Wide-Area Computing: Resource Sharing on a Large Scale

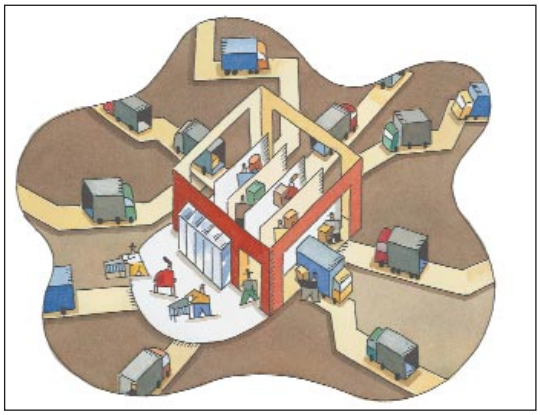
Computing over wide-area networks has been largely ad hoc, but as needs increase, piecemeal solutions no longer make sense. Legion, a network-level operating system was designed from scratch to target wide-area computing demands.

Andrew Grimshaw
Adam Ferrari
Frederick Knabe
Marty Humphrey
University of Virginia

Consider almost any computing resource today—whether hardware, software, or data—and it will invariably be networked. Networking, especially wide-area networking, has created dramatic new possibilities for resource sharing. Cooperating contractors want selected access to each other's enterprise systems. Researchers in geographically distant universities need to pool and analyze data from multisite experiments. Legacy codes on different computing platforms must exchange information to support data mining and other integrated applications.

These new possibilities depend on the ability to manage shared resources. But the sheer complexity of networked environments can turn this management problem into a nightmare. How do you share and manage resources yet maintain the autonomy of multiple administrative domains, hide the differences between incompatible computer architectures, communicate consistently as machines and network connections are lost, and respect overlapping security policies? The usual approach to these problems has been to deal with each situation individually. Piecemeal solutions are cobbled together from scripts, sockets, and various networking tools. If all goes well, a sophisticated programmer can build and maintain the application, but even then the implementation tends to be brittle and limited.

Resource management is traditionally an operating system problem, but large-scale collections of resources transcend classic operating system boundaries. What is needed is a *wide-area operating system* that can abstract over a complex set of resources and



provide a high-level way to share and manage them over the network. To be effective, such a system must address the challenges posed by real end-user applications (see the sidebar "Challenges for a Wide Area Operating System"). Scalability, security, and fault tolerance are just a few of the characteristics a viable solution must have.

Five years ago, we set out to design and build a wide-area operating system that would encompass *all* these challenges, allowing multiple organizations with diverse platforms to share and combine their resources. Our system, Legion (http://legion.virginia.edu), is now operational on hundreds of hosts across nine US sites, including the two NSF supercomputer centers (San Diego Supercomputer Center and National Center for Supercomputing Applications), two DoD supercomputer centers (Naval Oceanographic Office and Army

0018-9162/99/$10.00 © 1999 IEEE

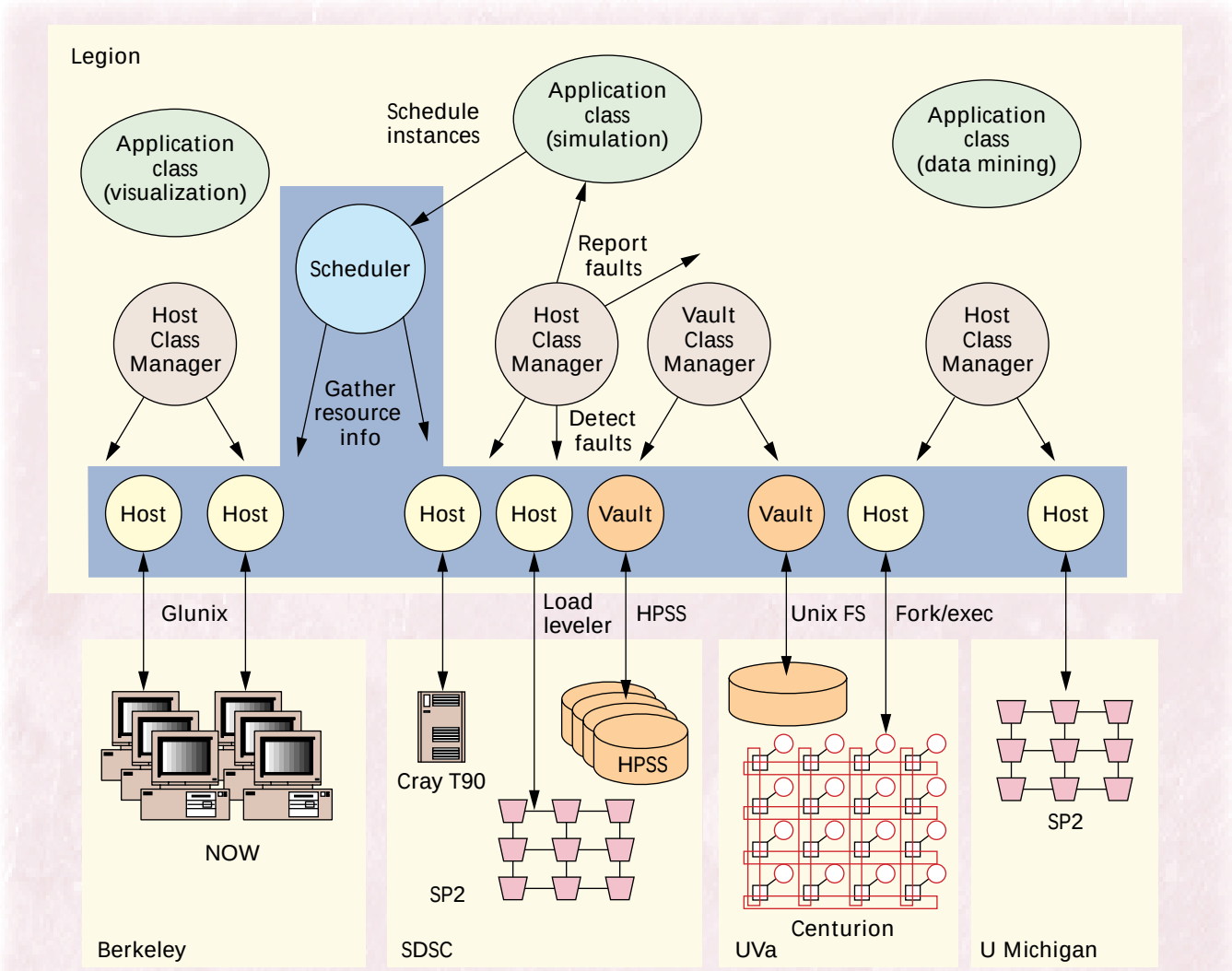


Figure 1. How the Legion wide-area operating system works. Legion Host and Vault proxy objects provide a uniform interface to heterogeneous collections of processing and storage resources. These resource proxy objects are managed by Class Manager objects. Class Managers detect and report resource faults, for example. Scheduler objects gather information about the system state; Class Managers use these Schedulers to select and access required resources. All these system components—resource objects, managers, schedulers, and application objects—are addressable in a single, system-wide, uniform object space.

Research Laboratory), NASA's Aeronautical Research Center, and several universities. Users have ported a range of scientific applications to Legion in areas such as molecular biology, materials science, ocean and atmospheric science, electrical engineering, and computer science.

Legion is essentially a conduit between the end user and widely distributed collections of resources. Like a traditional operating system, it supports services such as resource management and a distributed file system. This operating system-style interface leverages application programmer experience and simplifies porting legacy applications to the Legion platform. However, unlike typical operating systems, Legion is layered on top of existing software services. It uses the existing operating systems, resource management tools, and security mechanisms at host sites to implement higher-level system-wide services. Because of this middleware approach, Legion is able to reuse local services, and sites can retain familiar local software interfaces for applications that are not wide-area.

Legion is a component-based system: Distributed application components are represented as independent, active objects. This approach greatly simplifies the development of distributed applications and tools. Instead of facing the complexity of a wide range of distributed resources and service interfaces, the programmer works with the simple, uniform abstraction of distributed objects. Legion also supports a high level of site-autonomy. Local sites can select and configure the components that represent local resources and services in any way they see fit, retaining complete control over local access control policies, resource quota mechanisms, and so on. Legion's inherent flexibility is its greatest strength, and its most important defining characteristic.

HOW LEGION WORKS

With components that must interoperate in wide-area heterogeneous environments, Legion's fundamental object model resembles the Common Object Request Broker Architecture (CORBA). Programmers describe object interfaces in an interface description language (IDL) and then compile and link them to implementations in programming languages such as C++, Java, or Fortran. All system elements are objects and can communicate with one another regardless of location, heterogeneity, or implementation details. Within this object-based framework, Legion provides the services of a distributed operating system. Figure 1 shows how Legion works to meet a range of resource-sharing demands.

The easiest way to understand how Legion works is to consider how it handles classic operating system tasks, which we consider in turn.

Representing and managing resources

As the figure shows, local sites use Host and Vault objects to represent processors and storage, respectively.[2] Using objects to represent resources has two primary benefits:

- Objects define a simple, consistent interface to Legion's resources. Hosts provide the uniform interface for creating objects (tasks); Vaults provide the uniform interface for allocating persistent storage. These interfaces provide a consistent view of system resources, even though local resource access interfaces differ significantly in practice.
- The resource object model provides a tremendous degree of site autonomy. Applications (acting as resource clients) use the generic object interfaces for the resources they require. Resource providers can employ any desired implementation of the resource objects.

The second benefit is particularly significant. For example, if system administrators at a site want to enforce a specialized access control policy for their local hosts, they can extend or replace the basic Host implementation to enforce that policy. Similarly, some of the hosts in Legion systems may require access through a local queue management system such as Genias Software's Codine (http://www.genias.de) or IBM's LoadLeveler (http://www.rs6000.ibm.com/software/sp_products/loadlev.html). In these cases, resource providers simply use extended, queue-aware Host objects. Likewise, if a resource provider makes storage in a local file system available to Legion, yet wants to continue using local Unix-based accounting and quota tools, he can use a Vault object implemen-

Challenges for a Wide-Area Operating System

At Boeing Company, designers use simulation to make 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 data points and then send them results by tape. Boeing engineers then combine the information with their own simulation data and modify it accordingly. The process iterates.

In a completely different domain, Harvard Medical School researches the causes and symptoms of multiple sclerosis. They need to get MRI scans from multiple partner institutions and to make a database of image-processed results available to the partners. As a first step, they want a tool that can automatically identify pertinent MRI scans at partner hospitals, securely move those scans over the Internet to Harvard, and then process them. The partners will provide very little administrative support for the tool.

In another medical setting, seven competing Dayton, Ohio, hospitals are working together to reduce costs. By sharing patient records and making them electronically available to emergency room physicians, they avoid expensive and time-consuming tests and can provide better care more quickly. Each hospital has its own legacy medical records system, IS personnel, and procedures that must somehow be merged. However, each also has databases and programs that *cannot* be shared.

Finally, climate modeling groups at San Diego Supercomputer Center, UCLA, and Lawrence Berkeley Laboratory want to couple a global atmospheric circulation model with a regional, mesoscale weather model. The coupled models would feed data to each other, creating more accurate and detailed combined results. 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.

These applications characterize the spirit of wide-area computing. Some of the requirements are unique, while others overlap. The applications also illustrate the following significant challenges, from managing complexity to implementing flexible, robust security.

Provide a high-level programming model

Complexity is the programmer's nemesis: A large-scale system can comprise several different architectures, tens of sites, hundreds of applications, and potentially thousands

tation that allocates storage under the appropriate local Unix user-id for each Legion client.

Legion provides configurable default implementations of the basic resource objects, so resource providers generally need not write any code to make their resources available. However, through object extension and replacement, Legion is flexible enough to support new local resource interfaces and policies as they arise.

Managing tasks and objects

Traditional operating systems must provide interfaces for starting new tasks and controlling their execution (suspend, resume, terminate, and so on). In Legion, the notion of a task or process corresponds closely to the Legion object: Objects are the active computational entities within the system. Legion encapsulates object management functions in the Class Manager object type. Class Managers have three main functions:

- *They support a consistent interface for object management*. The Class Manager interface includes a natural set of object (or task) management operations, such as methods to create and destroy objects. Each Class Manager is responsible for a set of instances, which clients control through the Class Manager interface. Class Managers act as a policy makers for their instances. For example, an object's Class Manager determines which resources the object may use, and might enforce a policy that lets instances run only on a known set of trusted hosts.
- *They actively monitor their instances.* Class Managers query the status of their instances, detecting failures, and coordinating failure response (see Figure 1). In this role, Class Managers act as a distributed, agent-based fault-detection and response mechanism within Legion.
- *They support persistence.* All Legion objects can be persistent, existing arbitrarily beyond the life of their creating program. When an object is not in use, it can be deactivated: Its state is saved to stable storage and its containing process is deallocated (to conserve resources). This notion of object activation/deactivation is similar to traditional operating systems temporarily swapping out a job. To make object deactivation transparent to clients, the Class Manager acts as an automatic reactivation agent. If a client attempts to invoke a method on an inactive object, the Class Manager automatically reactivates it. Reactivation is thus as transparent as resuming swapped-out processes in traditional systems.

Decomposing object management responsibilities into an arbitrary number of Class Managers provides

of hosts. Reducing and managing complexity is therefore critical. The object-oriented paradigm and object-based programming 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 key to constructing robust, wide-area systems.

These techniques are not enough, however. Composable, high-level 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 build applications and the fragility of the resulting software.

Offer a single system image

To combat the daunting number of distinct hosts and file systems, programmers need a single system image—the abstraction of a single machine and associated storage. For some, a "single system image" means a single shared address space; for others, the ability to run *ps* and get a list of all processes throughout the system. We define a single system image as a universal name space and management infrastructure for all objects of interest to the system and its users: files, processes, processors (hosts), storage, users, services, and so on. The names should be location independent (not contain any location information) and should be usable from anywhere in the system. Further, as programmers use resources to create their own objects, they should not be forced to explicitly place objects on a particular host or disk—the system should handle that. Thus, the programmer or user can specify or know an object's location when necessary, but if this information is not relevant to his task, he can ignore it.

Accommodate diverse administrative policies

Most wide area computing requires joining multiple organizations and administrative domains. To makes this bridging easy, the system 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 each day from 1:00 p.m. to 6:00 p.m. no applications can be run that consume more than five CPU minutes. Extensibility and flexibility thus become essential—users must be able to readily extend and configure the system to satisfy local requirements.

Manage heterogeneous resources

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

a natural distribution of the system's object management activities. Also, because Class Managers are extensible, replaceable objects, it is easy to customize the system's object management mechanisms. For example, to enable certain forms of failure resilience, some Legion classes use replication. The specialized Class Managers used for these object classes create and manage replicas transparently to clients.

Naming

Naming is a basic interface issue in operating system design. For example, operating systems typically define a name space for identifying processes (such as Unix PIDs), as well as a file system name space for identifying files and directories. Legion represents all entities—files, processors, storage devices, networks, users, and so on—as objects. These objects are identified by a three-level naming scheme. At the lowest level, each object is assigned an *object address*—a list of network addresses for the object. An object address might contain an IP address and port number, for example. Because Legion objects can migrate, object addresses change over time. Legion thus defines an intermediate layer of location-independent names called *Legion object identifiers* (*LOIDs*). LOIDs are globally unique identifiers that are assigned to objects when they are created. Because they are binary, system-assigned names, they are not convenient for users. To address this deficiency, Legion supports a hierarchical directory service, *context space*, which lets users assign arbitrary Unix-like string paths to objects.

The Legion naming mechanism reduces the complexity of designing distributed applications because it provides a single global name space for all system entities. A typical distributed environment supports separate name spaces for files, hosts, and processes: Legion, in contrast, supports the same global name space for all these as well as additional entities. The interface to this global name space is very easy to use; at the highest level (context space) the user manipulates names in the familiar form of Unix-style paths. Furthermore, Legion's scalable replicated binding services make name translation automatic and efficient.[1]

Providing an extensible file system

Traditional operating systems typically rely on a file system to manage and represent persistent storage. However, Legion's global name space and persistent object model make a separate file system unnecessary—in practice, the generalized persistent object space defined by Legion serves all the purposes of conventional file systems. In Legion's "file system," users see familiar elements such as paths, directories, and universally accessible files, but they also see arbitrary object types such as Hosts, Class Managers, and application tasks.

Grow without limits

The system must be able to add 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 arbitrary limits on system size and capacity. 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, a component whose load (requests per second, for example) increases as the system expands will at some point become saturated, and performance will suffer.

Tolerate faults

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 a simple fact: Without 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.

Handle multilanguage 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 1980s. Hundreds of millions of lines of legacy code today are 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 be language-neutral.

Implement flexible, robust security

Security includes a range of topics, including 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 no one can read or modify my data in memory, on disk, or on the network?). Each of these issues is in the Boeing/Pratt & Whitney example. Clearly we must be able to provide high levels of security, but there is more to the problem. Security can be costly in performance, capability restriction, and other dimensions. Moreover, different users and organizations 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 to allow different users and organizations to interact.

Because of this generality, Legion's object space is more flexible than conventional file systems. For example, users can customize individual files to better suit application-specific behaviors such as specialized file access patterns. Consider a file that contains a two-dimensional grid of data items. In a traditional file interface, accessing a single grid row or column might require multiple file operations. In Legion, users can define an extended file type to represent the 2D file object, with additional methods to permit row and column access.

Enabling interprocess communication

At the lowest level, Legion objects communicate via message passing to transmit method parameters and results. However, applications for wide-area systems need tools to reduce communication and to tolerate high latencies. To address these requirements, Legion supports macrodataflow, a variation of the traditional remote method invocation model.

Like other asynchronous remote method mechanisms, macrodataflow permits multiple concurrent invocations and lets users overlap remote methods and local computation. However, unlike other remote method protocols, macrodataflow forwards method results directly to data-dependent receivers. For example, if the caller does not directly use the result of a remote method, but needs it only as a parameter for future invocations, the caller will never receive the result. The macrodataflow protocol avoids the unnecessary act of communicating the result back to the caller, and instead forwards the message directly to the objects where it is needed.

Legion fully automates the macrodataflow protocol. Clients can specify and execute program graphs of interdependent remote method invocations using macrodataflow library interfaces, or via Legion-aware compilers such as the Mentat Programmer Language Compiler.[2] Similarly, object developers need not be aware of macrodataflow; Legion automatically matches incoming method parameters from multiple sources into complete method invocations, and forwards outgoing results directly to data-dependent recipients.

Protecting resources and applications

Wide-area operating systems must protect the security of both local resource providers and application users. Resource providers require that the wide-area operating system manage local resources in accordance with local policies. Application programmers

How Legion Differs from…

Common Object Request Broker Architecture

CORBA 3.0 defines communication protocols, naming and binding mechanisms, invocation methods, persistence, and many other features and services essential for an object-based architecture.[1] As such, its feature set and Legion's overlap in many areas.

The two architectures differ in their underlying emphasis, however. 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. CORBA developers focused on enabling interoperability, and the architecture provides a common, object-based playing field where components can communicate and interact.

In contrast, Legion began with fundamental computing resources on a wide-area network—CPU, disk, data, and so on—and built an overarching framework for them. It emphasizes the ability to manage and reason about resources. The goal was to reconstruct a coherent computing environment with core operating system capabilities over a complex, heterogeneous environment. Thus, Legion can be used simply for its high-level operating system services to run, schedule, and manage legacy applications in a network, but it can mimic the CORBA standard for integrating applications. These 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.

Globe

The Globe project[2] at Vrije University also shares many 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 objects to abstract implementation details. However, the object models of the two systems differ in many respects. Globe objects are passive and are physically distributed over potentially many resources, whereas Legion objects are active, independent entities. Because of this difference, Legion provides a more unified view of system components. Whereas in Globe there is a dichotomy between objects and processes, in Legion objects are themselves the units of computation, providing the basis for distribution, scheduling, and resource management.

Globe and Legion both provide a platform for constructing applications based on interoperable components. But Legion differs significantly in also providing an integrated infrastructure for resource management. This hallmark of a wide-area operating system is essential for large-scale resource sharing.

Globus

The Globus project[3] at Argonne National Laboratory and the University of Southern California has the same base of target environments, technical objectives, and target end users as Legion, and shares some of its design features. However, Globus and

must satisfy the security requirements of their applications.

Legion's security mechanisms are an integral part of its object architecture. The basic Legion security service is user-selectable data privacy and integrity within the Legion message-passing layer. Legion lets messages 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 (http://www.rsa.com). To protect against certain kinds of public key tampering, objects encode their RSA public keys directly into their LOIDs. Simply by knowing an object's LOID, a client can 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. Invoked objects enforce access control autonomously invocation by invocation, 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, MayI makes access control decisions on the basis of credentials passed along with method parameters. Credentials consist of a free-form set of rights signed by a responsible client. The default MayI implementation is based on user-configurable access control lists, including the notion of groups.

In addition to access control mechanisms, operating systems must define mechanisms for user identity and authentication. Users (like all other Legion entities) are represented by objects, which are assigned unique LOIDs. The user's LOID contains his public key, but the user keeps his private key safe through arbitrary local means, such as a smart card. Trusted Legion programs executed by the user (the Legion login shell, for example) 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.

APPLICATIONS OF LEGION

Legion's services can accommodate a variety of domains and platforms. Two current applications illustrate its flexibility in supporting distributed enterprise computing.

Legion have fundamentally different high-level objectives. Globus provides a basic set of services that let users write applications for a wide-area environment. Working components become part of a composite distributed computing toolkit. Legion, in contrast, strives to reduce complexity and to provide the programmer with a single view of the underlying resources, so it builds higher-level system functionality on top of a single unified object model.

The Globus approach has several strong points. One is that it takes great advantage of code reuse, and builds on user knowledge of familiar tools and work environments. This approach also has several drawbacks. As the number of services grows, the lack of a common programming interface and model becomes a significant burden. By providing a common object programming model for all services, Legion permits users and tool builders to combine the many services available in the wide-area operating system: schedulers, I/O services, application components, and so on. For example, users can run the same access control tools to configure security for files and for hosts. We believe the long-term advantages of basing a system on a cohesive, comprehensive, and extensible design outweigh the short-term advantages of evolutionary composition of existing services.

The Web

The Web is not a single entity whose characteristics can be isolated and analyzed. Rather it is a broad category of applications, protocols, and libraries focused on content delivery to end-users running browsers. Advances in Web browser interfaces and functionality have driven the Web revolution, transforming it from an elitist tool to an omnipresent phenomenon. Given that the Web is most users' primary experience of distributed computing, it is important to define its role in wide-area computing.

The Web in its current form clearly does not constitute a wide-area operating system. Basic operating system issues, such as resource management and task scheduling, are simply part of the Web's structure. This is not an indictment of the Web, but a recognition of its true strength as a remote access medium for distributed content and a ubiquitous interface technology for accessing distributed applications. As such, the Web is the perfect front-end, or interface, to applications running in wide-area operating systems such as Legion. 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, HTTP, or higher-level interfaces provided by the wide-area operating system. Viewed this way, the Web and wide-area operating systems such as Legion are complementary. For many users, the Web provides the most natural window into the Legion universe.

References

1. "The CORBA Connection," *Comm. ACM*, Vol. 48, No. 11, Nov. 1998, special issue on CORBA, K. Seetharaman, ed.
2. M. Van Steen, P. Homburg, and A. Tanenbaum, "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, Vol. 7, No. 1, Jan. 1999, pp. 70-78.
3. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Int'l J. Supercomputer Applications*, Vol. 11, No. 2, 1997, pp. 115-128.

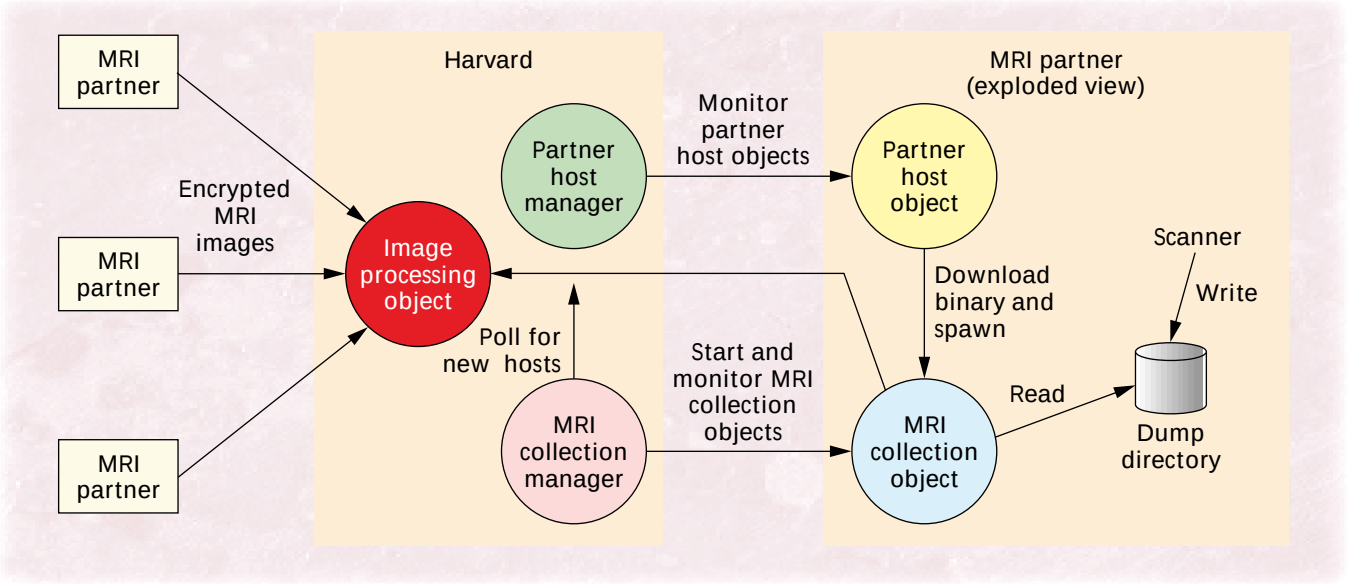


Figure 2. The MRI data collection system in development for Harvard Medical School. The components of the MRI data collection application run on central servers at Harvard and on front-end computers at the MRI centers.

MRI data collection

The MRI data collection system in development for Harvard Medical School (see first example in the sidebar "Challenges for a Wide-Area Operating System") is a good illustration of an application structure that fits well with Legion's services. The components of the MRI data collection application run on central servers at Harvard and on front-end computers at the MRI centers. Figure 2 shows the architecture. Each leaf node has an MRI collection object (blue) that scans the local disk for specially tagged MRI images that the scanner has dumped. The MRI collection object copies these images into its persistent data space so that they will not be deleted when the scanner's "dumping directory" is automatically cleaned up. Periodically the MRI collection object calls the image processing object at Harvard (red) to upload the data in encrypted form, authenticating itself by including appropriately signed certificates in the method invocations. When it receives a complete batch of scans, the image processing object starts an image-processing pipeline, which consists of objects automatically scheduled onto local compute servers. The final stage of the processing pipeline inserts the results in the project's image database.

When a leaf node is rebooted, the node's Host object (yellow) starts automatically and registers with its manager (green) in the larger Legion net. The Class Manager object (pink) for the MRI collection component detects, via polling of the green Host object Class Manager, that the node is up and requests a restart of the blue MRI collection object for that node. The yellow 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 include 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 are 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.

Climate modeling

Climate modeling has progressed beyond basic atmospheric simulations to include multiple aspects of the Earth system, such as full-depth ocean models, high-resolution land-surface models, sea ice models, and chemistry models. Typically, these models come from different research groups at a variety of institutions, are written in different languages, and require different resources. As described in the second example in the sidebar "Challenges for a Wide-Area Operating System," coupled applications composed from existing models require the ability to coordinate existing components and to manage combined resources.

Legion's ability to combine and add value to existing components to create more complex applications fits nicely with this application. To construct the coupled climate model system, developers use the existing simulations as implementations for two new Legion object types: Global Model and Mesoscale Model. In doing so, they modify the simulations to enable linkage to a Legion object interface (described in IDL), and modify the I/O calls in the models to use Legion

file objects in place of the local file system. Each new model object supports a method to request the execution of a simulation time interval. Coordination and coupling of the model objects is accomplished through the use of a Legion Coupler object. This object also transforms data from each model into the format required by the other (for example, the models employ geographic grids that differ by an order of magnitude in resolution).

Legion also satisfies this application's requirements for managing resources. For example, application developers can configure the Class Manager for the Global Model object to know that a Cray T3E is required for this object type. When the model becomes available on the IBM SP, they can reconfigure the Class Manager with a single command to account for this new resource selection possibility. When a users wants to run the complete coupled simulation, a standard Legion component, the Scheduler object, coordinates the acquisition of all needed resources (such as a T3E or SP to run the global model, a T90 to run the mesoscale model, and a workstation to host the Coupler object). Regardless of the resources selected, Legion automatically takes care of installing the needed application components at the target sites, and it uses the appropriate interfaces for the local site's task and storage allocation.

We are continuing to develop higher-level services in Legion as we acquire more information from applications. For example, broad classes of applications can profit from similar fault-response techniques. To address this need, we are designing drop-in fault-tolerance modules based on the existing detection and reporting infrastructure. We also plan to develop new application tools, such as an integrated Legion debugger, and port application toolkits such as Netsolve (http://www.cs.utk.edu/netsolve). These efforts are guided by our close collaborations with an expanding set of applications groups, such as the Harvard Medical School and the climate modeling groups mentioned earlier. Finally, we are actively engaged in commercializing the Legion platform for use in Internet and enterprise settings. For more information, visit the Legion site (http://legion.virginia.edu). ❖

Acknowledgments

We thank Charles Guttmann of the Department of Radiology, Harvard Medical School, for the MRI example, and Greg Follen of NASA's Lewis Research Center for briefing us on Boeing and Pratt & Whitney. We also thank Sarah Wells for her assistance.

References

1. A. Grimshaw et al., "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems," Tech. Report CS-98-12, Computer Science Dept., Univ. of Virginia, Charlottesville, Va., 1998.
2. A. Grimshaw, "Easy-to-Use Object-Oriented Parallel Processing with Mentat," *Computer*, May 1993, pp. 39-51.

Andrew Grimshaw is an associate professor of computer science at the University of Virginia, where his research interests include metasystems, high-performance parallel computing, heterogeneous parallel computing, compilers for parallel systems, and operating systems. He is the chief designer and architect of Mentat and Legion. He received a PhD in computer science from the University of Illinois at Urbana-Champaign.

Adam Ferrari is a research scientist with the Legion project in the Department of Computer Science at the University of Virginia. His research interests include high-performance distributed computing, operating systems, metacomputing, and computer security. He received an MS from Emory University and a PhD from the University of Virginia, both in computer science. He is a member of the IEEE Computer Society and the ACM.

Frederick Knabe is a senior research scientist in the Department of Computer Science at the University of Virginia. His research interests include wide-area computing, computer security, and software risks. He received a PhD in computer science from Carnegie Mellon University.

Marty Humphrey is a research assistant professor of computer science at the University of Virginia. His research interests include real-time operating systems, real-time scheduling, distributed computing, and metacomputing. He received a PhD in computer science from the University of Massachusetts and is a member of the IEEE.

Contact the authors at {grimshaw, ferrari, knabe, humphrey}@virginia.edu.


XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX


XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX


XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX