Legion: Lessons Learned Building a Grid Operating System

ANDREW S. GRIMSHAW AND ANAND NATRAJAN

Invited Paper

Legion was the first integrated grid middleware architected from first principles to address the complexity of grid environments. Just as a traditional operating system provides an abstract interface to the underlying physical resources of a machine, Legion was designed to provide a powerful virtual machine interface layered over the distributed, heterogeneous, autonomous, and fault-prone physical and logical resources that constitute a grid. We believe that without a solid, integrated, operating system-like grid middleware, grids will fail to cross the chasm from bleeding-edge supercomputing users to more mainstream computing. This paper provides an overview of the architectural principles that drove Legion, a high-level description of the system with complete references to more detailed explanations, and the history of Legion from first inception in August 1993 through commercialization. We present a number of important lessons, both technical and sociological, learned during the course of developing and deploying Legion.

Keywords*—Distributed object system, grid, grid architecture, grid design philosophy, large-scale distributed system, metaoperating systems, metasystems.*

I. INTRODUCTION

Grids (once called metasystems [20]–[23]) are collections of interconnected resources harnessed together in order to satisfy various needs of users [24], [25]. The resources may be administered by different organizations and may be distributed, heterogeneous, and fault-prone. The manner in which users interact with these resources as well as the usage policies for the resources may vary widely. A grid infrastructure must manage this complexity so that users can interact with resources as easily and smoothly as possible.

Our definition, and indeed a popular definition, is: A *grid* system, also called a grid, gathers resources—desktop and handheld hosts, devices with embedded processing resources such as digital cameras and phones or terascale supercomputers—and makes them accessible to users and applications in order to reduce overhead and accelerate projects. A grid application can be defined as an application that operates in a grid environment or is "on" a grid system. Grid system software (or middleware), is software that facilitates writing grid applications and manages the underlying grid infrastructure. The resources in a grid typically share at least some of the following characteristics.

- They are numerous.
- They are owned and managed by different, potentially mutually distrustful organizations and individuals.
- They are potentially faulty.
- They have different security requirements and policies.
- They are heterogeneous, e.g., they have different CPU architectures, are running different operating systems, and have different amounts of memory and disk.
- They are connected by heterogeneous, multilevel networks.
- They have different resource management policies.
- They are likely to be geographically separated (on a campus, in an enterprise, on a continent).

The above definitions of a grid and a grid infrastructure are necessarily general. What constitutes a "resource" is a deep question, and the actions performed by a user on a resource can vary widely. For example, a traditional definition of a resource has been "machine," or more specifically "CPU cycles on a machine." The actions users perform on such a resource can be "running a job," "checking availability in terms of load," and so on. These definitions and actions are legitimate, but limiting. Today, resources can be as diverse as

Manuscript received March 1, 2004; revised June 1, 2004. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under (Navy) Contract N66001-96-C-8527, in part by the Department of Energy (DOE) under Grant DE-FG02-96ER25290, in part by DOE under Contract Sandia LD-9391, in part by Logicon (for the DoD HPCMOD/PET program) under Contract DAHC 94-96-C-0008, in part by the DOE under Contract D459000-16-3C, in part by DARPA (GA) under Contract SC H607305A, in part by the National Science Foundation (NSF)-Next Generation Systems (NGS) under Grant EIA-9974968, in part by NSF-National Partnership for Advanced Computational Infrastructure (NPACI) under Grant ASC-96-10920, and in part by a grant from NASA-IPG.

The authors are with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904–4740 USA (e-mail: grimshaw@virginia.edu).

Digital Object Identifier 10.1109/JPROC.2004.842764

"biotechnology application," "stock market database," and "wide-angle telescope," with actions being "run if license is available," "join with user profiles," and "procure data from specified sector," respectively. A grid can encompass all such resources and user actions. Therefore, a grid infrastructure must be designed to accommodate these varieties of resources and actions without compromising on some basic principles such as ease of use, security, autonomy, etc.

A grid enables users to collaborate securely by sharing processing, applications, and data across systems with the above characteristics in order to facilitate collaboration, faster application execution, and easier access to data. More concretely, this means being able to do the following.

- Find and share data. Access to remote data should be as simple as access to local data. Incidental system boundaries should be invisible to users who have been granted legitimate access.
- Find and share applications. Many development, engineering and research efforts consist of custom applications—permanent or experimental, new or legacy, public domain or proprietary—each with its own requirements. Users should be able to share applications with their own data sets.
- Find and share computing resources. Providers should be able to grant access to their computing cycles to users who need them without compromising the rest of the network.

This paper describes one of the major grid projects of the last decade—Legion—from its roots as an academic grid project to its current status as the only commercial complete grid offering [3]–[7], [9], [10], [14], [16], [18], [19], [21]–[23], [26]–[29], [31]–[47].

Legion is built on the decades of research in distributed and object-oriented systems, and borrows many, if not most, of its concepts from the literature [48]–[82]. Rather than reinvent the wheel, the Legion team sought to combine solutions and ideas from a variety of different projects such as Eden/Emerald [48], [53], [55], [83], Clouds [67], AFS [72], Coda [84], CHOICES [85], PLITS [63], Locus [76], [81], and many others. What differentiates Legion from its progenitors is the scope and scale of its vision. While most previous projects focus on a particular aspect of distributed systems such as distributed file systems, fault tolerance, or heterogeneity management, the Legion team strove to build a complete system that addressed all of the significant challenges presented by a grid environment. To do less would mean that the end user and applications developer would need to deal with the problem. In a sense, Legion was modeled after the power grid system—the underlying infrastructure manages all the complexity of power generation, distribution, transmission, and fault-management so that end users can focus on issues more relevant to them, such as which appliance to plug in and how long to use it. Similarly, Legion was designed to operate on a massive scale, across WANs, and between mutually distrustful administrative domains, while most earlier distributed systems focused on the local area, typically a single administrative domain.

Beyond merely expanding the scale and scope of the vision for distributed systems, Legion contributed technically in a range of areas as diverse as resource scheduling and high-performance I/O. Three of the more significant technical contributions were: 1) the extension of the traditional event model to ExoEvents [13]; 2) the naming and binding scheme that supports both flexible semantics and lazy cache coherence [9]; and 3) a novel security model [17] that started with the premise that there is no trusted third party.

What differentiates Legion first and foremost from its contemporary grid projects such as Globus[1] [86]–[93] is that Legion was designed and engineered from first principles to meet a set of articulated requirements, and that Legion focused from the beginning on ease of use and extensibility. The Legion architecture and implementation was the result of a software engineering process that followed the usual form of the following.

1) Develop and document requirements.
2) Design and document solution.
3) Test design on paper against all requirements and interactions of requirements.
4) Repeat 1–3 until there exists a mapping from all requirements onto the architecture and design.
5) Build and document 1.0 version of the implementation.
6) Test against application use cases.
7) Modify design and implementation based on test results and new user requirements.
8) Repeat steps 6–7.

This is in contrast to the approach used in other projects of starting with some basic functionality, seeing how it works, adding/removing functionality, and iterating toward a solution.

Second, Legion focused from the very beginning on the end-user experience via the provisioning of a transparent, reflective, abstract virtual machine that could be readily extended to support different application requirements. In contrast, the Globus approach was to provide a basic set of tools to enable the user to write grid applications and manage the underlying tools explicitly.

The remainder of this paper is organized as follows. We begin with a discussion of the fundamental requirements for any complete grid architecture. These fundamental requirements continue to guide the evolution of our grid software. We then present some of the principles and philosophy underlying the design of Legion. We then introduce some of the architectural features of Legion and delve slightly deeper into implementation in order to give an understanding of grids and Legion. Detailed technical descriptions exit elsewhere in the literature and are cited. We then present a brief history of Legion and its transformation into a commercial grid product, Avaki 2.5. We then present the major lessons, not all technical, learned during the course of the project. We then summarize with a few observations on trends in grid computing.

Keep in mind that the objective here is not to provide a detailed description of Legion, but to provide a perspective

[1]See [45] for a more complete comparison.

with complete references to papers that provide much more detail.

II. Requirements

Clearly, the minimum capability needed to develop grid applications is the ability to transmit bits from one machine to another—all else can be built from that. However, several challenges frequently confront a developer constructing applications for a grid. These challenges lead us to a number of requirements that any complete grid system must address. The designers of Legion believed and continue to believe that all of these requirements must be addressed by the grid infrastructure in order to reduce the burden on the application developer. If the system does not address these issues, then the programmer must—forcing programmers to spend valuable time on basic grid functions, thus needlessly increasing development time and costs. The requirements are as follows.

- ***Security***. Security covers a gamut of issues, including authentication, data integrity, authorization (access control) and auditing. If grids are to be accepted by corporate and government IT departments, a wide range of security concerns must be addressed. Security mechanisms must be integral to applications and capable of supporting diverse policies. Furthermore, we believe that security must be built in firmly from the beginning. Trying to patch security in as an afterthought (as some systems are attempting today) is a fundamentally flawed approach. We also believe that no single security policy is perfect for all users and organizations. Therefore, a grid system must have mechanisms that allow users and resource owners to select policies that fit particular security and performance needs, as well as meet local administrative requirements.
- ***Global name space***. The lack of a global name space for accessing data and resources is one of the most significant obstacles to wide-area distributed and parallel processing. The current multitude of disjoint name spaces greatly impedes developing applications that span sites. All grid objects must be able to access (subject to security constraints) any other grid object *transparently* without regard to location or replication.
- ***Fault tolerance***. Failure in large-scale grid systems is and will be a fact of life. Machines, networks, disks and applications frequently fail, restart, disappear, and behave otherwise unexpectedly. Forcing the programmer to predict and handle all of these failures significantly increases the difficulty of writing reliable applications. Fault-tolerant computing is known to be a very difficult problem. Nonetheless, it must be addressed, or else businesses and researchers will not entrust their data to grid computing.
- ***Accommodating heterogeneity***. A grid system must support interoperability between heterogeneous hardware and software platforms. Ideally, a running application should be able to migrate from platform to platform if necessary. At a bare minimum, components running on different platforms must be able to communicate transparently.
- ***Binary management and application provisioning***. The underlying system should keep track of executables and libraries, knowing which ones are current, which ones are used with which persistent states, where they have been installed and where upgrades should be installed. These tasks reduce the burden on the programmer.
- ***Multilanguage support***. Diverse languages will always be used, and legacy applications will need support.
- ***Scalability***. There are over 500 million computers in the world today and over 100 million network-attached devices (including computers). Scalability is clearly a critical necessity. Any architecture relying on centralized resources is doomed to failure. A successful grid architecture must adhere strictly to the distributed systems principle: the service demanded of any given component must be independent of the number of components in the system. In other words, the service load on any given component must not increase as the number of components increases.
- ***Persistence***. I/O and the ability to read and write persistent data are critical in order to communicate between applications and to save data. However, the current files/file libraries paradigm should be supported, since it is familiar to programmers.
- ***Extensibility***. Grid systems must be flexible enough to satisfy current user demands and unanticipated future needs. Therefore, we feel that mechanism and policy must be realized by replaceable and extensible components, including (and especially) core system components. This model facilitates development of improved implementations that provide value-added services or site-specific policies while enabling the system to adapt over time to a changing hardware and user environment.
- ***Site autonomy***. Grid systems will be composed of resources owned by many organizations, each of which desires to retain control over its own resources. The owner of a resource must be able to limit or deny use by particular users, specify when it can be used, etc. Sites must also be able to choose or rewrite an implementation of each Legion component as best suits their needs. If a given site trusts the security mechanisms of a particular implementation, it should be able to use that implementation.
- ***Complexity management***. Finally, but importantly, complexity management is one of the biggest challenges in large-scale grid systems. In the absence of system support, the application programmer is faced with a confusing array of decisions. Complexity exists in multiple dimensions: heterogeneity in policies for resource usage and security, a range of different failure modes and different availability requirements, disjoint namespaces and identity spaces, and the sheer number of components. For example, professionals who are

not IT experts should not have to remember the details of five or six different file systems and directory hierarchies (not to mention multiple user names and passwords) in order to access the files they use on a regular basis. Thus, providing the programmer and system administrator with clean abstractions is critical to reducing their cognitive burden.

III. Philosophy

To address these basic grid requirements, we developed the Legion architecture and implemented an instance of that architecture, the Legion runtime system [11], [94]. The architecture and implementation were guided by the following design principles that were applied at every level throughout the system.

- ***Provide a single-system view***. With today's operating systems and tools such as Load Sharing Facility (LSF), Sun Grid Engine (SGE), and Portable Batch System (PBS), we can maintain the illusion that our LAN is a single computing resource. But once we move beyond the local network or cluster to a geographically dispersed group of sites, perhaps consisting of several different types of platforms, the illusion breaks down. Researchers, engineers, and product development specialists (most of whom do not want to be experts in computer technology) are forced to request access through the appropriate gatekeepers, manage multiple passwords, remember multiple protocols for interaction, keep track of where everything is located, and be aware of specific platform-dependent limitations (e.g., this file is too big to copy or to transfer to that system; that application runs only on a certain type of computer, etc.). Re-creating the illusion of a single computing resource for heterogeneous distributed resources reduces the complexity of the overall system and provides a single namespace.
- ***Provide transparency as a means of hiding detail***. Grid systems should support the traditional distributed system transparencies: access, location, heterogeneity, failure, migration, replication, scaling, concurrency, and behavior. For example, users and programmers should not have to know where an object is located in order to use it (access, location, and migration transparency), nor should they need to know that a component across the country failed—they want the system to recover automatically and complete the desired task (failure transparency). This behavior is the traditional way to mask details of the underlying system.
- ***Provide flexible semantics***. Our overall objective was a grid architecture that is suitable to as many users and purposes as possible. A rigid system design in which policies are limited, tradeoff decisions are preselected, or all semantics are predetermined and hard-coded would not achieve this goal. Indeed, if we dictated a single system-wide solution to almost any of the technical objectives outlined above, we would preclude large classes of potential users and uses. Therefore, Legion allows users and programmers as much flexibility as possible in their applications' semantics, resisting the temptation to dictate solutions. Whenever possible, users can select both the *kind* and the *level* of functionality and choose their own tradeoffs between function and cost. This philosophy is manifested in the system architecture. The Legion object model specifies the functionality but not the implementation of the system's core objects; the core system, therefore, consists of extensible, replaceable components. Legion provides default implementations of the core objects, although users are not obligated to use them. Instead, we encourage users to select or construct object implementations that answer their specific needs.
- ***Reduce user effort***. In general, there are four classes of grid users who are trying to accomplish some mission with the available resources: end users of applications, applications developers, system administrators, and managers. We believe that users want to focus on their jobs, e.g., their applications, and not on the underlying grid plumbing and infrastructure. Thus, for example, to run an application a user may type *legion_run my_application my_data* at the command shell. The grid should then take care of all of the messy details such as finding an appropriate host on which to execute the application, moving data and executables around, etc. Of course, the user may optionally be aware and specify or override certain behaviors, for example, specify an architecture on which to run the job, or name a specific machine or set of machines, or even replace the default scheduler.
- ***Reduce "activation energy."*** One of the typical problems in technology adoption is getting users to use it. If it is difficult to shift to a new technology, then users will tend not to take the effort to try it unless their need is immediate and extremely compelling. This is not a problem unique to grids—it is human nature. Therefore, one of our most important goals was to make using the technology easy. Using an analogy from chemistry, we kept the activation energy of adoption as low as possible. Thus, users can easily and readily realize the benefit of using grids—and get the reaction going—creating a self-sustaining spread of grid usage throughout the organization. This principle manifests itself in features such as "no recompilation" for applications to be ported to a grid and support for mapping a grid to a local operating system file system. Another variant of this concept is the motto "no play, no pay." The basic idea is that if you do not need a feature, e.g., encrypted data streams, fault resilient files, or strong access control, you should not have to pay the overhead of using it.
- ***Do no harm***. To protect their objects and resources, grid users and sites will require grid software to run with the lowest possible privileges.

- ***Do not change host operating systems***. Organizations will not permit their machines to be used if their operating systems must be replaced. Our experience with Mentat [95] indicates, though, that building a grid on top of host operating systems is a viable approach. Furthermore, Legion must be able to run as a user level process and not require root access.

Overall, the application of these design principles *at every level* provides a unique, consistent, and extensible framework upon which to create grid applications.

IV. Legion—The Grid Operating System

The traditional way in computer science to deal with complexity and diversity is to build an abstraction layer that masks most, if not all, of the underlying complexity. This approach led to the development of modern operating systems, which were developed to provide higher level abstractions—both from a programming and management perspective—to end users and administrators. In the early days, one had to program on the naked machine, write one's own loaders, device drivers, etc. These tasks were inconvenient and forced all programmers to perform them repetitively. Thus, operating systems were born. Blocks on disk become files, CPUs and memory were virtualized by CPU multiplexing and virtual memory systems, security for both user processes and the system was enforced by the kernel, and so on.

Viewed in this context, grid software is a logical extension of operating systems from single machines to large collections of machines—from supercomputers to handheld devices. Just as, in the early days of computing, one had to write one's own device drivers and loaders, in the early days of grid computing users managed moving binaries and data files around the network, dealing directly with resource discovery, etc. Thus, grid systems are an extension of traditional operating systems applied to distributed resources. And as we can see from the above requirements, many of the same management issues apply: process management, interprocess communication, scheduling, security, persistent data management, directory maintenance, accounting, and so on.

Legion started out with the "top-down" premise that a solid architecture is necessary to build an infrastructure of this scope. Consequently, much of the initial design time spent in Legion was in determining the underlying infrastructure and the set of core services over which grid services could be built.

In Fig. 1, we show a layered view of the Legion architecture. Below we briefly expand on each layer, starting with the bottom layer.

The bottom layer is the local operating system—or execution environment layer. This layer corresponds to true operating systems such as Linux, AIX, Windows 2000, etc., as well hosting environments such as J2EE. We depend on process management services, local file system support, and interprocess communication services delivered by this layer, e.g., UDP, TCP, or shared memory.

A. Naming and Binding

Above the local operating services layer, we built the Legion ExoEvent system and the Legion communications layer. The communication layer is responsible for object naming and binding as well as delivering sequenced arbitrarily long messages from one object to another. Delivery is accomplished regardless of the location of the two objects, object migration, or object failure. For example, object A can communicate with object B even while object B is migrating from Charlottesville, VA, to San Diego, CA, or even if object B fails and subsequently restarts. This transparency is possible because of Legion's three-level naming and binding scheme, in particular the lower two levels.

The lower two levels of the naming scheme consist of location-independent abstract names called Legion Object IDentifiers (LOIDs) and object addresses (OAs) specific to communication protocols, e.g., an IP address and a port number. The binding between a LOID and an OA can, and does, change over time. Indeed it is possible for there to be no binding for a particular LOID at some times if, for example, the object is not running currently. Maintaining the bindings at runtime in a scalable way is one of the most important aspects of the Legion implementation.

The basic problem is to allow the $\langle \text{LOID}, \text{OA} \rangle$ binding to change arbitrarily while providing high performance. Clearly, one could bind a LOID to an OA on every method call on an object. The performance, though, would be poor, and the result nonscalable. If the bindings are cached for performance and scalability reasons, we run the risk of an incoherent binding if, for example, an object migrates. To address that problem, one could have, for example, callback lists to notify objects and caches that their bindings are stale, as is done in some shared-memory parallel processors. The problem with callbacks in a distributed system is scale—thousands of clients may need notification—and the fact that many of the clients themselves may have migrated, failed, or become disconnected from the grid, making notification unreliable. Furthermore, it is quite possible that the bindings will never be used by many of the caches again—with the result that significant network bandwidth may be wasted.

At the other extreme, one could bind at object creation—and never allow the binding to change, providing excellent scalability, as the binding could be cached throughout the grid without an coherence concerns.

The Legion implementation combines all of the performance and scalability advantages of caching, while eliminating the need to keep the caches strictly coherent. We call the technique lazy coherence. The basic idea is simple. A client uses a binding that it has acquired by some means, usually from a cache. We exploit the fact that the Legion message layer can detect if a binding is stale, e.g., the OA endpoint is not responding or the object (LOID) at the other end is not the one the client expects because an address is being reused or masqueraded. If the binding is stale, the client requests a new binding from the cache while informing the cache not to return the same bad binding. The cache then looks up the

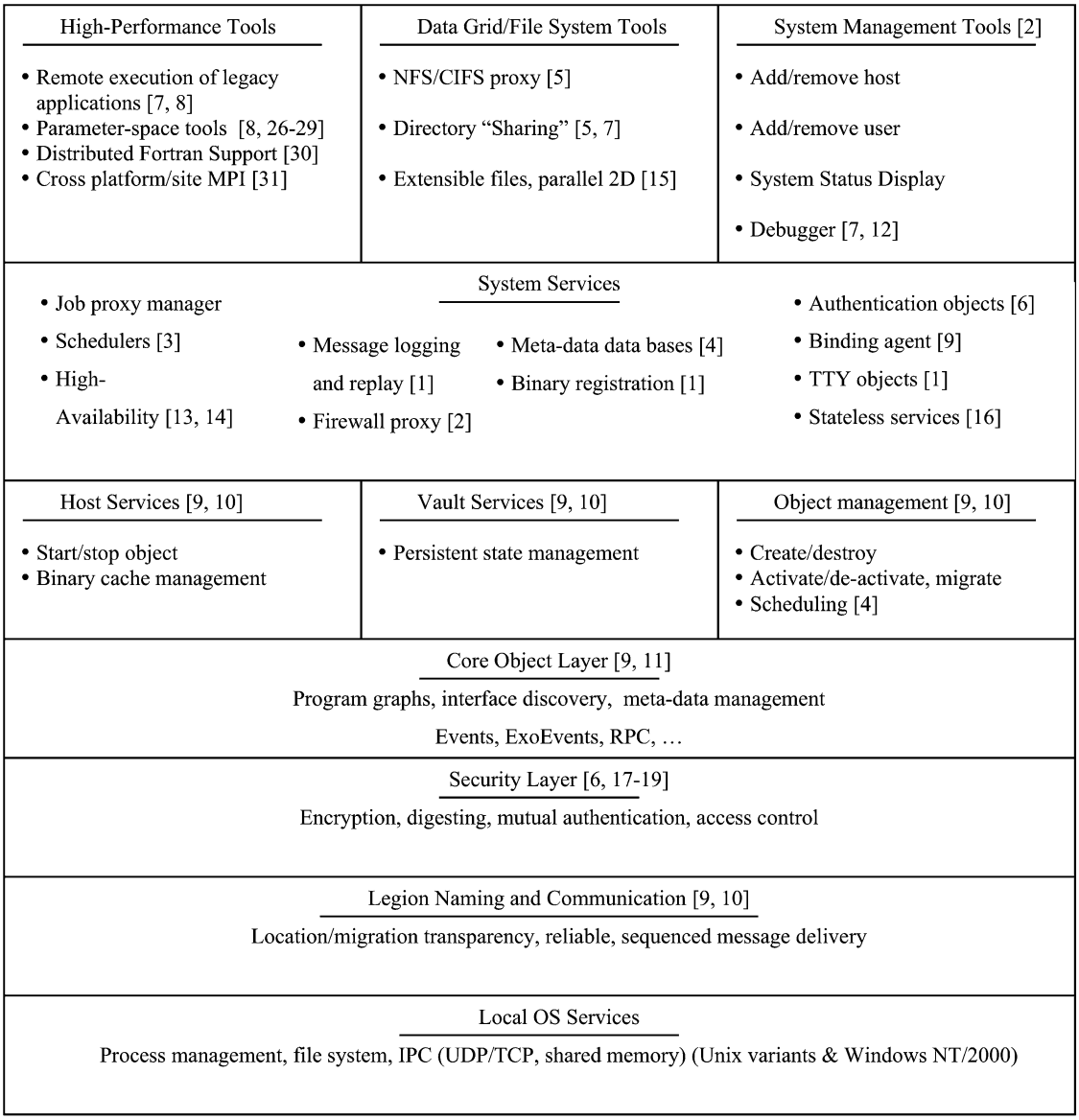

Fig. 1. Legion architecture viewed as a series of layers with references to relevant papers.

LOID. If it has a different binding than the one the client tried, the cache returns the new binding. If the binding is the same, the cache requests an updated binding either from another cache (perhaps organized in a tree similar to a software combining tree [96]), or goes to the object manager. The net result of this on-demand, lazy coherence strategy is that only those caches where the new binding is actually used are updated.

B. Core Object Management and Security

The next layers in the Legion architecture are the security layer and the core object layers. The security layer implements the Legion security model [6], [17] for authentication, access control, and data integrity (e.g., mutual authentication and encryption on the wire). The security environment in grids presents some unusual challenges. Unlike single enterprise systems, where one can often assume that the administrative domains "trust" one another,[2] in a multiorganizational grid there is neither mutual trust nor a single trusted third party. Thus, a grid system must permit flexible access control, site autonomy, local decisions on authentication, and delegation of credentials and permissions [97].

The core object layer [9]–[11] addresses method invocation, event processing (including ExoEvents [13]), interface discovery and the management of metadata. Objects can have arbitrary metadata, such as the load on a machine or the parameters that were used to generate a particular data file.

Above the core object layer are the core services that implement object instance management (*class managers*), abstract processing resources (*hosts*), and storage resources (*vaults*). These are represented by base classes that can be extended to provide different or enhanced implementations. For example, a *host* represents processing resources. It has

[2]In our experience though, the units within a larger organization rarely truly trust one another.

methods to start an object given a LOID, a persistent storage address, and the LOID of an implementation to use; stop an object given a LOID; kill an object; provide accounting information; and so on. The UNIX and Windows versions of the host class, called *UnixHost* and *NTHost*, use UNIX processes and Windows spawn, respectively, to start objects. Other versions of hosts interact with backend third-party queuing systems (*BatchQueueHost*) or require the user to have a local account and run as that user (*PCDHost* [2]).

The object instance managers are themselves instances of classes called *metaclasses*. These metaclasses can also be overloaded to provide a variety of object metabehaviors. For example, replicated objects for fault tolerance, or stateless objects for performance and fault tolerance [13], [14], [16]. This ability to change the basic metaimplementation of how names are bound and how state is managed is a key aspect of Legion that supports extensibilty.

Above these basic object management services is a whole collection of higher level system service types and enhancements to the base service classes. These include classes for object replication for availability [14], message logging classes for accounting [2], or postmortem debugging [12], firewall proxy servers for securely transiting firewalls, enhanced schedulers [3], [4], databases called collections that maintain information on the attributes associated with objects (these are used extensively in scheduling), job proxy managers that "wrap" legacy codes for remote execution [7], [26]–[29], and so on.

C. Application-Layer Support, or "Compute" and "Data" Grids

An application support layer built over the layers discussed above contains user-centric tools for parallel and high-throughput computing, data access and sharing, and system management. See [42] for a more detailed look at the user-level view.

The high-performance toolset includes tools [1] to wrap legacy applications (*legion_register_program*) and execute them remotely (*legion_run*) both singly and in large sets as in a parameter-space study (*legion_run_multi*). Legion Message Passing Interface (MPI) tools support cross-platform, cross-site execution of MPI programs [31], and Basic Fortran Support (BFS) [30] tools wrap Fortran programs for running on a grid.

Legion's data grid support is focused on extensibility, performance, and reducing the burden on the programmer [5], [45]. In terms of extensibility, there is a basic file type (basicfile) that supports the usual functions—read, write, stat, seek, etc. All other file types are derived from this type. Thus, all files can be treated as basic files and be piped into tools that expect sequential files. However, versions of basic files such as two-dimensional (2-D) files support read/write operations on columns, rows, and rectangular patches of data (both primitive types as well as "structs"). There are file types to support unstructured sparse data, as well as parallel files where the file has been broken up and decomposed across several different storage systems.

Data can be copied into the grid, in which case Legion manages the data, deciding where to place it, how many copies to generate for higer availability, and where to place those copies. Alternatively, data can be "exported" into the grid. When a local directory structure is exported into the grid it is mapped to a chosen path name in the global name space (directory structure). For example, a user can map data/sequences in his local Unix/Windows file system into/home/grimshaw/sequences using the legion_export_dir command, *legion_export_dir data/sequences/home/grimshaw/sequences*. Subsequent access from anywhere in the grid (whether read or write) are forwarded to the files in the user's Unix file system, subject to access control.

To simplify ease of use, the data grid can be accessed via a daemon that implements the NFS protocol. Therefore, the entire Legion namespace, including files, hosts, etc., can be mapped into local operating system file systems. Thus, shell scripts, Perl scripts, and user applications can run unmodified on the Legion data grid. Futhermore, the usual Unix commands such as "ls" work, as does the Windows browser.

Finally, there are the user portal and system management tools to add and remove users, add and remove hosts, and join two separate Legion grids together to create a grid of grids, etc. There is a Web-based portal interface for access to Legion [8], as well as a system status display tool that gathers information from system-wide metadata collections and makes it available via a browser (see Fig. 2 below for a screen shot). The Web-based portal (Figs. 3–6) allows an alternative, graphical interface to Legion. Using this interface, a user could submit an Amber job (a three-dimensional (3-D) molecular modeling code) to National Partnership for Advanced Computational Infrastructure (NPACI)-Net (see Section V) and not care where it executes at all. In Fig. 4, we show the portal view of the intermediate output, where the user can copy files out of the running simulation, and in which a Chime plug-in is being used to display the intermediate results.

In Fig. 5, we demonstrate the Legion job status tools. Using these tools the user can determine the status of all of the jobs that they have started from the Legion portal—and access their results as needed.

In Fig. 6, we show the portal interface to the underlying Legion accounting system. We believed from very early on that grids must have strong accounting or they will be subject to the classic tragedy of the commons, in which everyone is willing to use grid resources, yet no one is willing to provide them. Legion keeps track of who used which resource (CPU, application, etc.), starting when, ending when, with what exit status, and with how much resource consumption. The data is loaded into a relational DBMS and various reports can be generated. (An LMU is a "Legion Monetary Unit," typically, one CPU second normalized by the clock speed of the machine.)

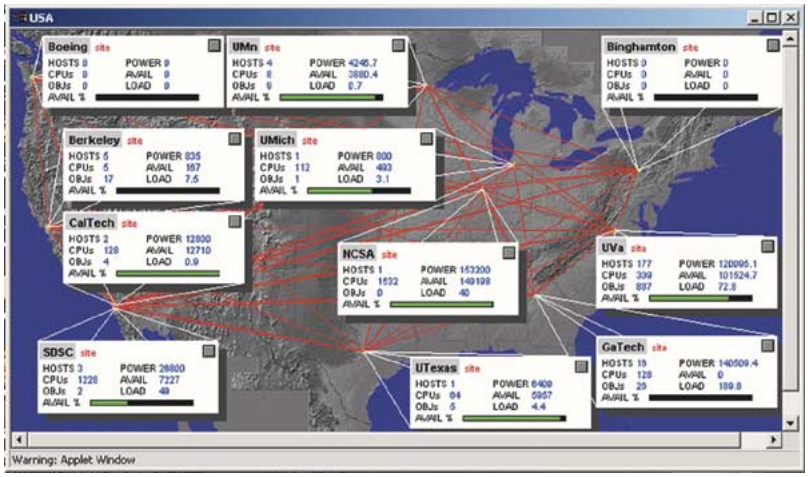

Fig. 2. Legion system monitor running on NPACI-Net in 2000 with the "USA" site selected. Clicking on a site opens a window for that site, with the individual resources listed. Sites can be composed hierarchically. Those resources in turn can be selected, and then individual objects are listed. "POWER" is a function of individual CPU clock rates and the number and type of CPUs. "AVAIL" is a function of CPU power and current load; it is what is available for use.

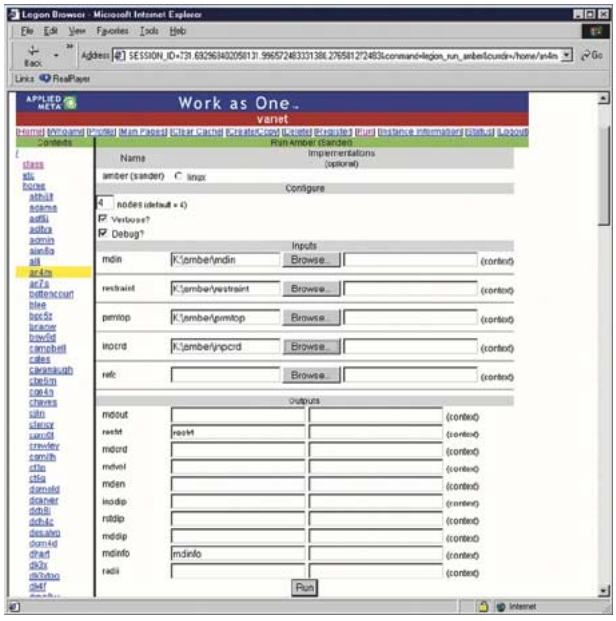

Fig. 3. Job submission window for Amber using the Legion Web portal.

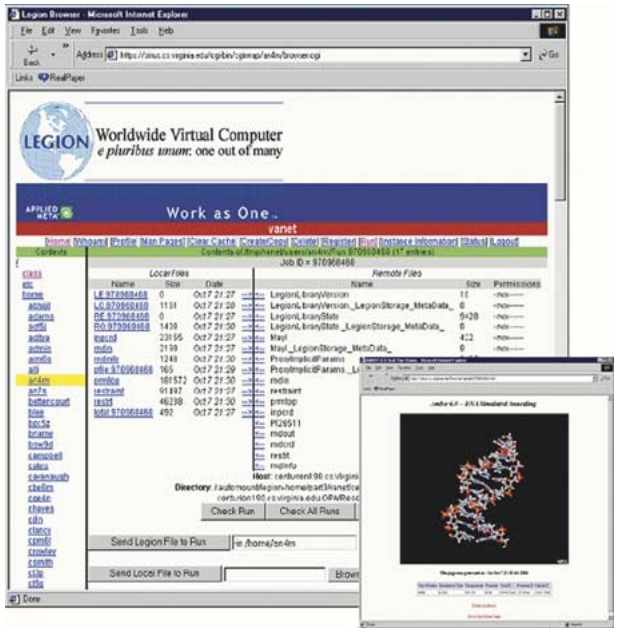

Fig. 4. Chime plugin displays updated molecule and application status.

V. Legion to Avaki—The Path of Commercialization

Legion was born in late 1993 with the observation that dramatic changes in WAN bandwidth were on the horizon. In addition to the expected vast increases in bandwidth, other changes such as faster processors, more available memory, more disk space, etc. were expected to follow in the usual way as predicted by *Moore's Law*. Given the dramatic changes in bandwidth expected, the natural question was, how will this bandwidth be used? Since not just bandwidth will change, we generalized the question to, "Given the expected changes in the physical infrastructure—what sorts of applications will people want, and given that, what is the system software infrastructure that will be needed to support those applications?" The Legion project was born with the determination to build, test, deploy, and ultimately transfer to industry, a robust, scalable grid computing software infrastructure. We followed the classic design paradigm of first determining requirements, then completely designing the system architecture on paper after numerous design meetings, and finally, after a year of design work, coding. We made a decision to write from scratch rather than extend and modify an existing system, Mentat, that we had been using as a prototype. We felt that only by starting from scratch could we ensure adherence to our architectural principles. First funding was obtained in early 1996, and the first line of Legion code was written in June 1996.

By November 1997, we were ready for our first deployment. We deployed Legion at the University of

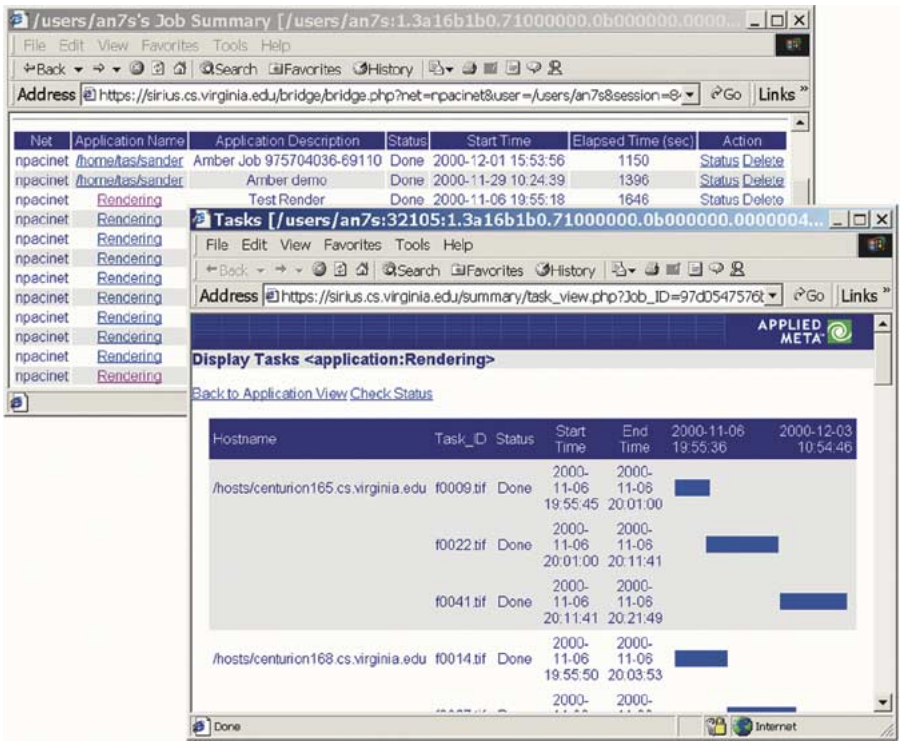

Fig. 5. Legion job status tools accessible via the Web portal.

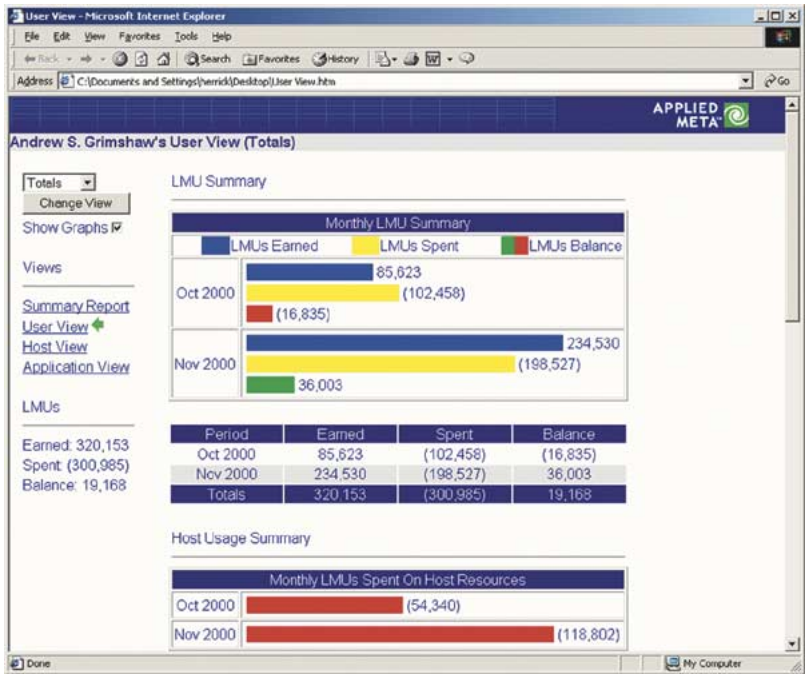

Fig. 6. Legion accounting tool. Units are normalized CPU seconds. Displays can be organized by user, machine, site, or application.

Virginia, Charlottesville; the San Diego Supercomputer Center (SDSC), San Diego; the National Center for Supercomputing Applications (NCSA); and the University of California, Berkeley, for our first large-scale test and demonstration at Supercomputing 1997. In the early months, keeping the mean time between failures (MTBF) over 20 h under continuous use was a challenge. This is when we learned several valuable lessons. For example, we learned that the world is not "fail-stop." While we intellectually knew this, it was really brought home by the unusual failure modes of the various hosts in the system.

By November 1998, we had solved the failure problems and our MTBF was in excess of one month, and heading toward three months. We again demonstrated Legion—now on what we called NPACI-Net. NPACI-Net consisted of hosts at the University of Virginia; SDSC; the California

Institute of Technology, Pasadena; the University of California, Berkeley; Indiana University, Bloomington; NCSA; the University of Michigan, Ann Arbor; Georgia Tech, Atlanta; Tokyo Institute of Technology, Tokyo, Japan; and Vrije Universiteit, Amsterdam, The Netherlands. By that time, dozens of applications had been ported to Legion from areas as diverse as materials science, ocean modeling, sequence comparison, molecular modeling, and astronomy. NPACI-Net grew through 2003 with additional sites such as the University of Minnesota, Duluth; the University of Texas, Austin; the State University of New York, Binghamton; and the Pittsburgh Supercomputing Center (PSC). Supported platforms included Windows 2000, the Compaq iPaq, the T3E and T90, IBM SP-3, Solaris, Irix, HPUX, Linux, True 64 Unix, and others.

From the beginning of the project, a "technology transfer" phase had been envisioned in which the technology would be moved from academia to industry. We felt strongly that grid software would move into mainstream business computing only with commercially supported software, help lines, customer support, services, and deployment teams. In 1999, Applied MetaComputing was founded to carry out the technology transition. In 2001, Applied MetaComputing raised $16 million in venture capital and changed its name to AVAKI [32]. The company acquired legal rights to Legion from the University of Virginia and renamed Legion to "Avaki." Avaki was released commercially in September 2001.

VI. Lessons Learned

Traveling the path from an academic project to a commercial product taught us many lessons. Very quickly we discovered that requirements in the commercial sector are remarkably different from those in academic and government sectors. These differences are significant even when considering research arms of companies involved in bioinformatics or pharmaceuticals (drug discovery). Whereas academic projects focus on issues like originality, performance, security, and fault tolerance, commercial products must address reliability, customer service, risk-aversion, and return on investment [98]. In some cases, these requirements dovetail nicely—for example, addressing the academic requirement of fault tolerance often addresses the commercial requirement of reliability. However, in many cases, the requirements are so different as to be conflicting—for example, the academic desire to be original conflicts with the commercial tendency to be risk-averse. Importantly for technologists, the commercial sector does not share academia's interest in solving exciting or "cool" problems. Several examples illustrate the last point:

- MPI and parallel computing in general are very rare. When encountered, they are employed almost exclusively in the context of low-degree parallelism problems, e.g., four-way codes. Occasionally parallelism is buried away in some purchased application, but even then it is rare because parallel hardware, excepting high-throughput clusters, is rare.
- Interest in cross-platform or cross-site applications, MPI or otherwise, is low. The philosophy is, if the machine is not big enough to solve the problem, just buy a bigger machine instead of cobbling together machines.
- Hardware expense (and, thus, hardware savings) is less important than people expense.
- Remote visualization is of no interest. If it is a matter of hardware, more can be bought to perform visualization locally.
- Parallel schedulers and metaschedulers are of no interest. All that is desired when doing anything in the multisite compute sense is a load sharing facility. The politics of CPU sharing can become overwhelming because of "server-hugging."
- Nobody wants to write new applications to "exploit" the grid. They want their existing applications to run on the grid, preferably with no changes.
- Data is the resource that must be shared between sites and research groups. Whether the data is protein data, computer-aided design (CAD) data, or financial data, large companies have groups around the world that need each other's data in a timely, coherent fashion.
- Bandwidth is not nearly as prevalent as imagined. Typical commercial customer sites have connections ranging from T1 to 10 Mb/s. The only real exception is in the financial services sector, where we encountered very high bandwidth in the OC-12 range.

As a result, the commercial version of Legion, Avaki, was a grid product that was trimmed down significantly. Many of the features on which we had labored so hard (for example, extensible files, flexible schedulers, cross-site MPI) were discarded from the product to reduce the maintenance load and clarify the message to the customer.

Building, and more importantly running, large-scale Legion networks, taught us a number of important lessons, some relearned, some pointing to broad themes, some technological. We have divided these into three broad areas: technological lessons pertain to what features of Legion worked well and what worked poorly, grid lessons pertain to what the grid community as a whole can learn from our experience, and sociological lessons pertain to the differences in perspective between academia and industry.

A. Technological Lessons

- *Name transparency is essential*. Legion's three-layer name scheme repeatedly produced architectural benefits and was well received. One of the frequent concerns expressed about a multilevel naming scheme is that the performance can be poor. However, the lower two layers—abstract names (LOIDs) to OAs—did not suffer from poor performance because we employed aggressive caching and lazy invalidation. The top layer, which consisted of human-readable names for directory structures and metadata greatly eased access and improved the user experience. Users could name applications, data, and resources in a manner that made

sense for their application instead of using obtuse, abstract, or location-specific names.

- *Trade off flexibility for lower resource consumption*. Building on the operating systems container model, early implementations of Legion created a process for every *active* Legion object. *Active* objects were associated with a process as well as state on disk, whereas *inactive* objects had only disk state. As a result, object creation and activation were expensive operations, often involving overheads of several seconds. These overheads detracted from the performance of applications having many components or opening many files. In later versions of Legion, we expanded the container model to include processes that each managed many, perhaps thousands, of objects within a single address space. Consequently, activation and creation became much cheaper, since only a thread and some data structures had to be constructed. Hence, multiplexing objects to address spaces is necessary for all but the coarsest grain applications (e.g., applications that take over 1 min to execute) even though multiplexing results in some loss of flexibility. In general, lower flexibility is acceptable if it reduces the footprint of the product on the existing IT infrastructure.
- *Trade off bandwidth for improved latency and better performance*. We have alluded earlier that in the commercial sector bandwidth is not as prevalent as imagined. However, judiciously *increasing* bandwidth consumption can result in savings in latency and performance as well as later savings in bandwidth. Consider our experience with determining the managers of objects. In Legion, a frequent pattern is the need to determine the manager of an object. For example, if the LOID-to-OA binding of an object became stale, the manager of the object would be contacted to determine the authoritative binding. One problem with our design was that the syntax we chose for the abstract names did not include the manager name. Instead, manager names were stored in a logically global single database. This database represents a glowing hotspot. Although manager lookups tend to be relatively static and can be cached extensively, when large, systemic failures occur, large caches get invalidated all at once. When the system revives, the single database is bombarded with requests for manager lookups, becoming a drag on performance. Our design of storing manager information in a single database was predicated on there being a hierarchy of such managers—sets of managers would have their own metamanagers, sets of those would have meta-metamanagers, and so on. In practice, the hierarchy rarely went beyond three levels, i.e., objects, managers and metamanagers. In hindsight, we should have encoded manager information within a LOID itself. This way, if an object's LOID-to-OA binding was stale, the manager of the class could be looked up within the LOID itself. Encoding manager information would have increased LOID size by a few bytes, thus increasing bandwidth consumption every time a LOID went on the wire, but the savings in latency and performance would have been worth it. Moreover, when manager lookups were needed, we would have saved the bandwidth of contacting the database by looking up the same information locally.
- *Trade off consistency for continued access*. In some cases, users prefer accessing data known to be stale rather than waiting for access to current data. One example is the support for disconnected operations, which are those that are performed when a client and server are disconnected. For example, if the contents of a file are cached on a client, and the client does not care whether or not the contents are perfectly up to date, then serving cached contents even when the cache is unable to contact the server to determine consistency is a disconnected operation. For the most part, Legion did not perform disconnected operations. We were initially targeting machines that tended to be connected all of the time, or if they became disconnected, reconnected quickly with the same or a different address.[3] As a result, if an object is unreachable, we made the Legion libraries time out and throw an exception. Our rationale (grounded in academia) was that if the contents are not perfectly consistent, they are useless. In contrast, our experience (from industry) showed us that there are many cases where the client can indicate disinterest in perfect consistency, thus permitting disconnected operations.

B. General Grid Lessons

- *Fault tolerance is hard but essential*. By fault tolerance we mean both fault-detection and failure recovery. L. Lamport is said to have once quipped that "a distributed system is a system where a machine I've never heard of fails and I can't get any work done." As the number of machines in a grid grows from a few, to dozens, to hundreds at dozens of sites, the probability that there is a failed machine or network connection increases dramatically. It follows that when a service is delivered by composing several other services, possibly on several different machines, the probability that it will complete decreases as the number of machines increases. Many of the existing fault-tolerance algorithms assume components, e.g., machines, operate in a "fail-stop" mode wherein it either performs correctly or does nothing. Unfortunately, real systems are not fail-stop. We found that to make Legion highly reliable required constant attention and care to handling both failure and time-out cases.
- *A powerful event/exception management system is necessary*. Related to the fault-tolerance discussion above is the need for a powerful event notification system.

[3]Legion does handle devices that are disconnected and reconnected, perhaps resulting in a changed IP address. An object periodically checks its IP address using methods in its IP communication libraries. If the address changes, the object raises an event that is caught higher up in the protocol stack, causing the object to reregister its address with its class. Thus, laptops that export services or data can be moved around.

Without such a system the result is a series of *ad hoc* decisions and choices, and an incredibly complex problem of determining exactly what went wrong and how to recover. We found that traditional publish/subscribe did not suffice because it presumes you know the set of services or components that might raise an event you need to know about. Consider a service that invokes a deep call chain of services on your behalf and an event of interest that occurs somewhere down the call chain. An object must register or subscribe to an event not just in one component, but in the transitive closure of all components it interacts with, that too for the duration of one call only. In other words, beyond the usual publish/subscribe, a throw-catch mechanism that works across service boundaries is needed.

- *Keep the number of "moving parts" down*. This lesson is a variant of the "Keep It Simple, Stupid" (KISS) principle. The larger the number of services involved in the realization of a service the slower it will be (because of the higher cost of interservice communication, including retries and timeouts). Also, the service is more likely to fail in the absence of aggressive fault-detection and recovery code. Obvious as this lesson may seem, we relearned it several times over.
- *Debugging wide-area programs is difficult, but simple tools help a lot*. If there are dozens to hundreds of services used in the execution of an application, keeping track of all of the moving parts can be difficult, particularly when an application error occurs. Two Legion features greatly simplified debugging. First, shared global console or "TTY" objects could be associated with a shell in Unix or Windows, and a Legion implicit parameter[4] set to name the TTY object. All Legion applications that used the Legion libraries checked for this parameter, and if set, redirected *stdout* and *stderr* to the *legion_tty*. In other words, all output from all components wherever they executed would be displayed in the window to which the TTY object was attached. Second, a log object, also named by an implicit parameter, could be set, enabling copies of all message traffic to be directed to the log object. The log could then be examined and replayed against Legion objects inside a traditional debugger [12].

C. Sociological Lessons

- *Technology must be augmented with service*. Solving the technical issues of data transport, security, etc., is not enough to deploy a large-scale system across organizations. The more challenging problem is to win the hearts and minds of the IT staff, e.g., system administrators and firewall managers. We found that even if senior management at an organization mandated joining a grid, administrators, who rightly or wrongly possess a sense of proprietorship over resources, can, and will, do a "slow roll" in which nothing ever quite gets done. Thus, you must work with them and get them to think it is their idea, because once they buy in, progress is rapid. In order to work with them, a services staff comprising technically competent personnel who also have strong people skills is valuable. Services personnel sometimes get a "poor cousin" treatment from technologists; we feel such an attitude is unfair, especially if the services staff brings strong engineering skills in addition to required human skills.
- *Marketing matters*. It is insufficient merely to have more usable and robust software. One must also "market" one's ideas and technology to potential users, as well as engage in "business development" activities to forge alliances with opinion leaders. Without such activities, uptake will be slow, and more importantly, users will commit to other solutions even if they are more difficult to use. That marketing matters is well known in industry and the commercial sector. What came as a surprise to us is how much it matters in the academic "market" as well.
- *End users are heterogeneous*. Many people talk about "end users" as if they are alike. We found four classes of end users: applications users who do not write code but use applications to get some job done; applications developers who write the applications; systems administrators who keep the machines running; and IT management whose mission it is to provide IT services in support of the enterprise. These four groups have very different objectives and views on what are the right tradeoffs. Do not confuse the groups.
- *Changing status quo is difficult*. Applications developers do not want to change anything—their code or yours. We found that most applications developers do not want to change their code to exploit some exciting feature or library that the grid provides. They want one code base for both their grid and nongrid versions. Furthermore, even "#ifdefs" are frowned upon. From their perspective, modifications to support grids are very risky; they have enough trouble just adding new application-specific features to their code.
- *Reflection is a nice computer science concept, but of little interest to application developers*. Legion is a reflective system, meaning that the internal workings are visible to developers if they wish to change the behavior of the system. Most aspects of Legion can be replaced with application-specific implementations. For example, the default scheduler can be replaced with a scheduler that exploits application properties, or the security module (MayI) can be replaced with a different access control policy. We found that *nobody* outside the Legion group wanted to write their own implementations of these replaceable components. Instead, we found that reflection is primarily of use to system developers to customize features for different communities (e.g., Kerberos-based authentication for DoD MSRCs).
- *Good defaults are critical*. If nobody wants to change the implementation of a service—then the default behavior of the service needs to be both efficient and do

[4]Implicit parameters are ⟨name, value⟩ pairs that are propagated in the calling context, and are the grid analogue to Unix environment variables.

a "good" job. For example, the default scheduler must not be too simple. You cannot count on people overriding defaults. Instead, they will just think that it is slow or that it makes poor choices.

VII. Summary

The challenges encountered in grid computing are enormous. To expose most programmers to the full complexity of writing a robust, secure grid application is to invite either failure or delays. Further, when using low-level tools, the duplication of effort incurred as each application team tackles the same basic issues is prohibitive.

Much like the complexity and duplication of effort encountered in early single CPU systems was overcome by the development of operating systems, so to can grid computing be vastly simplified via a grid operating system or virtual machine that abstracts the underlying physical infrastructure from the user and programmer.

The Legion project was begun in 1993 to provide just such a grid operating system. The result was a complete, extensible, fully integrated grid operating system by November 2000. NPACI-Net, the largest deployed Legion system in terms of number of hosts, had its peak at over a dozen sites on three continents with hundreds of independent hosts, and over four thousand processors. Legion provided a high level of abstraction to end users and developers alike, allowing them to focus on their applications and science—and not on the complex details. The result was over 20 different applications running on Legion—most of which required no changes to execute in Legion.[5]

Acknowledgment

Any project of Legion's scope is the result of a strong team working together. The authors were fortunate enough to have a tremendous group of people working on Legion over the last decade. The authors' thanks go out to all of them: N. Beekwilder, S. Chapin, L. Cohen, A. Ferrari, J. French, K. Holcomb, M. Humphrey, M. Hyatt, L. -J. Jin, J. Karpovich, D. Katramatos, D. Kienzle, J. Kingsley, F. Knabe, M. Lewis, G. Lindahl, M. Morgan, A. Nguyen-Tuong, S. Parsons-Wells, B. Spangler, T. Spraggins, E. Stackpole, C. Viles, M. Walker, C. Wang, E. West, B. White, and B. Wulf.

References

[1] S. Wells, *Legion 1.8 Basic User Manual*. Charlottesville, VA: Dept. Comput. Sci., Univ. Virginia, 2003.

[2] ——, *Legion 1.8 System Administrator Manual*. Charlottesville, VA: Dept. Comput. Sci., Univ. Virginia, 2003.

[3] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw, "Grid resource management in Legion," in *Grid Resource Management: State of the Art and Future Trends*, J. Nabrzyski and J. M. Schopf, Eds. Norwell, MA: Kluwer, 2003.

[4] S. J. Chapin *et al.*, "Resource management in Legion," *J. Future Gener. Comput. Syst.*, vol. 15, pp. 583–594, 1999.

[5]Globus, Globus Toolkit, GGF, Legion, Global Grid Forum, PBS, LSF, Avaki, Avaki Data Grid, ADG, LoadLeveler, Codine, SGE, IBM, Sun, DCE, MPI, CORBA, OMG, SSL, OpenSSL, GSS-API, MDS, SOAP, XML, WSDL, Windows (NT, 2000, XP), DCE, J2EE, NFS, AIX, and Kerberos are all trademarks or service marks of their respective holders.

[5] B. White *et al.*, "LegionFS: A secure and scalable file system supporting cross-domain high-performance applications," presented at the Supercomputing Conf. '01, Denver, CO.

[6] S. J. Chapin *et al.*, "A new model of security for metasystems," *J. Future Gener. Comput. Syst.*, vol. 15, pp. 713–722, 1999.

[7] D. Katramatos *et al.*, "JobQueue: A computational grid-wide queuing system," in *Lecture Notes in Computer Science, Grid Computing—GRID 2001*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2242, pp. 99–110.

[8] A. Natrajan *et al.*, "The Legion grid portal," *Concurrency Comput. Pract. Exper. (Special Issue on Grid Computing Environments)*, vol. 14, no. 13–15, pp. 1365–1394, 2002.

[9] M. J. Lewis *et al.*, "Support for extensibility and site autonomy in the Legion grid system object model," *J. Parallel Distrib. Comput.*, vol. 63, pp. 525–538, 2003.

[10] M. J. Lewis and A. S. Grimshaw, "The core Legion object model," presented at the Symp. High Performance Distributed Computing (HPDC-5), Syracuse, NY, 1996.

[11] C. L. Viles *et al.*, "Enabling flexibility in the Legion run-time library," presented at the Int. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, NV.

[12] M. Morgan, "Post mortem debugger for Legion," M.S. thesis, Univ. Virginia, Charlottesville, 1999.

[13] A. Nguyen-Tuong and A. S. Grimshaw, "Using reflection for incorporating fault-tolerance techniques into distributed applications," *Parallel Process. Lett.*, vol. 9, no. 2, pp. 291–301, 1999.

[14] A. Nguyen-Tuong, "Integrating fault-tolerance techniques into grid applications," Ph.D. dissertation, Dept. Comput. Sci., Univ. Virginia, Charlottesville, 2000.

[15] J. F. Karpovich, A. S. Grimshaw, and J. C. French, "Extensible file systems (ELFS): An object-oriented approach to high performance file I/O," presented at the ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications '94, Portland, OR.

[16] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett, "Exploiting data-flow for fault-tolerance in a wide-area parallel system," in *Proc. 15th Int. Symp. Reliable and Distributed Systems*, 1996, pp. 1–11.

[17] W. Wulf, C. Wang, and D. Kienzle, "A new model of security for distributed systems," Dept. Comput. Sci., Univ. Virginia, Charlottesville, Tech. Rep. CS-95-34, 1995.

[18] A. J. Ferrari *et al.*, "A flexible security system for metacomputing environments," presented at the 7th Int. Conf. High-Performance Computing and Networking Europe (HPCN'99), Amsterdam, The Netherlands, 1999.

[19] M. Humphrey *et al.*, "Accountability and control of process creation in metasystems," presented at the 2000 Network and Distributed Systems Security Conf. (NDSS'00), San Diego, CA.

[20] L. Smarr and C. E. Catlett, "Metacomputing," *Commun. ACM*, vol. 35, no. 6, pp. 44–52, 1992.

[21] A. S. Grimshaw *et al.*, "Metasystems," *Commun. ACM*, vol. 41, no. 11, pp. 46–55, 1998.

[22] A. S. Grimshaw *et al.*, "Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 21, no. 3, pp. 257–270, 1994.

[23] A. S. Grimshaw, "Enterprise-wide computing," *Science*, vol. 256, pp. 892–894, 1994.

[24] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA: Morgan Kaufmann, 1999.

[25] F. Berman, G. C. Fox, and A. J. G. Hey, *Grid Computing: Making the Global Infrastructure a Reality*, ser. Wiley Series in Communication Networking and Distributed Systems. New York: Wiley, 2003.

[26] A. Natrajan *et al.*, "Studying protein folding on the grid: Experiences using CHARMM on NPACI resources under Legion," *Concurrency Comput. Pract. Exper. (Special Issue on Grid Computing Environments 2003)*, vol. 16, no. 4, pp. 385–397, 2004.

[27] A. Natrajan, M. Humphrey, and A. Grimshaw, "Capacity and capability computing using Legion," presented at the 2001 Int. Conf. Computational Science, San Francisco, CA, 2001.

[28] A. Natrajan, M. Humphrey, and A. S. Grimshaw, "The Legion support for advanced parameter-space studies on a grid," *Future Gener. Comput. Syst.*, vol. 18, pp. 1033–1052, 2002.

[29] ——, "Grids: Harnessing geographically-separated resources in a multi-organizational context," presented at the 15th Annu. Symp. High Performance Computing Systems and Applications, Windsor, ON, Canada, 2001.

[30] A. Ferrari and A. Grimshaw, "Basic Fortran support in Legion," Dept. Comput. Sci., Univ. Virginia, Charlottesville, Tech. Rep. CS-98-11, 1998.

[31] M. Humphrey *et al.*, "Legion MPI: High performance in secure, cross-MSRC, cross-architecture MPI applications," presented at the 2001 DoD HPC Users Group Conf., Biloxi, Mississippi, 2001.

[32] Avaki Corp. [Online]. Available: http://www.avaki.com/

[33] S. J. Chapin, D. Katramatos, J. F. Karpovich, and A. S. Grimshaw *et al.*, "The Legion resource management system ," in *Lecture Notes in Computer Science, Job Scheduling Strategies for Parallel Processing*. Heidelberg, Germany: Springer-Verlag, 1999, vol. 1659, pp. 162–178.

[34] A. J. Ferrari, S. J. Chapin, and A. S. Grimshaw, "Heterogeneous process state capture and recovery through process introspection," *Cluster Comput.*, vol. 3, no. 2, pp. 63–73, 2000.

[35] A. Grimshaw, "Meta-systems: An approach combining parallel processing and heterogeneous distributed computing systems," presented at the 6th Int. Parallel Processing Symp. Workshop Heterogeneous Processing, Beverly Hills, CA, 1992.

[36] A. S. Grimshaw and W. A. Wulf, "Legion—A view from 50 000 feet," in *Proc. 5th IEEE Int. Symp. High Performance Distributed Computing*, 1996, p. 89.

[37] ——, "Legion flexible support for wide-area computing," presented at the 7th ACM SIGOPS Eur. Workshop, Connemara, Ireland, 1996.

[38] ——, "The Legion vision of a worldwide virtual computer," *Commun. ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[39] A. S. Grimshaw *et al.*, "Campus-wide computing: Early results using Legion at the university of Virginia," *Int. J. Supercomput. Appl.*, vol. 11, no. 2, pp. 129–143, 1997.

[40] A. S. Grimshaw *et al.*, "Wide-area computing: Resource sharing on a large scale," *IEEE Computer*, vol. 32, no. 5, pp. 29–37, May 1999.

[41] A. S. Grimshaw *et al.*, "Architectural support for extensibility and autonomy in wide-area distributed object systems," presented at the 2000 Network and Distributed Systems Security Conf. (NDSS'00), San Diego, CA.

[42] A. S. Grimshaw *et al.*, "From Legion to Avaki: The persistence of vision," in *Grid Computing: Making the Global Infrastructure a Reality*. ser. Wiley Series in Communication Networking and Distributed Systems, F. Berman, G. Fox, and T. Hey, Eds. New York: Wiley, 2003.

[43] A. Grimshaw, "Avaki data grid—Secure transparent access to data," in *Grid Computing: A Practical Guide to Technology And Applications*, A. Abbas, Ed. Hingham, MA: Charles River Media, 2003.

[44] A. S. Grimshaw, M. A. Humphrey, and A. Natrajan, "A philosophical and technical comparison of Legion and globus," *IBM J. Res. Develop.*, vol. 48, no. 2, pp. 233–254, 2004.

[45] B. White, A. Grimshaw, and A. Nguyen-Tuong, "Grid based file access: The Legion I/O model," presented at the Symp. High Performance Distributed Computing (HPDC-9), Pittsburgh, PA, 2000.

[46] B. Clarke and M. Humphrey, "Beyond the "Device as portal": Meeting the requirements of wireless and mobile devices in the Legion grid computing system," presented at the 2nd Int. Workshop Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing (associated with IPDPS 2002), Ft. Lauderdale, FL, 2002.

[47] H. Dail *et al.*, "Application-aware scheduling of a magnetohydrodynamics application in the Legion metasystems," presented at the Int. Parallel Processing Symp. Workshop Heterogeneous Processing, Cancun, Mexico, 2000.

[48] E. D. Lazowska *et al.*, "The architecture of the Eden system," in *Proc. 8th Symp. Operating System Principles*, 1981, pp. 148–159.

[49] G. R. Andrews and F. B. Schneider, "Concepts and notions for concurrent programming," *ACM Comput. Surv.*, vol. 15, no. 1, pp. 3–44, 1983.

[50] W. F. Applebe and K. Hansen, "A survey of systems programming languages: Concepts and facilities," *Softw. Pract. Exper.*, vol. 15, no. 2, pp. 169–190, 1985.

[51] H. Bal, J. Steiner, and A. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, 1989.

[52] R. Ben-Naten, *CORBA: A Guide to the Common Object Request Broker Architecture*. New York: McGraw-Hill, 1995.

[53] B. N. Bershad and H. M. Levy, "Remote computation in a heterogeneous environment," Dept. Comput. Sci., Univ. Washington, Seattle, Tech. Rep. 87-06-04, 1987.

[54] B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Presto: A system for object-oriented parallel programming," *Softw. Pract. Exper.*, vol. 18, no. 8, pp. 713–732, 1988.

[55] A. Black *et al.*, "Distribution and abstract types in Emerald," Univ. Washington, Seattle, Tech. Rep. 85-08-05, 1985.

[56] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. Softw. Eng.*, vol. 14, no. 2, pp. 141–154, Feb. 1988.

[57] R. Chin and S. Chanson, "Distributed object-based programming systems," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 91–127, 1991.

[58] Object Management Group, "The common object request broker: Architecture and specification,", Needham, MA, OMG Doc. No. 93.xx.yy, Revision 1.2, Draft 29, 1993.

[59] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Perform. Eval. Rev.*, vol. 6, pp. 53–68, 1986.

[60] ——, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 5, pp. 662–675, May 1986.

[61] ——, "The limited performance benefits of migrating active processes for load sharing," *Perform. Eval. Rev.*, vol. 16, pp. 63–72, 1986.

[62] ——, "Speedup versus efficiency in parallel systems," *IEEE Trans. Comput.*, vol. 38, no. 3, pp. 408–423, Mar. 1989.

[63] J. A. Feldman, "High level programming for distributed computing," *Commun. ACM*, vol. 22, no. 6, pp. 353–368, 1979.

[64] P. B. Gibbond, "A stub generator for multi-language RPC in heterogeneous environments," *IEEE Trans. Softw. Eng.* , vol. SE-13, no. 1, pp. 77–87, Jan. 1987.

[65] A. Hac, "Load balancing in distributed systems: A summary," *Perform. Eval. Rev.*, vol. 16, pp. 17–25, 1989.

[66] C. A. R. Hoare, "Monitors: An operating system structing concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[67] R. H. LeBlanc and C. T. Wilkes, "Systems programming with objects and actions," in *Proc. 7th Int. Conf. Distributed Computer Systems*, 1985, pp. 132–138.

[68] E. Levy and A. Silberschatz, "Distributed file systems: Concepts and examples," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 321–374, 1990.

[69] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 381–414, 1983.

[70] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," presented at the SIGPLAN'88 Conf. Programming Language Design and Implementation, Atlanta, GA.

[71] F. Manola *et al.*, "Distributed object management," *Int. J. Intell. Coop. Inf. Syst.*, vol. 1, no. 1, 1992.

[72] J. H. E. A. Morris, "Andrew: A distributed personal computing environment," *Commun. ACM*, vol. 29, no. 3, 1986.

[73] R. Mirchandaney, D. Towsley, and J. Stankovic, "Adaptive load sharing in heterogeneous distributed systems," *J. Parallel Distrib. Comput.*, vol. 9, pp. 331–346, 1990.

[74] O. M. Nierstrasz, "Hybrid: A unified object-oriented system," *IEEE Database Eng. Bull.*, vol. 8, no. 4, pp. 49–57, Dec. 1985.

[75] D. E. A. Notkin, "Interconnecting heterogeneous computer systems," *Commun. ACM*, vol. 31, no. 3, pp. 258–273, 1988.

[76] G. E. A. Popek, "LOCUS: A network transparent, high reliability distributed system," in *Proc. 8th Symp. Operating System Principles*, 1981, pp. 169–177.

[77] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," in *Proc. 9th ACM Symp. Operating System Prinicples*, 1983, pp. 110–119.

[78] A. S. Tanenbaum and R. V. Renesse, "Distributed operating systems," *ACM Comput. Surv.*, vol. 17, no. 4, pp. 419–470, 1985.

[79] R. N. E. A. Taylor, "Foundations for the Arcadia environment architecture," in *Proc. 3rd ACM SIGSOFT/SIGPLAN Symp. Practical Software Development*, 1989, pp. 1–13.

[80] M. M. Theimer and B. Hayes, "Heterogeneous process migration by recompilation," presented at the 11th Int. Conf. Distributed Computing Systems, Arlington, TX, 1991.

[81] B. E. A. Walker, "The LOCUS distributed operating system," presented at the 9th ACM Symp. Operating Systems Principles, Bretton Woods, NH, 1983.

[82] S. A. E. A. Yemini, "Concert: A heterogeneous high-level-language approach to heterogeneous distributed systems," in *IEEE Int. Conf. Distributed Computing Systems*, Newport, CA.

[83] B. N. E. A. Bershad, "A remote procedure call facility for interconnecting heterogeneous computer systems," *IEEE Trans. Software. Eng.*, vol. SE-13, no. 8, pp. 880–894, Aug. 1987.
[84] M. Satyanarayanan, "Scalable, secure, and highly available distributed file access," *IEEE Computer*, vol. 23, no. 5, pp. 9–21, May 1990.
[85] R. H. Campbell *et al.*, "Principles of object oriented operating system design," Dept. Comput. Sci., Univ. Illinois, Urbana, Tech. Rep. R89-1510, 1989.
[86] A. Chervenak *et al.*, "The data grid: Toward an architecture for the distributed management and analysis of large scientific datasets," *J. Netw. Comput. Appl.*, vol. 23, pp. 187–200, 2001.
[87] I. Foster *et al.*, "Software infrastructure for the I-WAY high performance distributed computing experiment," in *5th IEEE Symp. High Performance Distributed Computing*, 1997, pp. 562–571.
[88] *Globus Project*. Globus.
[89] K. Czajkowski, "A resource management architecture for metacomputing systems," in *Proc. IPPS/SPDP'98 Workshop Job Scheduling Strategies for Parallel Processing*, pp. 62–82.
[90] K. Czajkowski, I. Foster, and C. Kesselman, "Co-allocation services for computational grids," presented at the 8th IEEE Symp. High Performance Distributed Computing, Redondo Beach, CA, 1999.
[91] K. Czajkowski *et al.*, "SNAP: A protocol for negotiating service level agreements and coordinating resource management in distributed systems," in *Lecture Notes in Computer Science, Job Scheduling Strategies for Parallel Processing*, 2002, vol. 2537, pp. 153–183.
[92] K. Czajkowski *et al.*, "Agreement-based service management (WS-agreement)," Global Grid Forum, draft-ggf-graap-agreement-1, 2004.
[93] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *Int. J. Supercomput. Appl.*, vol. 11, no. 2, pp. 115–128, 1997.
[94] Legion Web pages (1997). [Online]. Available: http://www.legion.virginia.edu
[95] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable runtime support for dynamic object-oriented parallel processing," *ACM Trans. Comput. Syst.*, vol. 14, no. 2, pp. 139–170, 1996.
[96] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 388–395, Apr. 1987.
[97] G. Stoker *et al.*, "Toward realizable restricted delegation in computational grids," presented at the Int. Conf. High Performance Computing and Networking Europe (HPCN Eur. 2001), Amsterdam, The Netherlands, 2001.
[98] A. S. Grimshaw, "The ROI case for grids," *Grid Today*, vol. 1, no. 27, 2002.

Andrew S. Grimshaw received the M.S. and Ph.D. degrees from the University of Illinois, Urbana-Champaign, in 1986 and 1988, respectively.

He is Professor of Computer Science at the University of Virginia, Charlottesville, and Founder and CTO of Avaki Corp., Burlington, MA. He is the Chief Designer and Architect of Mentat and Legion. Legion is one of the earliest and largest grid computing projects in the world and was the genesis for Avaki—the realization of production grids in a commercial setting. He has published extensively on both parallel computing and grid computing and regularly speaks on grid computing around the world. His research interests include grid computing, high-performance parallel computing, heterogeneous parallel computing, operating systems, and the use of grid computing in the life sciences.

Dr. Grimshaw is currently a member of the Global Grid Forum (GGR) Steering Committee and the Architecture Area Director in the GGF. Grimshaw has served on the National Partnership for Advanced Computational Infrastructure (NPACI) Executive Committee, the DoD MSRC Programming Environments and Training (PET) Executive Committee, the CESDIS Science Council, the NRC Review Panel for Information Technology, Board on Assessment of NIST Programs, and others.

Anand Natrajan received the Ph.D. degree from the University of Virginia, Charlottesville, in 2000.

He continued at the University of Virginia as a Research Scientist with the Legion project for another two years. Currently, he is a Senior Software Engineer and Consultant on grid systems for Avaki Corp., Burlington, MA. He has written tools for scheduling, running and monitoring large numbers of legacy parameter-space jobs on a grid, and for browsing the resources of a grid. He has published several related papers as well as participated in several fora related to grids. His previous research interests included multirepresentation modeling for distributed interactive simulation. His doctoral thesis addressed the problem of maintaining consistency among multiple concurrent representations of entities being modeled. His current research focuses on harnessing the power of distributed systems for user applications. In addition to distributed systems, he is interested in computer architecture and information retrieval.