Legion: An Operating System for Wide-Area Computing

Andrew Grimshaw, Adam Ferrari, Fritz Knabe, Marty Humphrey

Department of Computer Science
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903-2442
{grimshaw, ajf2j, knabe, humphrey}@cs.virginia.edu

Abstract

Applications enabled by the increasing availability of high-performance networks require the ability to share resources that are spread over complex, large-scale, heterogeneous, distributed environments spanning multiple administrative domains. We call this the wide-area computing problem*. We argue that the right way to solve this problem is to build an operating system for the network that can abstract over a complex set of resources and provide high-level means for sharing and managing them. We describe the design of one such wide-area operating system: Legion. Through discussion of application examples, we demonstrate the attractive features of Legion approach to constructing a wide-area operating system using distributed object components.*

1. The Challenge of Resource Sharing

The Boeing Company's designers use simulation as a key tool in making ever more complex airframes at a manageable cost. Pratt & Whitney, which designs and supplies jet engines to Boeing, also relies heavily on simulation. When Boeing's engineers simulate an airframe's behavior, they need to know how the engine coupled to that airframe will perform under various conditions. However, Pratt & Whitney cannot release its proprietary engine simulations because of the significant intellectual property they encode. This requires an unwieldy information exchange process, in which Boeing engineers ask Pratt & Whitney engineers to run their simulation at specified datapoints and then send the results to Boeing by tape. Boeing engineers then combine the information with their own simulation data and make necessary modifications. Then the whole process iterates again.

In a completely different domain, Harvard Medical School is performing research on the causes and symptoms of multiple sclerosis. The core research group has developed image processing pipelines that build three-dimensional models of characteristic brain lesions from MRI scans. To significantly advance the

research, they need MRI scans from multiple partner institutions as well as a way to make a database of image-processed results available to their research partners. As a first step, they would like a tool that can automatically identify MRI scans pertaining to the study whenever they are made at partner hospitals, securely move those scans over the Internet to Harvard, and then process them. Very little administrative support for the tool can be expected at any of the partners.

In another medical setting, seven competing Dayton, Ohio, hospitals [13] are working together to reduce costs. By sharing patient records and making them electronically available to emergency room physicians, expensive and time-consuming tests can be avoided and better care can be provided more quickly. However, each hospital has its own legacy medical records system, IS personnel, and procedures that must be brought together in the overall solution. Moreover, each has a computing base that hosts databases and programs can*not* be shared. These characteristics significantly increase the challenge of delivering a common application that spans all the institutions.

Finally, climate modeling groups at SDSC, UCLA and Lawrence Berkeley Laboratory want to couple a global atmospheric circulation model with a regional, meso-scale, weather model. The coupled models would feed data to each other, creating more accurate and detailed combined results. However, the existing regional model runs only on a Cray T90, while the global model runs on a Cray T3E and is being migrated to the IBM SP. The applications need a way to coordinate and exchange data with one another at run time, be scheduled to run simultaneously on separate supercomputers, and be easily controlled by a researcher at a single workstation.

All of these disparate examples share a common thread: the need to share and manage resources. Those resources may be hardware, software, or data, but when resources are spread over a networked environment combining multiple administrative domains, computing platforms, support levels, security policies, and myriad other factors, sharing and managing them clearly becomes significantly more difficult. We call this the *wide-area computing problem*. Though instances of the problem certainly appear in LAN environments, the rise of ubiquitous high-bandwidth networking has created both the need and the opportunity to address it. The preceding cases represent just a sampling of what people want to do.

Resource sharing is a classic computing problem with a long history. The monolithic mainframe environment led to program, data, and file sharing, mediated by the operating system. As LANs appeared, remote file and printer sharing entered the scene. The Internet allowed the most extensive file sharing mechanism of all, the World Wide Web.

The wide-area computing problem can be solved on an ad hoc, case-by-case, basis for each application, and to date that is often the approach that is used. Piecemeal solutions are cobbled together with scripts, sockets, and various network tools, and if all goes well the application can be deployed. However, these solutions tend to be brittle and limited, and require significant programmer sophistication to implement in the first place.

From a computer science perspective, the right way to solve the problem is to build an operating system for the network that can abstract over a complex set of resources and provide high-level means for sharing and managing them. But will that approach be useful to the aeronautical engineers and the medical researchers? What do these users and the developers of their applications need?

2. Demands on a Wide-Area Computing Environment

It is not difficult to produce a long list of desirable features, and we will not develop an exhaustive (and exhausting) list here. Instead we will briefly look at some essential points.

Complexity management. Complexity is the programmer's nemesis: A large-scale system composed of different architectures, many different sites, hundreds of different applications, and potentially thousands of hosts. Reducing and managing complexity is therefore critical. The object-oriented paradigm and object-based programming techniques provide programmers and application designers with encapsulation features and tools for abstraction that reduce and compartmentalize complexity. We firmly believe that object-based techniques are the key to constructing robust, wide-area systems.

These techniques are not enough, however. Composable, high-level interoperating services must replace low-level interfaces such as *rsh* and *sockets* in the programmer's toolbox. Without such services the complexity of distributed programming goes up dramatically, increasing both the skill set required to construct applications and the fragility of the resulting software.

Single system image. A major source of complexity in a wide-area system is the large number of distinct hosts and file systems. This can be tackled by providing programmers with an abstraction of a single machine and associated storage, or a single system image. "Single system image" means different things to different people--for some it means a single shared address space, for some the ability to run *ps* and get a list of all processes throughout the system. For our uses, we define a single system image as a universal shared name space that names all objects of interest to the system and its users: files, processes, processors (hosts),

storage, users, services, everything. The names should be location independent (i.e., they do not contain any location information) and should be usable from anywhere in the system. Furthermore, as the programmer uses resources to create his own objects, he should not be forced to explicitly place them on a particular host or disk---the system should handle that. This does not mean that a programmer or user cannot specify or know an object's location, but rather that if this information is not relevant to the programmer's task, it does not need to be known.

Multiple organizations. Our initial application examples illustrate the need to join multiple organizations and administrative domains. A system that facilitates this sort of bridging cannot require that sites follow a single set of policies. Instead, it must accommodate a diverse set of local use policies, access control policies, and computational cultures. For example, a site might insist that users authenticate via Kerberos before using its resources, or that users sign an "acceptable use policy" statement, or that from 1:00 PM until 6:00 PM everyday no applications be run that consume more than five CPU minutes. Extensibility and flexibility thus become essential system aspects---it must be possible to readily extend and configure the system to satisfy local requirements.

Resource heterogeneity. Resource heterogeneity is a natural part of the distributed environment. It includes processor heterogeneity, data format heterogeneity, configuration heterogeneity (e.g., how much memory and disk, which libraries are available on a host), and operating system heterogeneity. If heterogeneity is not managed individual users and programmers must deal with the complexity induced by all of the possible permutations of hardware, OS, and resources, a task that can rapidly overwhelm even the best programmers.

Scalability. The system must be able to grow without limit, adding new hosts and resources over time. If the past has shown us anything it is that the number of interconnected computational resources will only increase. Users and organizations do not want to have arbitrary limits placed on system size and capacity. Any solution to the wide-area computing problem must be able to comfortably accommodate the growth. System architectures must therefore be scalable and conform to the distributed systems principle that "the amount of service required of any single component of the system must not grow as the system grows." If an architecture does not conform, then a component whose load (e.g., requests per second) increases as the system expands will at some point become saturated, and performance will suffer.

Fault tolerance. Several years ago Leslie Lamport quipped, "A distributed system is one in which I cannot get something done because a machine I've never heard of is down." This indictment is driven by the fact that in the absence of mechanisms to deal with failure, application availability is the product of component

availability. In today's business climate, an unavailable application can easily cost thousands of dollars per minute. A wide-area system must therefore be resilient to failure and provide a failure and recovery model and associated services to applications developers, so that they can write robust applications. The model must include notions of fault detection, fault propagation, and a set of useful failure mode assumptions.

Multi-language and legacy applications. "I don't know what computer language they'll be using in a hundred years, but it will be called Fortran" was a popular refrain in the 80s. There are hundreds of millions of lines of legacy code today written in languages as varied as Lisp, RPG, Cobol, assembler, C/C++, Java, and (of course) Fortran. One thing is certain: those codes will not be replaced overnight and we will still want to be able to run them in distributed environments. The implication is that there must be a mechanism for supporting legacy code without modification, and it must be able to support a variety of programming languages. A wide-area computing environment must therefore be language-neutral.

Security. Finally, there is security. This includes a wide range of topics, such as authentication (how do I know who you are?), access control (who can do what to each resource?), and data integrity (how can I make sure that no one can read or modify my data in memory, on disk, or on the network?). Each of these three issues is present in the Boeing/Pratt & Whitney example above. Clearly we must be able to provide high levels of security, but there is more to the problem. Security can be fairly expensive in performance, restricting capabilities, and other dimensions, and different users and organizations have very different requirements and want to enforce very different policies. The challenge is to provide each user and organization with just the right mechanism and policy rules but still allow different users and organizations to interact.

If we consider these characteristics together it is clear that no commercially available middleware or operating system meets all of them. The requirements demand a *wide-area operating system*---not just an assortment of scripts and glue. We have spent the last five years designing and implementing one such wide-area OS: Legion.

3. The Legion Wide-Area Operating System

Legion is structured as a system of distributed objects. All of the entities within Legion are represented by independent, active objects that communicate using a uniform remote method invocation service. In many ways, Legion's fundamental object model is similar to CORBA's [1]: object interfaces are described using an

interface description language (IDL), and are compiled and linked to implementations in a given language (e.g., C++, Java, Fortran). This approach enables component interoperability between multiple programming languages and heterogeneous execution platforms. Objects provide a clean, natural approach to the problems of encapsulation and interoperability: because all of the elements in the system are objects they can communicate with one another regardless of location, heterogeneity, or implementation details.

Objects are thus our building blocks for constructing a wide-area OS. To understand that system, we will examine how Legion solves traditional operating system problems such as resource representation and management, task scheduling and control, naming, file systems, interprocess communication, and protection.

3.1. Resource Representation and Management

The abstraction, control, and management of underlying hardware resources is among the most fundamental services provided by any operating system. Because a Legion system runs on top of the unmodified operating system of each host in the net, it does not need to manage very low-level resources---the local OS does that job. At Legion's level, the resource base instead consists of multiple heterogeneous processors and storage devices.

Both processors and storage resources are represented as objects, called host objects and vault objects [2]. There are two primary benefits resulting from this object-based approach. First, each object defines a uniform interface to host and vault resources in a Legion system. Host objects provide a uniform interface to object (task) creation, and vault objects provide a uniform storage allocation interface, even though there may be many different implementations of each of these. Second, these objects naturally act as resource guardians and policy makers. For example, the host objects used to manage the processor resources at a given site are the points of access control for task creation at that site. If an organization participating in Legion wishes to restrict job creation on local resources exclusively to local users, the host objects at the organization's site can enforce this policy.

This object-based model for resource representation allows a tremendous degree of extensibility and site autonomy. Applications (acting as resource clients) need only be aware of the generic object interfaces for the resources they require. Resource providers can provide desired implementations of the resource objects. If the administrators of a local site wish to enforce a specialized access control policy for their processing resources, they can extend the implementation of the basic host object provided by Legion to incorporate the desired policy. If the owner of a disk wants to use local Unix-based disk-usage accounting and quota tools,

he can use a vault object implementation that allocates storage under the appropriate local Unix user-id for each client. Of course, Legion provides reasonably configurable default implementations of the basic resource objects; resource providers do not need to write any code to make their resources available to Legion. As new local resource usage policies become desirable, however, Legion explicitly supports such natural evolution.

It is important to note that resource interfaces are not carved in stone. If new interfaces for underlying resources are required, new classes of resource objects can be created to extend or replace existing interfaces. For example, a number of the processing resources in several deployed Legion networks require access through a local queue management system such as Codine [4] or LoadLeveler [5]. On such hosts, an extended queue-aware version of the Legion host object is used.

3.2. Task/Object Management

Traditional operating systems must provide the user with interfaces to start new tasks and to control the execution of existing tasks (e.g., suspend, resume, terminate). In Legion, the notion of a task or process corresponds closely to the Legion object---objects are the active computational entities within the system. The Legion interface for object control (as well as the implementation of object management functions, such as failure monitoring) is associated with a Legion object type called the *Class Manager*. Class Managers are otherwise normal Legion objects that are responsible for the management of a set of other Legion objects. The objects in the set are known as the Class Manager's instances. These instances have many similarities: each exports the same object interface and each is subject to the management policies implemented by the shared Class Manager. Class Managers are themselves managed by higher-order Class Managers, forming a rooted hierarchy known as a Legion *domain*. A complete Legion system can be composed of any number of domains, forming a forest of Class Manager hierarchies.

The interface exported by Class Managers supports a natural set of object (or task) management operations, such as methods to create objects, destroy objects, and query instance status and location. Furthermore, Class Managers serve as policy makers for their instances, controlling activities such as resource usage (permitting its instance to run only on a known set of trusted hosts, for example). Internally, Class Managers act as active monitors for their instances, maintaining up-to-date status information about each. Class Managers monitor their instances for failures, and coordinate failure response activities in case of faults.

An additional Class Manager service is persistence---all Legion objects can be persistent, existing arbitrarily beyond the lifetime of their creating program. Since Legion is intended to enable systems containing billions of objects, it supports the notion of object deactivation to avoid overloading the system with idle processes in the presence of large numbers of persistent objects. When an object is not in use it can be deactivated: its state is saved to stable storage and its containing process is deallocated. This notion of object activation/deactivation is similar to traditional operating systems temporarily swapping out a running job to stable storage then later recovering the job's state, allowing it to resume. To make object deactivation transparent to clients, the Class Manager acts as an automatic reactivation agent for its instances. If a client attempts to invoke a method on an inactive object, the object's Class Manager automatically reactivates the object, making reactivation in Legion as transparent as resuming swapped-out processes in traditional systems.

The decomposition of object management responsibilities into an arbitrary number of Class Managers provides a natural distribution and resulting scalability of object management activities in Legion. Furthermore, since Class Managers are extensible, replaceable objects, they provide a natural means for extending or replacing object management mechanisms in Legion. For example, to enable certain forms of failure resilience some Legion classes employ replication, in which case an extended version of the Class Manager creates and manages the replicas of each instance transparently to clients.

3.3. Naming

Naming is a basic interface issue in operating system design. For example, modern operating systems typically define a name space for identifying processes (e.g., PIDs in Unix), as well as a file system name space for identifying files and directories. In Legion, all entities---files, processors, storage devices, networks, users, etc.---are represented by objects, so the object naming mechanism is of central importance.

Legion objects are identified by a three-level naming scheme. At the lowest level each object is assigned an *Object Address* (OA), which contains a list of network addresses that can be used to pass messages to the object (an OA might contain an IP address and port number). But since Legion objects can migrate, OAs will vary over time. Furthermore, clients may not care about object locations. Therefore Legion defines an intermediate layer of location-independent names called *Legion Object Identifiers*, or LOIDs: unique, immutable identifiers that are assigned to objects on creation. Although higher level than OAs, LOIDs are binary, globally unique, variable-length identifiers, and do not constitute a convenient user-level naming scheme. To address this, the third level naming layer is a user-level, hierarchical directory service called

context space, which allows arbitrary Unix-like string paths to be assigned to objects. As part of its naming mechanism Legion provides scalable replicated binding services that allow translation from higher-level names to lower-level names (i.e., context paths to LOIDs and LOIDs to OAs). Also, to reduce overall binding traffic, clients cache bindings in their own memory space.

The Legion naming mechanism effectively reduces the complexity of distributed application design by providing a single global name space for all entities within the system. A typical distributed environment supports separate name spaces for files, hosts, and processes, whereas Legion supports the same global name space for all of these entities and more. Furthermore, at the highest level (context space) this naming mechanism presents an extremely simple interface of Unix-style paths.

3.4. File System

In traditional operating systems, persistent storage is typically managed in the form of a file system. Legion's use of persistent objects, coupled with the Legion global naming service, enables Legion to fully subsume the notion of a file system. Users are presented with familiar concepts of paths, directories, and universally accessible files, but Legion's "file system" is also populated with other arbitrary object types such as Host objects, Class Managers, and user application tasks.

Legion's support of a generalized persistent object space in place of a traditional rigid file system provides the basis for an extensible file system service in which individual files are customized to better suit application requirements. For example, Legion file objects can be made to support application-specific access patterns. Consider a file logically containing a two-dimensional grid of data items: in a traditional file interface access to a single row or column of the grid might require multiple file operations, but in Legion an extended file type can be used to represent the 2-D file object, providing additional methods allowing row and column access.

3.5. Interprocess Communication

To enable interprocess communication, Legion supports a variation of remote method invocation designed to address the needs of wide-area applications. Wide-area systems communication can be costly, in terms of both latency and bandwidth. Applications in the wide-area operating system require effective tools for reducing interprocess communication, and for tolerating the high latencies involved. To address this issue, Legion supports a remote method invocation model known as macro-dataflow (MDF) in addition to (and built upon) a basic, low-level message-passing service.

MDF is an asynchronous remote method invocation protocol that enables multiple concurrent method invocations from a single client as well as the overlap of remote methods and local computation. Furthermore, MDF methods encode data-dependencies for remote method results. In MDF, a remote method caller need never receive the results of that method. If the results are needed only as parameters for other future method invocations, this fact is encoded in the method invocation protocol and the remote method implementation will forward the results directly to the objects that will handle the appropriate future invocations. Legion automates this protocol, enabling the client (typically through the use of a Legion-aware compiler such as MPLC [6]) to specify complete program graphs of interdependent remote method invocations, and enabling objects to match incoming parameters into complete method invocations (including data dependencies).

3.6. Protection

Security is an integral part of the services required from a wide-area OS. Resource providers in the system desire protection from user applications and from parts of the system outside their local domain. Furthermore, they require the ability to ensure that local resources are managed by the wide-area OS in a manner that preserves local policies. Applications programmers have a complementary set of concerns, wanting to ensure that the desired security properties of their applications are achieved. To enable the expression and enforcement of security policies, by both resource providers and application programmers, Legion provides a set of security mechanisms developed as an integral part of the Legion object architecture.

The basic security service provided by Legion is user-selectable data privacy and integrity within the Legion message passing layer. Legion allows messages to be fully encrypted for privacy, digested and signed for integrity checking, or sent in the clear if low performance overhead is an application priority. Cryptographic services in Legion are based on the RSA public key system [7]. To protect against certain kinds of public key tampering, objects encode their RSA public keys directly into their LOIDs. Simply by knowing the name (LOID) for an object, a client is assured of being able to communicate securely with that object.

In any operating system, access control and resource protection are central issues. In Legion, all resources are represented by objects, so access control and resource protection are specified entirely at the object level. Access control in Legion is enforced autonomously by invoked objects on a per-method basis using a mandatory internal method called *MayI*. When a method invocation arrives at an object, it is first processed by the object's MayI method, which can enforce an arbitrary access control policy. Typically, access control decisions are made by MayI on the basis of credentials passed along with method parameters. Credentials

consist of a free-form set of rights signed by a responsible client. For example, a credential might read, "the bearer has the right to call the read method on file 'paper.txt', signed Adam," where "paper.txt" and "Adam" are object names. The default MayI implementation is based on user-configurable access control lists, including the notion of groups (supported by Group objects).

In order for system-level access control mechanisms to interface with and apply to human users, operating systems must define mechanisms for user identity and authentication. Like all other Legion entities, users are represented by objects which are assigned unique LOIDs. While the user's LOID contains his public key the user keeps his private key safe through arbitrary local means, such as a smart card. Trusted Legion programs executed by the user (e.g., the Legion login shell) rely on the user's private key to sign appropriate credentials for outgoing methods. These credentials form the basis for authenticating the user and are typically used in conjunction with per-object access control lists to enforce user access control.

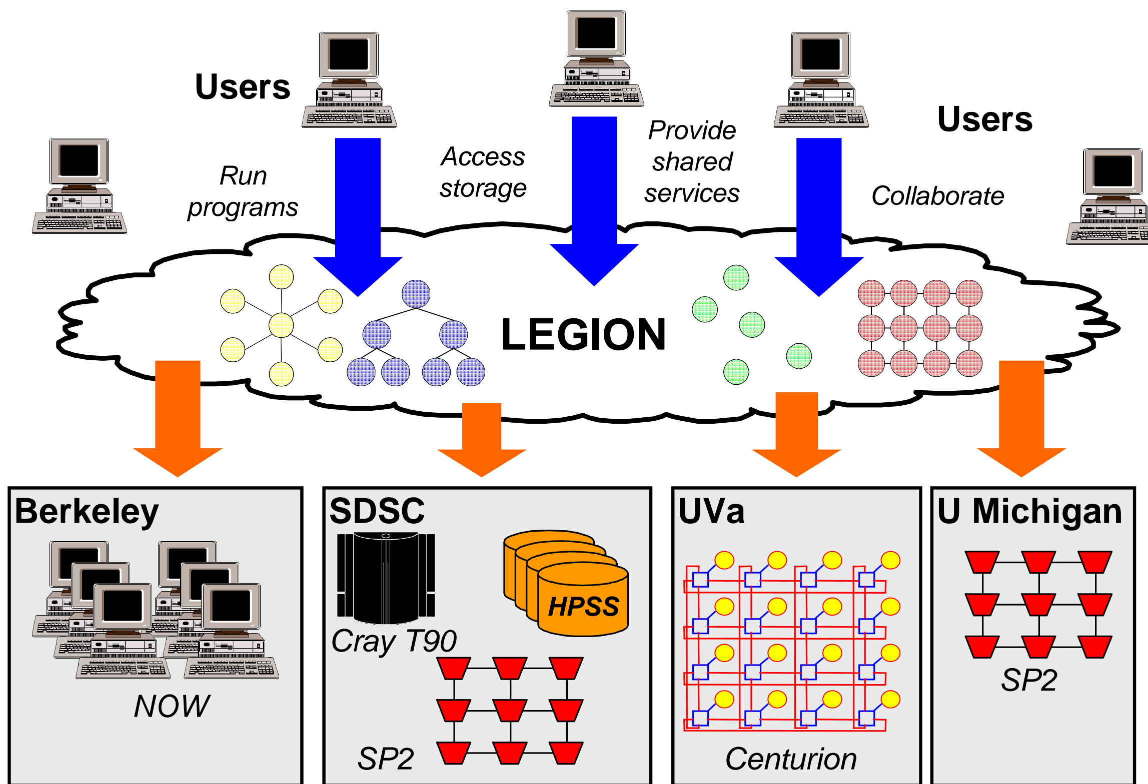

Figure 1. The Legion System. Legion acts as a wide-area operating system, providing the abstraction of a globally accessible object space to users. This object space is supported by, and provides an interface to, a widely distributed, heterogeneous resource base spanning multiple administrative domains.

4. Legion in Practice

The set of services provided by Legion comprise a wide area, cross-domain, heterogeneous operating system (see Figure 1, above). The primary goal of this operating system is to effectively support wide-area applications. So how are applications structured to use Legion's services and to satisfy their requirements in this wide-area OS environment? To answer this question we return to one of our motivating examples: the MRI data collection system for Harvard Medical School.

The components of the MRI data collection application run on central servers at Harvard and on front-end computers located at the MRI centers. The architecture is a simple star. At each leaf node there is an MRI collection object that scans the local disk for specially tagged MRI images that have been dumped by the scanner. These images are copied into the persistent data space of the object so that they will not be lost when the scanner's "dumping directory" is automatically wiped. Periodically the MRI collection object calls the central processing object at Harvard to upload the data in encrypted form, authenticating itself through the inclusion of appropriately signed certificates in the method invocations. When it receives a complete batch of scans, the central processing object starts an image processing pipeline, which consists of objects automatically scheduled onto local compute servers. The results are inserted in the project's image database.

When a leaf node is rebooted, the node's host object starts automatically and registers with its manager in the larger Legion net. The Class Manager object for the MRI collection component detects, via polling of the host object manager, that the node is up and requests a restart of the MRI collection object for that node. The host object on the node handles the request, detecting simultaneously if the MRI collection object has been upgraded and, if so, downloading the new executable automatically. As it comes up the MRI collection object recovers its state, which may included as-yet-untransmitted MRI scans.

Both the host object and MRI collection object Class Managers have replicated persistent state. If the Class Manager goes down, its own higher-order Class Manager will detect the loss and restart it using the replica. This detection and restart behavior recurses up a tree of metamanagers (typically only one or two levels) to the root Legion manager object, which has a hot spare.

The Class Manager, host, and other objects in the system are all configured with strict access control. Calls to various objects must present credentials to gain authorization. The MRI collection application and its Legion infrastructure is owned and accessible only by a small set of Legion users at Harvard. These users

can centrally monitor and configure the system using Legion tools that provide views of all the hosts, objects, etc., that are running or down.

5. Related Work

There is a rich literature on distributed systems going back over two decades. A good starting place is Sape Mullender's two distributed systems books [9,10]. These books are a collection of lecture notes from five separate instances of "The Advanced Course in Distributed Systems." Another good source is the Coulouris, Dollimore, and Kindberg textbook [11]. An excellent treatment of distributed operating systems can be found in Tanenbaum [12].

5.1 Metacomputing Systems

The Globe project [13], at Vrije University, shares many common goals and attributes with Legion. Both occupy middleware roles (running on top of existing host operating systems and networks), both support implementation flexibility, both have a single uniform object model and architecture, and both use class objects to abstract implementation details. But where a Globe object is passive and is assumed to be physically distributed over potentially many resources in the system, a Legion object is active. In addition, we don't preclude the possibility of an object being physically distributed over multiple resources but we expect that it will usually reside within a single address space. These different 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 use of core object types. Our core objects are designed to have interfaces that provide useful abstractions that in turn enable a wide variety of implementations. 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.

The Globus project [14], 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. However, we have fundamentally different philosophies driven by fundamentally different high level objectives. Globus strives to provide a basic set of services that makes it possible to write applications that operate in a wide-area environment. The Globus implementation is based on the composition of working components into a composite metacomputing toolkit. Legion strives to reduce complexity and provide the programmer with a single view of the underlying resources. Legion builds higher-level system functionality on top of a single unified object model.

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. But this sum-of-services approach has a number of drawbacks: as the number 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, traditional system-level agents such as schedulers can be migrated in Legion just as normal application processes are, since 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.

5.2. Legion and CORBA

CORBA, the Common Object Request Broker Architecture [8], is a well-known distributed object standard. CORBA's most recent version, 3.0, defines communication protocols, naming and binding mechanisms, invocation methods, persistence, and many other features and services essential for an object-based architecture. Its feature set and Legion's overlap in many areas.

Nevertheless, the two architectures are distinct in their underlying emphasis. CORBA was initially a reaction to the software integration problem. Differences between software components in location, vendor, implementation language, or execution platform made building integrated applications difficult if not impossible. The CORBA developers focused on enabling interoperability, and the architecture provides a common, object-based playing field where components can communicate and interact.

In contrast, the Legion project began with fundamental computing resources on a wide-area network---CPU, disk, data, etc.---and built an overarching framework for them. This OS-style approach targeted the ability to manage and reason about these resources. The goal was to reconstruct a coherent computing environment with core OS capabilities over a complex, heterogeneous environment. One outcome of this approach is that Legion can be used simply for its high-level OS services to run, schedule, and manage legacy applications in a network. But it also provides the same sort of common playing field as CORBA (and can mimic the CORBA standard) for integrating applications. The two aspects combined give Legion its real power.

As CORBA evolves, some operating system-type services are starting to be defined for it. Scalability and other wide-area concerns are becoming more important. It remains to be seen how well its architecture will accommodate these changes.

5.3. The World Wide Web

What is commonly referred to as the "Web" is not a single entity whose characteristics can be isolated and analyzed. Instead, the Web is a broad category of applications, protocols, and libraries, primarily focused upon content delivery to end-user clients running Web browsers. Advances in Web browser interfaces and functionality have driven the Web revolution, transforming the World Wide Web from a tool used by a few scientists into the omnipresent phenomenon it is today. Given the Web's broad scope, and the fact that the Web is most users' primary experience of distributed computing today, it is important to consider the Web's role in the wide-area OS.

First, we argue that the Web in its current form clearly does not constitute a wide-area OS of its own. Basic operating system issues, such as resource management and task scheduling, are simply not defined as part of the Web's structure. We view this not as an indictment of the Web, but a recognition of the Web's real strengths as a remote access medium for distributed content, and as a ubiquitous interface technology for accessing distributed applications.

Given these strengths, the Web constitutes a perfect front-end, or interface, to applications running in wide-area OSs, such as Legion (see Figure 2, below). Application interfaces can be written in Java, or they may use HTML and the Common Gateway Interface (CGI). They can communicate with back-end applications using either native socket protocols or HTTP, or using higher-level interfaces provided by the wide-area OS. Viewed this way, the Web and wide-area OSs such as Legion are complementary. For many users, the Web provides the most natural window into the Legion universe.

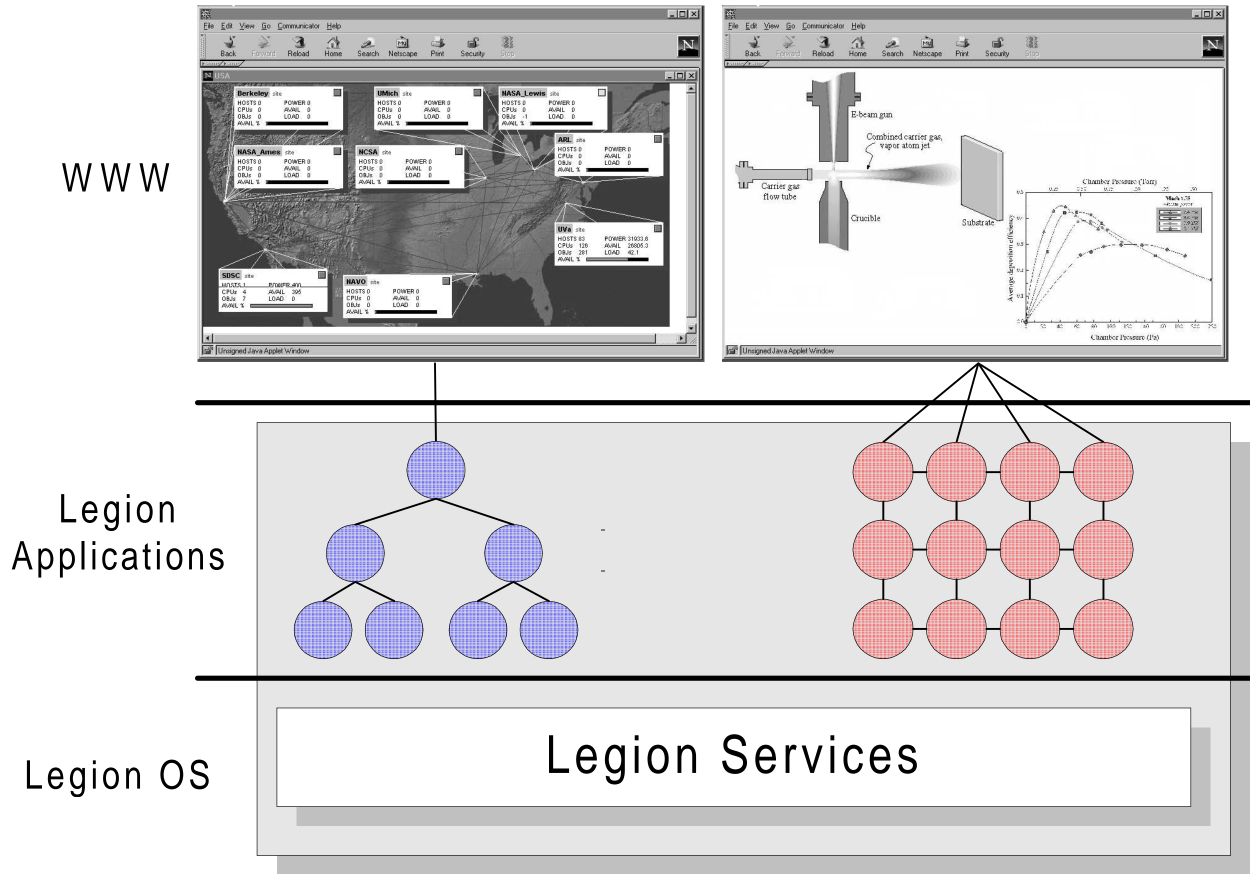

Figure 2. Legion and the Web. The Web provides an ideal interface technology for building front-ends for distributed applications. Core applications execute within Legion, using the Legion wide-area OS services. Clients running web browsers interact with applications in a variety of ways, ranging from standard HTTP and CGI, to control via active content using sockets or Legion method invocation as communication media.

6. Conclusion

Rapidly falling wide-area bandwidth prices are bringing with them increasing demand for wide-area applications: applications whose physical and software components are geographically distributed, run on multiple platforms, and often overlap multiple administrative domains. Complexity management in this environment is critical to reduce the cognitive burden on designers and programmers. We believe that simply extending existing tool sets and cobbling together ad-hoc solutions with scripts and sockets is a fundamentally flawed approach to the problem. Instead, system software that provides higher level abstractions is required.

Wide-area OS software has the ability to simplify the construction of applications for wide-area systems much in the same way that operating systems simplified the development of applications for single CPU

systems over thirty years ago. We further believe that object-based approaches to wide-area operating systems are best-suited to the problem due to their complexity encapsulation properties.

Five years ago we set out to design and build a wide-area OS. We started from scratch and designed the system from first principles to meet the needs of a wide-area, multi-organization system. The result, Legion, is an operational system that is running at a number of sites in the United States, including the two NSF supercomputer centers (SDSC and NCSA), two of the DoD supercomputer centers (NAVO and ARL), NASA ARC, and at a number of universities. (See <http://legion.virginia.edu> for more information on Legion.)

A number of scientific applications have been ported to Legion from areas as diverse as molecular biology, materials science, ocean and atmospheric science, electrical engineering, and computer science. Our experience to date has been good---users have responded particularly well to the concept of a single global object space, and the subsidiary notion of a global extensible file system that the object space supports. Furthermore, the object model has proven a convenient medium for expressing a range of user-required system services such as the Message Passing Interface (MPI), a popular library for developing distributed memory parallel programs. Starting from our successful base of system deployment and application support, we are continuing the development of higher-level services in Legion, driven by the demands of exemplary applications, such as those described in this paper.

Acknowledgments

The authors thank Charles Guttmann of the Department of Radiology, Harvard Medical School, for the MRI example, and Greg Follen of NASA LERC for briefing us on Boeing and Pratt & Whitney. Finally, thanks to Sarah Wells for her assistance on the manuscript.

References:

[1] De, P., and T. W. Ferratt, "An Information System Involving Compteting Organizations," *Communications of the ACM*, vol. 41, no. 12, pp. 90-98, December, 1998.

[2] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, OMG Document 96.03.04, Framingham, MA, 1988.

[3] Grimshaw, A.S., M. Lewis, A.J. Ferrari, and J.F. Karpovich, "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems," *Technical Report CS-98-12*, Department of Computer Science, University of Virginia, June, 1998.

[4] GENIAS Software GmbH. *Codine reference manual*, Genias, May 1993. http://www.genias.de/

[5] IBM. *LoadLeveler User's Guide, Release 2.1* (IBM SH26-7226) http://www.rs6000.ibm.com/software/sp_products/loadlev.html

[6] Grimshaw, A.S. "Easy-to-use object-oriented parallel processing with Mentat," *IEEE Computer*, pp. 39-51, May 1993.

[7] RSA Laboratories. *RSA Reference Library Implementation 2.0.* http://www.rsa.com

[8] Seetharaman, K., ed. "The CORBA Connection", special issue *Communications of the ACM*, vol. 48, no. 11, November, 1998.

[9] Mullender, S. *Distributed Systems - Second Edition*, Addison-Wesley, ACM Press, 1993.

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

[11] Coulouris, G., J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*, Addison-Wesley, 1994.

[12] Tanenbaum, A.S. *Distributed Operating Systems*, Prentice-Hall, 1995.

[13] Van Steen, M., P. Homburg, and A.S. Tanenbaum. "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, vol. 7, no. 1, pp. 70-78, January, 1999.

[14] Foster, I. and C. Kesselman. "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, 11(2):115-128, 1997.