speed networks such as the vBNS and the I-way preview both the promise and pitfalls of such technology. There are many difficulties: few approaches scale to millions of machines, the tools for writing applications are primitive, faults abound and mechanisms to handle them are not available, issues of security are treated in a patchwork manner, and site autonomy — controlling ones own resources while still playing in the global infrastructure — is often not addressed.

As usual, the fundamental difficulty is software — specifically, we believe the problem is an inadequate *conceptual model*. In the face of the onrush of hardware, the community has tried to stretch an existing paradigm, interacting autonomous hosts, into a regime for which it was not designed. The result is a collection of partial solutions — some good in isolation, but lacking coherence and scalability — that make the development of even a single wide-area application demanding at best.

Thus, the challenge to the computer science community is to provide a solid, integrated, foundation on which to build applications that unleash the potential of so many diverse resources. The foundation must 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, provide a secure environment for both resource owners and users, and it must scale to millions of autonomous hosts.

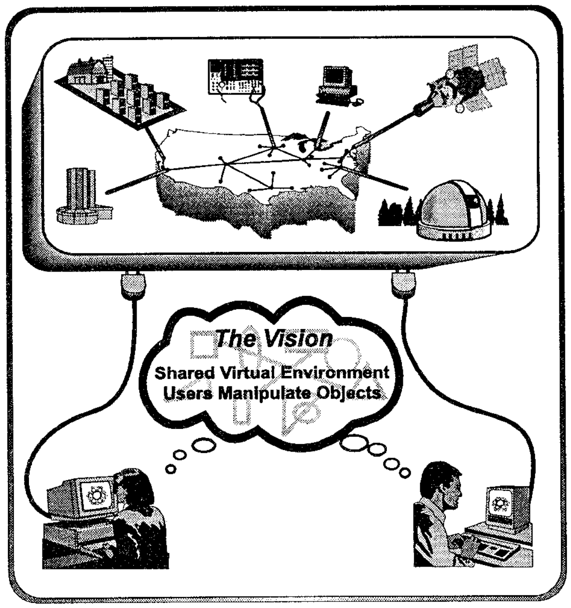
The technology to meet this challenge largely exists: (1) parallel compilers that support execution on distributed memory machines, (2) advances in distributed systems software that manage complex distributed environments, (3) the widespread acceptance of the object-oriented paradigm because of its encapsulation and reuse properties, and (4) advances in cryptography and cryptographic protocols.

Legion is a metasystems software project at the University of Virginia. Begun in the Fall of 1993, our goal is a highly usable, efficient and scalable system based on

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, and graduate students Adam Ferrari, John Karpovich, Darrell Kienzle, Anh Nguyen-Tuong, and Chenxi Wang.

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 each of the issues raised earlier: 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.

Our vision of Legion is of a system 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. We use terminal in its most liberal sense. Terminal could mean anything, a workstation, an immersive environment such as a head-mounted display or CAVE^{TM}, or a portable Personal Digital Assistant. 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, or linear accelerators. Naturally the objects being manipulated may be shared with other users – allowing the construction of shared virtual workspaces.



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 a high-level overview of Legion, its vision, objectives, brief sketches of how some of those objectives will be met, and the current status of the project. Unfortunately, because of the scope of the project we cannot hope to present the details needed to satisfy the serious reader in just twelve pages. There are several sources for more information. First, there is a paper in these proceedings by Mike Lewis on the core object model. That paper describes the object model in much more detail. There are also a series of technical reports available on-line off of the Legion home page, http://www.cs.virginia.edu/~legion. This includes technical reports on experiences with the prototype system [10], the security model [24], resource management [13], persistence, and a large number of slides that have been used for various presentations. The current implementation as well as all design documents and interfaces are also available at the web-site.

We begin with a discussion of project objectives. We next present the object foundation and the philosophical themes that drive all of our design decisions. Next we take a look at how we intend to achieve five of our objectives. We finish with discussions of the project status and what the future holds.

2 Project Objectives

To realize the Legion vision is not a trivial matter. 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 object space; security for both users and resource providers; resource management and exploitation of resource heterogeneity; multi-language support and inter-operability; and fault tolerance.

- *Site autonomy*: Legion will not be a monolithic system. It will be composed of resources owned and controlled by an array of organizations. There is simply no way that thousands of organizations and millions of users will subject themselves to the dictates of a "big brother" centralized control mechanism or subject their resources and data to external management. Organizations, quite properly, will 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.

  Another aspect of site autonomy is autonomy of implementation. Sites must be able to choose which implementations of Legion components to use, either because they "trust" one implementation over another, for performance, or for whatever reason they may have to choose one implementation over another.

- *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.

  Further, consistent with our site autonomy objective, the core system components themselves must be extensible and replaceable. This will allow third party (or site local) implementations which provide value added services to be developed and used.
- *Scalable architecture*: Because Legion will consist of millions of hosts, it must have a scalable software architecture; there must be no centralized structures or servers — the system must be 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 — for example, 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. If Legion is not transparent and easy to use then it will provide little benefit over the *status quo* and will not be used.

  Tempering our transparency objective is the knowledge that there are "power users" with demanding applications that will require, and demand, the capability to make low-level decisions and to interface with low-level system mechanism. Therefore we must accommodate *both* end users that just want to get their work done and not worry about the details, and power users who are compelled to tune their applications.
- *High performance via parallelism*: Legion must support easy-to-use parallel processing with large degrees of parallelism. This includes task and data parallelism and their arbitrary combinations.

  We do not mean though that all applications will be parallel — Legion will necessarily best support relatively coarse-grain applications. Our high-performance via parallelism objective should not be misinterpreted to mean that we think a single huge application will ever use all of the computers in the country. Most parallel applications will use only a small subset of the total resource pool at any time.
- *Single, persistent object space*: One of the most significant obstacles to wide area parallel processing is the lack of a single name space for data and resource access. The existing multitude of disjoint name spaces makes writing applications that span sites extremely difficult. Any Legion object should 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*: We believe very firmly that security must be built firmly into the core from the very beginning. To try to patch security on as an afterthought, as is being attempted today in many contexts, is fundamentally flawed. We also believe that there is no one security policy that is perfect for all users.

  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. Therefore, we must provide mechanism for users to select policies that fit their 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 applications, e.g., vectorizable codes. These affinities, and the costs of exploiting them, must be factored into scheduling decisions and policies.
- *Multiple language support and 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 as well as work with emerging standards such as CORBA [2] and DCE [16].
- *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 — both for Legion itself, and for applications.

2.1 Constraints

In addition to these goals, several constraints restrict our design—for example:

- *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 [9] 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 — it may provide some additional benefit and be acceptable to some sites.

3 Legion's Object Foundation

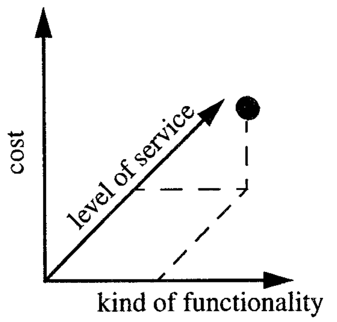
The common framework that enables a coherent solution to these problems is object-orientation. In Legion all components of interest to the system are objects, and all objects are instances of defined classes. Thus users, data, applications and even class definitions are objects. Use of an object-oriented 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 also have associated inherited timing, fault, persistence, priority, and protection characteristics. Naturally these may be overloaded to provide different functionality on a class by class basis. Similarly, a class may have multiple implementations with the same interface.

3.1 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.* We must design Legion to allow users and class implementors the greatest 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 multi-dimensional 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, or whatever. 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 [24]. Some users may require 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.

Next consider consistency semantics in a distributed file system. To achieve good performance it is often desirable to make copies of all or portions of a file. If updates to the file are permitted the different copies may begin to diverge. There are many ways to attack this problem, don't replicate writable files, use a cache invalidate protocol, use lazy updates to a master copy, and so on. Each has an associated cost and semantics. Some applications don't require all copies to be the same, others require a strict "reads deliver the last value written" semantics, others know that the file is read only so that consistency protocols are a waste of time, while others may need different semantics for the file in different regions of the application. Independent of the file semantics, some users may need automatic backup and archiving frequently, while others may not. The point is that the system should not make such decisions for users, they should select the kind and level of service they require.

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.

The object model provides a natural way to achieve this kind of flexibility. Files, for example, are not part of Legion itself. Anyone may define a new class whose general semantics we would recognize as those of a file, but whose specifics match the particular semantics match that user's needs. We (the Legion team) need to provide an initial collection of file classes that reflect the most common needs — but we do not have to anticipate all possible future requirements.

4 Achieving Our Objectives

Below we briefly sketch how we intend to achieve five of our ten objectives. See our home page and technical reports for more information, particularly on security.

4.1 Multiple languages and inter-operability

While we are committed to the object-oriented paradigm we recognize that Legion will need to support applications written in a variety of languages in order to support existing legacy code, permit organizations to use familiar languages (C, Fortran), and support parallel processing languages. We intend to provide multilanguage support, and inter-operability between user objects written in different languages in three ways, by generating object "wrappers" for codes written in languages such as Fortran, ADA, and C; by exporting the Legion run-time system interface and retargeting existing compilers and by a combination of the two.

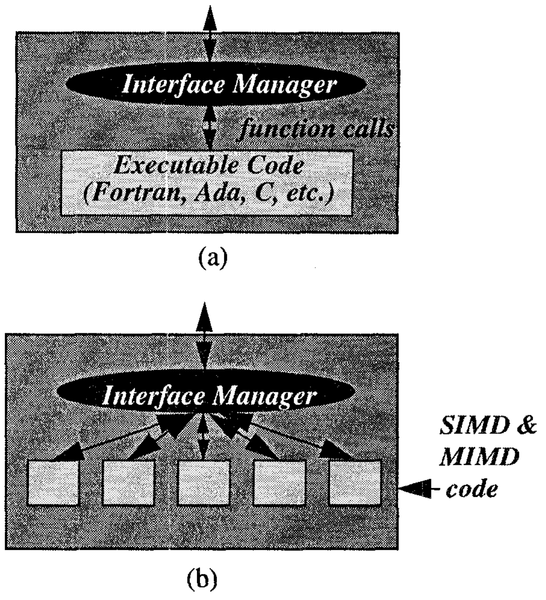


Figure 1 Object wrappers applied to (a) sequential codes and (b) parallel codes.

Object wrappers: Legacy codes and other language codes can be incorporated into Legion applications by encapsulating them in an object wrapper as shown in Figure 1-a. Object wrappers can be either hand generated using the Mentat programming language or by using an IDL and an interface compiler. In the IDL case the programmer provides a "class definition" of the object that lists the functions, the type and number of parameters, whether the object has persistent state, etc., and a compiler generates the interface. This is a common technique and is used in the OMG ORB [2]. We intend to support CORBA and possibly OSF DCE [16]. Additional IDL's can be readily supported because class objects have the capability to export multiple interfaces or views of a class.

Exposing the Legion run-time system: Our second method of supporting other languages is to export the Legion interface to third party compiler writers and tool builders, allowing them to readily port their products.

This is particularly important for parallel systems (see section 4.2).

Finally, our third method is to combine object wrappers and retargeting compilers for other parallel languages, permitting the use of parallel applications developed under other systems as components of meta-applications. Thus, the components would be distributed across Legion, as opposed to across a single MPP.

4.2 High performance via parallelism

High performance for applications will be achieved in Legion in one of two ways, resource selection (scheduling) of *jobs* using resource availability information and resource affinity information, and via parallel computation. The former method exploits resources available throughout the system and is an extension of existing job-based schedulers in wide-spread use such as Condor [3], DQS [7], and LoadLeveler [12].

The second mechanism, parallel execution of application components, is one of the major thrust areas for the project. Legion will support task and data parallelism, as well as combined task and data parallel applications, that are written in a variety of languages.

An application will not become a parallel application simply by executing in Legion, it must first be parallelized. We intend to support parallel execution in Legion using four mechanisms, wrapping existing parallel codes in Legion wrappers (see 4.1), supporting parallel method invocation and object management, exposing the Legion run-time interface to parallel language compilers and toolkit builders, and by supporting popular message passing API's such as PVM [22] and MPI [7].

Naturally not all applications will benefit from parallel execution under Legion. As is true in any parallel processing system there will be applications that either do not parallelize well, or are too fine-grain for the environment. Applications that will perform well under Legion will be latency tolerant and relatively large-grain.

Object wrappers for parallel components: In addition to encapsulating sequential codes, object wrappers can be used to encapsulate parallel programs as components, such as a C* program for the TMC CM-2 or CM-5. This permits the use of optimized parallel applications as components in larger meta-applications, such as multi-disciplinary optimization (MDO) problems (Figure 1-b).

Parallel method invocation: We will support the Mentat programming language (MPL) with Legion from the very beginning. MPL is an extension of C++ designed to support the parallel execution of applications. Several real applications have been developed using MPL. These applications will be used to test the efficacy of our approach from a very early stage.

Exposing the Legion run-time: Legion will be an open system in order to encourage third party software development. We will expose the Legion run-time to compiler writers and toolkit builders to permit third party providers (e.g., HPF Fortran, DataParallel C, C*, pC++, ADA) to port their compilers to Legion. Already two toolkits, POOMA [20] and POET [17] have been ported to Legion.

Projects such as the above typically consist of a compiler and a run-time environment. The RTE is often built upon a primitive set of operations, e.g., load, send, receive, broadcast, global sum, etc.,. These operations are either provided by the host operating system, or by a portable communication fabric such as PVM or MPI.

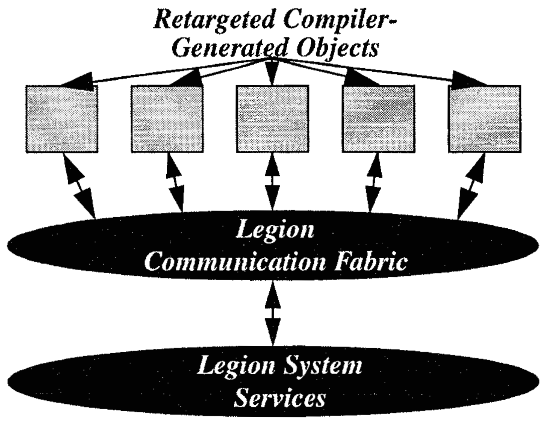

Retargeted Compiler-Generated Objects
Legion Communication Fabric
Legion System Services

Figure 2 Exposing the Legion run-time system to other parallelizing compilers.

We intend to support these other parallel processing systems by providing the same underlying communication fabric that is typically found. The compiler writers may either retarget their compilers to directly use our communication facilities, or to use a compatibility library.

Message passing API: A large number of parallel applications have been written using low-level message passing. While we do not believe that this is the right paradigm for the construction of large software systems we recognize that we must support these applications, at least in transition. Therefore we will provide several message passing compatibility libraries to programmers. We already have PVM support and are working on MPI.

4.3 Single, persistent object space

File and data access is one of the most crucial issues for Legion, particularly with respect to providing a seamless environment. Today distributed file systems such as NFS, Andrew, and Locus are commonplace in local area networks. The unified level of service and the naming scheme that they present to their users make them one of the most successful components of contemporary distributed systems. In Legion we intend to provide the same level of naming and access transparency provided in local area networks. This cannot be accomplished though either by directly extending current systems onto a national scale, or by imposing a single file system for both local and Legion access. Instead we propose to adopt a federated file system approach in much the same manner that federated database systems are constructed. The Legion file system will provide naming, access, location, fault, and replication transparency. It will permit users (or library writers) to extend the basic services provided by the file system in a clean and consistent fashion via class derivation and file-object instantiation and manipulation. The extensions that we intend to design and implement ourselves include application specific file objects designed to improve application performance by reducing observed I/O latency.

Although we will continue to refer to the Legion "file system", we intend to create a persistent object space as has been proposed for distributed object management systems [18], [19]. There are several other efforts in the distributed object literature with which we share many goals, for example SHORE [4] and CORBA [2]. However, Legion is distinguish from these efforts by the emphasis we place on performance -- Legion expects to provide a high performance computing environment and this goal is paramount. To this end we will focus more on file system support rather than database support.

The model that we will employ is simple and driven by the observation that the traditional distinction between files and other objects is somewhat of an anachronism. Files really are objects - they happen to live on disk, and as a consequence are slower to access and persist when the computer is turned off. We define a file-object as a typed object with an interface. The interface defines the operations that can be performed on it such as the traditional open, seek, and read, as well as other operations defined on a class by class basis such as read_vector, select_sub_space, etc.The interface can also define object properties such as its persistence, fault, synchronization, and performance characteristics. Thus, not all files need be the same, eliminating the need to, for example, provide Unix synchronization semantics, for all files even when many applications simply do not require those semantics. Instead, the right semantics along many dimensions can be selected on a file by file basis, and potentially changed at run-time.

Preliminary work on ELFS [13], an ExtensibLe File System, has already begun. ELFS will provide language and system support for a new set of file abstractions providing the following capabilities: (1) User specification of caching and prefetch strategies. This feature allows the user to exploit application domain knowledge about access patterns and locality to tailor the caching and prefetch strategies to the application. (2) Asynchronous I/O. ELFS permits overlapping I/O operations, including prefetching, with the applications's computation. (3) Multiple outstanding I/O requests. ELFS allows the application to request data long before it is actually needed. The application can then do some computation while I/O is being processed. By the time the data is needed, it can be had with almost zero latency. (4) Data format heterogeneity. ELFS classes may be constructed so as to hide data format heterogeneity, automatically translating data as it is read or written.

4.4 Management and exploitation of resource heterogeneity

A Legion system will encompass a potentially huge number of heterogeneous physical resources (e.g. processors, memory, networks, permanent storage, etc.) each with different capabilities and owners. Legion users will accomplish their tasks by invoking objects which will need to consume some portion of these physical resources. The question is, how do Legion objects get assigned to physical resources?

We believe that in order for a system like Legion to work, the rights of both resource providers and resource consumers must be respected. Our philosophy is that resource allocation should be by mutual consent and we therefore support a negotiation process between consumers and providers. Legion itself does not make resource allocation decisions, but rather it provides the basic mechanisms needed to 1) make informed mapping decisions between resources and objects, and 2) carry out these mapping decisions.

Figure 4 shows our model of the placement process.

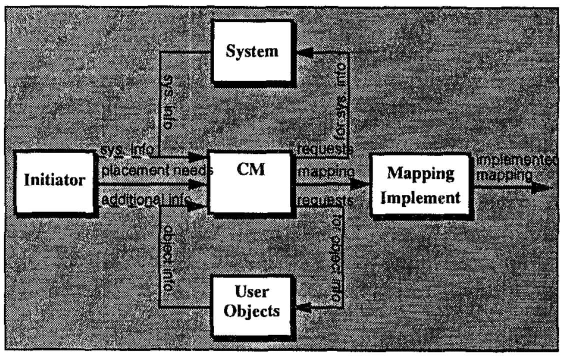


Figure 3 Object Placement Process Model

The Coordinator/Mapper (CM) is the entity that is logically responsible for making the mapping decision. Users have complete freedom to develop their own CMs tailored to their objectives and the characteristics of their applications, or they can use or modify a CM that already exists. By allowing users to control both the objective function and the search technique for the placement process, Legion can support a wide range of users and application types, including new ones not yet even imagined.

However, since users can create their own CMs, these CMs can possibly generate decisions that violate either host or object constraints. To ensure that object constraints are met, object activation is managed by the object's class object which can perform any checks it deems important and reject the placement decision if necessary. If the class object approves of the mapping decision, it contacts the target host object to request that the object be placed on the specified host. The host object is responsible for enforcing the policies of the resource provider and may accordingly also reject the request. In order to avoid generating a rash of unacceptable mappings, the LegionHost and LegionClass base classes will include member functions to allow retrieval of policy information and to allow CMs to probe to see whether particular mappings are acceptable. For more information about our plans for object placement in Legion, see [13].

4.5 Fault-tolerance

Three primary observations guide our approach to fault-tolerance in Legion: (1) in a large system, hardware and software failures affecting hosts, networks and devices will be routine occurrences, (2) fault-tolerance is not a static concept; each application may have different requirements, some may need to tolerate host failures, others network partitioning, and yet others will require no fault-tolerance at all, and (3), writing fault-tolerant applications is difficult and error prone.

Thus, Legion will not mandate any fault-tolerance policies. Instead, applications will be responsible for selecting the level of fault-tolerance that they need and are willing to pay for. This philosophy also applies to Legion itself: core Legion services may provide different fault-tolerance guarantees.

Legion will facilitate the writing of fault-tolerant applications by encapsulating fault-tolerance protocols within generic base classes. Users will then be able to select from a wide variety of protocols simply by using inheritance. The key point is that users will concern themselves with writing their applications and will not need to worry about fault-tolerance.

To realize this goal, we are investigating methods to allow objects to override their default method invocation behavior. By inheriting from a fault-tolerance class, the user is essentially allowing that class to define the default invocation mechanism and redirect the flow of control. For example, a replication base class can intercept method invocations to object X, multicast it to X's replicates, vote on the reply, and return the result to X's caller, all without any user intervention.

In addition, the Legion core model presents several features that will enable fault-tolerance: (1) the ability to dynamically query class objects for their IDL and obtain semantic information on a per class or per method basis (2) built-in support for object replications, (3) the mandatory inclusion of the SaveState() and RestoreState() methods in legion objects. These are the basic building blocks for fault-tolerance protocols based on checkpoint/restart.

5 Experiences - The CWVC and the I-way

In the summer of 1995 we released our first prototype Legion implementation, the *Campus Wide Virtual Computer* (CWVC). This first implementation is based on an earlier object-oriented parallel processing system, Mentat [9]. Mentat was originally designed to operate in homogenous, dedicated environments but has been extended to operate in an environment with heterogeneous hosts, disjoint file systems, local resource autonomy, and host failure. We could have continued to stretch Mentat but felt that one can only transform a system so far before it begins to show signs of the stress; it is often better to design from the ground up so that the resulting system has a clean coherent architecture, rather than a patchwork of modifications based on a solution for a different problem.

The campus-wide virtual computer is a direct extension of Mentat onto a larger scale, and is a prototype for the nationwide system and reflects the fact that the university is a microcosm of the world. The computational resources at the University are operated by many different departments, there is no shared name space, and sharing of resources is currently rare.

Even though the CWVC is much smaller, and the components much closer together, than in the envisioned nationwide Legion, it still presents many of the same challenges. The processors are heterogeneous, the interconnection network is irregular, with orders of magnitude differences in bandwidth and latency, and the machines are currently in use for on-site applications that must not be negatively impacted. Further, each department operates essentially as an island of service, with its own NFS mount structure, and trusting only machines in the island.

The CWVC is both a prototype and a demonstration project. The objectives are to:

- demonstrate the usefulness of network-based, heterogeneous, parallel processing to university computational science problems,
- provide a shared high-performance resource for university researchers,
- provide a given level of service (as measured by turn-around time) at reduced cost,
- act as a testbed for the nationwide Legion.

The CWVC consists of over one hundred workstations and an IBM SP-2 in six buildings using two completely disjoint underlying file systems. We have developed a suite of tools to address common problems encountered (Table 1). In collaboration with domain scientists both at the University of Virginia and elsewhere we have also developed a set of applications that exploit the environment (Table 2).

In addition to the local production environment we have also demonstrated the CWVC on wide-area systems. During Supercomputing '95 in San Diego we ran the CWVC on the I-Way, an experimental network connecting the NSF supercomputer centers, several of the DOE and NASA labs, and a number of other sites. Many of the connections were at DS-3 (45 mb/sec) and OC-3 (155 mb/sec) rates. The CWVC was installed at three sites using seven hosts of three different architectures. At NCSA (Urbana) we used for SGI Power Challenges and the Convex Exemplar. At the CTC (Cornell) and ANL (Argone) we used IBM SP-2's.

Once the IP routing tables had been properly configured moving the CWVC to the wide-area environment was relatively simple. We copied the CWVC to the platforms, adjusted the tables to use IP names that routed through the high-speed network, and tested the system. As expected, files could be accessed in a location transparent fashion, executables were transparently copied from one location to another as needed, the scheduler worked, and the system automatically reconfigured on host failure. Utilities and tools such as the debugger also migrated easily. The real bonus though was that user applications required no changes to run in the new environment.

For our demonstration we exercised our utilities, and ran one of our applications, *complib*, on the I-way. *Complib* compares two DNA or protein sequence databases using one of several selectable algorithms [11]. The first database was located at ANL, while the second was located at NCSA. The application transparently accessed the databases using the Legion file system

TABLE 1. Campus Wide Virtual Computer Toolset

Problem	Tools available
Writing parallel application	CWVC-aware PVM, parallel C++, Fortran wrappers
Multiple separate file systems	Federated file system - transparent file access
Heterogeneous resources	Automatic scheduling, binary selection and migration, application specific scheduling tools
Multiple resource owners	Owner control of resource consumption, detailed resource consumption accounting
Debugging parallel programs is hard	Post-mortem playback using off-the-shelf debuggers, e.g., dbx.
Host/network failure	Automatic system reconfiguration and limited application fault-tolerance

TABLE 2. Sample of existing CWVC applications

Discipline	Application
Biology	DNA&protein sequence comparison
Computer Science	Parallel databases & I/O, genetic algorithms
Electrical Engineering	Automatic test pattern generation, VLSI routing,
Engineering Physics	Trajectory and range of ions in matter
Physics	2D electromagnetic finite element mesh

while the underlying system schedulers placed application computation objects throughout the three-site system. All communication, placement, synchronization, and code and data migration was handled completely transparently by Legion.

Since Supercomputing we have repeated the demonstration several times, and are now in the process of constructing a more permanent prototype. The new prototype will span NCSA and SDSC and will operate as a part of the DARPA funded Distributed Object Computation Testbed.

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 [16] 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 runtime 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[6] provides communication and resource management facilities for parallel language compilers. Castle[5] 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[23] 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[21] 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 and the Future

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. The number of 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 strength is that its object model that was designed from its very inception both 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. While we have a prototype, the purpose of the prototype is to demonstrate the feasibility of constructing a wide-area system and to permit application and tool development to occur concurrently with system implementation. It is not designed to evolve directly into a complete Legion implementation.

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, debuggers, and so on, 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.

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] A. Bricker, M. Litzkow, and M. Livny, "Condor Technical Summary," Computer Sciences Department, University of Wisconsin - Madison, 10/9/91.

[4] M.J. Carey, et. al., "Shoring Up Persistent Applications," *SIGMOD 1994.*

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

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

[7] T.P. Green and J. Snyder, "DQS, A Distributed Queueing System," Florida State University, March 1993.

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

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

[10] 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.

[11] A. S. Grimshaw, E. A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, pp. 309-328, Vol. 5, issue 4, June, 1993.

[12] IBM, "IBM LoadLeveler: User's Guide (SH26-7226-02)," IBM Publication number ST00-9696, October 1994.

[13] 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.

[14] J.F. Karpovich, A.S. Grimshaw, J.C. French, "Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O", *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, pp. 191-204, Portland, OR, October 1994.

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

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

[17] J. F.Macfarlane and R. Armstrong, "POET: A Parallel Object-Oriented Environment and Toolkit for Enabling High-Performance Scientific Computing," http://www-stag.lbl.gov/poet/Poet.html.

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

[19] J.R. Nicol, C.T. Wilkes, and F.A. Manola, "Object-Orientation in Heterogeneous Distributed Systems," *IEEE Computer*, Vol. 26, No. 6, pp. 57-67, June 1993.

[20] J. Rynders, "The POOMA Framework," http://www.acl.lanl.gov/PoomaFramework/PoomaFramework.html.

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

[22] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December 1990.

[23] 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.

[24] 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.