The Globus Project: A Status Report

Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Carl Kesselman
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292-6695

Abstract

The Globus project is a multi-institutional research effort that seeks to enable the construction of computational grids *providing pervasive, dependable, and consistent access to high-performance computational resources, despite geographical distribution of both resources and users. Computational grid technology is being viewed as a critical element of future high-performance computing environments that will enable entirely new classes of computation-oriented applications, much as the World Wide Web fostered the development of new classes of information-oriented applications. In this paper, we report on the status of the Globus project as of early 1998. We describe the progress that has been achieved to date in the development of the Globus toolkit, a set of core services for constructing grid tools and applications. We also discuss the Globus Ubiquitous Supercomputing Testbed (GUSTO) that we have constructed to enable large-scale evaluation of Globus technologies, and we review early experiences with the development of large-scale grid applications on the GUSTO testbed.*

1 Introduction

Advances in networking technology and computational infrastructure make it possible to construct large-scale high-performance distributed computing environments, or *computational grids* that provide dependable, consistent, and pervasive access to high-end computational resources. These environments have the potential to change fundamentally the way we think about computing, as our ability to compute will no longer be limited to the resources we currently have on hand. For example, the ability to integrate TFLOP/s computing resources on demand will allow us to integrate sophisticated analysis, image processing, and real-time control into scientific instruments such as microscopes, telescopes, and MRI machines. Or, we can call upon the resources of a nationwide *strategic computing reserve* to perform time-critical computational tasks in times of crisis, for example to perform diverse simulations as we plan responses to an oil spill.

In the past, high-performance distributed computation has been achieved on a limited scale by heroic efforts such as the CASA Gigabit testbed [26] and the I-WAY [12]. The work of ourselves and others on computational grids differs from these ground-breaking efforts in that we seek to make commonplace the integration of remote resources into a computation. To a large extent, the development of usable computational grids is hindered not by available hardware capabilities but by limitations in the software abstractions and services that are currently in use. Existing network tools are focused on supporting communication, not computation, while current distributed computing systems are not performance driven and typically are limited to client/server models of computation. Clearly, the success of computational grids will depend on the existence of grid-specific middleware that addresses the needs of computations including dynamic resource allocation, resource co-allocation, heterogeneous and dynamic computational and communication substrates, and process-oriented security.

We have been studying the problems associated with constructing usable computational grids since 1995, first in the context of the I-WAY networking experiment [12] and subsequently as part of a project called Globus. The goal of Globus is to understand application requirements for a usable grid and to develop the essential technologies required to meet these requirements. In pursuit of this goal, we have developed a research program comprising three broad activities:

- developing the basic technology and high-level tools required for computational grids;
- constructing a large-scale, prototype computational grid (i.e., testbed) using the basic technologies and tools we have developed; and

- executing realistic applications on the prototype grid, in order to evaluate the utility of our technologies and of the grid concept.

In this paper, we describe the status of the Globus project in each of these three areas, as of early 1998. This description updates the original Globus paper [13] and a subsequent project summary in [14] by providing a more complete and up-to-date description of the Globus toolkit and by reviewing early experiments with the Globus Ubiquitous Supercomputing Testbed (GUSTO) grid prototype, the largest computational grid constructed to date.

The organization of this paper is as follows. In the next section, we outline the basic architecture of the Globus system, identifying the basic principles that motivate its design. In Sections 3–7, we describe the set of basic services that constitute the Globus toolkit that underlies our approach, and in Section 8 we review some of the higher-level tools that have been constructed with this toolkit. In Section 9, we describe our experiences deploying these tools in the GUSTO grid testbed, and in Section 10 we review our experiences developing applications. We conclude the paper with a brief survey of some related work (Section 11) and a description of our future plans (Section 12).

2 Globus Overview

A central element of the Globus system is the Globus Metacomputing Toolkit, which defines the basic services and capabilities required to construct a computational grid. The design of this toolkit was guided by the following basic principles.

The toolkit comprises a set of components that implement basic services for security, resource location, resource management, communication, etc.. The services currently defined by Globus are listed in Table 1. Computational grids must support a wide variety of applications and programming models. Hence, rather than providing a uniform programming model, such as the object-oriented model defined by the Legion system [18], the Globus toolkit provides a "bag of services" from which developers of specific tools or applications can select to meet their needs.

Because services are distinct and have well-defined interfaces, they can be incorporated into applications or tools in an incremental fashion. We illustrate this mix-and-match approach to metacomputing in Sections 8 and 10, where we describe how different parallel tools and a large application can be made grid aware by incorporating different services.

The toolkit distinguishes between local services, which are kept simple to facilitate deployment, and global services, which are constructed on top of local services and may be more complex. Computational grids require that a wide range of services be supported on a highly heterogeneous mix of systems and that it be possible to define new services without changing the underlying infrastructure. An established architectural principle in such situations, as exemplified by the Internet Protocol suite [6], is to adopt a layered architecture with an "hourglass" shape (Figure 1). A simple, well-defined interface—the neck of the hourglass—provides uniform access to diverse implementations of local services; higher-level global services are then defined in terms of this interface. To participate in a grid, a local site need provide only the services defined at the neck, and new global services can be added without local changes. We discuss this organization in greater detail in Section 3.

Interfaces are defined so as to manage heterogeneity, rather than hiding it. These so-called translucent interfaces provide structured mechanisms by which tools and applications can discover and control aspects of the underlying system. Such translucency can have significant performance advantages because, if an implementation of a higher-level service can understand characteristics of the lower-level services on which the interface is layered, then the higher-level service can either control specific behaviors of the underlying service or adapt its own behavior to that of the underlying service. Translucent interfaces do not imply complex interfaces. Indeed, we will show that translucency can be provided via simple techniques, such as adding an attribute argument to the interface. We discuss these issues at greater length in Section 4, when we describe Globus communication services.

An information service is an integral component of the toolkit. Computational grids are in a constant state of flux as utilization and availability of resources change, computers and networks fail, old components are retired, new systems are added, and software and hardware on existing systems are updated and modified. It is rarely feasible for programmers to rely on standard or default configurations when building applications. Rather, applications must *discover* characteristics of their execution environment dynamically and then either *configure* aspects of system and application behavior for efficient, robust execution or *adapt* behavior during program execution. A fundamental requirement for discovery, configuration, and adaptation is an *information-rich environment* that provides pervasive and uniform access to information about the current state of the grid and its underlying components. In the Globus toolkit, a component

Table 1: Core Globus services. As of early 1998, these include only those services deemed essential for an evaluation of the Globus design philosophy on realistic applications and in medium-scale grid environments. Other services such as accounting, auditing, and instrumentation will be addressed in future work

Service	Name	Description
Resource management	GRAM	Resource allocation and process management
Communication	Nexus	Unicast and multicast communication services
Security	GSI	Authentication and related security services
Information	MDS	Distributed access to structure and state information
Health and status	HBM	Monitoring of health and status of system components
Remote data access	GASS	Remote access to data via sequential and parallel interfaces
Executable management	GEM	Construction, caching, and location of executables

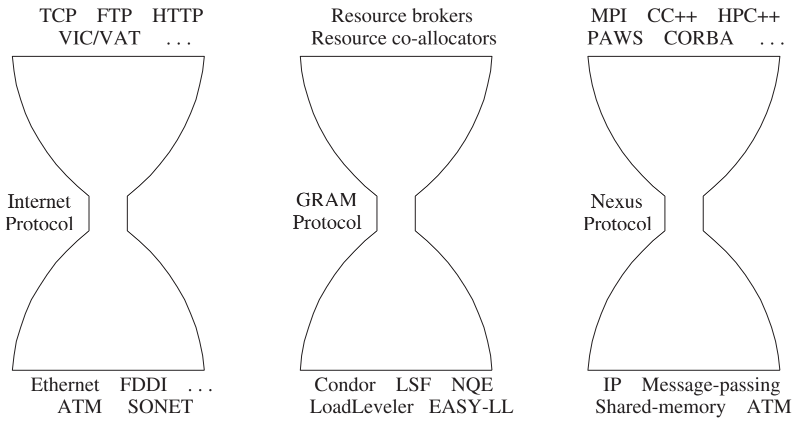

Figure 1: The hourglass principle, as applied in the Internet Protocol suite, Globus resource management services, and Globus communication services

called the Metacomputing Directory Service [9], discussed in Section 5, fulfills this role.

The toolkit uses standards whenever possible for both interfaces and implementations. We envision computational grids as supporting an important niche of applications that must co-exist with more general-purpose distributed and networked computing applications such as CORBA, DCE, DCOM, and Web-based technologies. The Internet community and other groups are moving rapidly to develop official and de facto standards for interfaces, protocols, and services in many areas relevant to computational grids. There is considerable value in adopting these standards whenever they do not interfere with other goals. Consequently, the Globus components we will describe are not, in general, meant to replace existing interfaces, but rather seek to augment them. The utility of standards is emphasized in Section 6, which describes the Globus security infrastructure.

3 Resource Management

We now dexcribe more fully the Globus components listed in Table 1. We start by considering resource management. Both this discussion and the current Globus implementation focus on the management of computational resources. Management of memory, storage, networks, and other resources is clearly also important and is being considered in current research.

Globus is a layered architecture in which high-level global services are built on top of an essential set of core local services. At the bottom of this layered architecture, the Globus Resource Allocation Manager (GRAM) provides the local component for resource management [8]. Each GRAM is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system, such as Load Sharing Facility (LSF) or Condor. For example, a single manager could provide access to the nodes of a parallel computer, a cluster of workstations, or a set of machines operating within a Condor pool [25]. Thus, a computational grid built with Globus typically contains many GRAMs, each responsible for a particular "local" set of resources.

GRAM provides a standard network-enabled interface to local resource management systems. Hence, computational grid tools and applications can express resource allocation and process management requests in terms of a standard application programming interface (API), while individual sites are not constrained in their choice of resource management tools. GRAM can currently operate in conjunction with six different local resource management tools: Network Queuing Environment (NQE), EASY-LL, LSF, LoadLeveler, Condor, and a simple "fork" daemon. Within the GRAM API, resource requests are expressed in terms of an extensible *resource specification language* (RSL); as we describe below, this language plays a critical role in the definition of global services.

GRAM services provide building blocks from which we can construct a range of global resource management strategies. Building on GRAM, we have defined the general resource management architecture [8] illustrated in Figure 2. RSL is used throughout this architecture as a common notation for expressing resource requirements. Resource requirements are expressed by an application in terms of a high-level RSL expression. A variety of *resource brokers* implement domain-specific resource discovery and selection policies by transforming abstract RSL expressions into progressively more specific requirements until a specific set of resources is identified. For example, an application might specify a computational requirement in terms of floating-point performance (MFLOPs). A high-level broker might narrow this requirement to a specific type of computer (an IBM SP2, for example), while another broker might identify a specific set of SP2 computers that can fulfill that request. At this point, we have a so-called ground RSL expression in which a specific set of GRAMs are identified.

The final step in the resource allocation process is to decompose the RSL into a set of separate resource allocation requests and to dispatch each request to the appropriate GRAM. In high-performance computations, it is often important to *co-allocate* resources at this point, ensuring that a given set of resources is available for use simultaneously. Within Globus, a *resource co-allocator* is responsible for providing this service: breaking the RSL into pieces, distributing it to the GRAMs, and coordinating the return values. Different co-allocators can be constructed to implement different approaches to the problems of allocating and managing ensembles of resources. We currently have two allocation services implemented. The first defines a simple *atomic co-allocation* semantics. If any of the requested resources are unavailable for some reason, the entire co-allocation request fails. In practice, this strategy has proven to be too inflexible in many situations. Based on this experience, we have implemented a second co-allocator, which allows components of the submitted RSL expression to be modified until the application or broker issues a commit operation.

Notice that a consequence of the Globus resource management architecture is that resource and com-

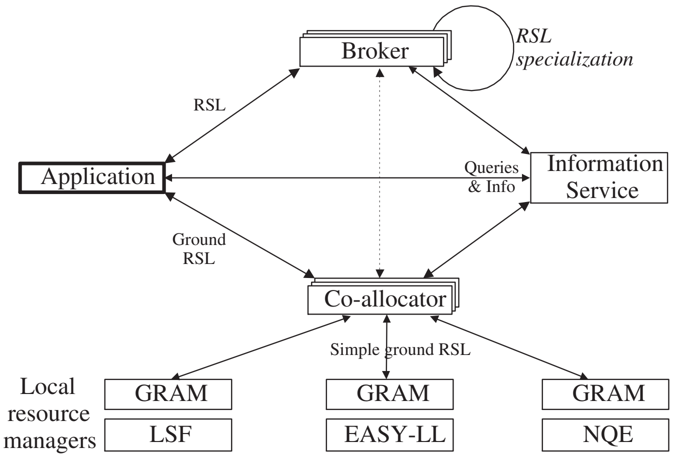

Figure 2: The Globus resource management architecture, showing how RSL specifications pass between application, resource brokers, resource co-allocators, and local managers (GRAMs). Notice the central role of the information service.

putation management services are implemented in a hierarchical fashion. An individual GRAM supports the creation and management of a set of processes, or Globus job, on a set of local resources. A computation created by a global service may then consist of one or more jobs, each created by a request to a GRAM and managed via management functions implemented by that GRAM.

This discussion of Globus resource management services illustrates how simple local services, if appropriately designed, can be used to support a rich set of global functionality.

4 Communication

Communication services within the Globus toolkit are provided by the Nexus communication library [15]. As illustrated in Figure 1, Nexus defines a relatively low-level communication API that is then used to support a wide range of higher-level communication libraries and languages, based on programming models as diverse as message passing, as in the Message Passing Interface (MPI) [10]; remote procedure call, as in CC++ [5]; striped transfer, as in the Parallel Application Workspace (PAWS); and distributed database updates for collaborative environments, as in CAVERNsoft. Nexus communication services are also used extensively in the implementation of other Globus modules.

The communication needs of computational grid applications are diverse, ranging from point-to-point message passing to unreliable multicast communication. Many applications, such as instrument control and teleimmersion, use several modes of communication simultaneously. In our view, the Internet Protocol does not meet these needs: its overheads are high, particularly on specialized platforms such as parallel computers; the TCP streaming model is not appropriate for many interactions; and its interface provides little control over low-level behavior. Yet traditional high-performance computing communication interfaces such as MPI do not provide the rich range of communication abstractions that grid applications will require. Hence, we define an alternative communication interface designed to support the wide variety of underlying communication protocols and methods encountered in grid environments and to provide higher-level tools with a high degree of control over the mapping between high-level communication requests and underlying protocol operations. We call this interface Nexus [15, 11].

Communication in Nexus is defined in terms of two basic abstractions. A *communication link* is formed by binding a communication startpoint to a communication endpoint (Figure 4); a communication operation is initiated by applying a *remote service request* (RSR) to a startpoint. This one-sided, asynchronous remote procedure call transfers data from the startpoint to the associated endpoint(s) and then integrates the data into the process(es) containing the endpoint(s) by invoking a function in the process(es). More than one startpoint can be bound to an endpoint and vice versa,

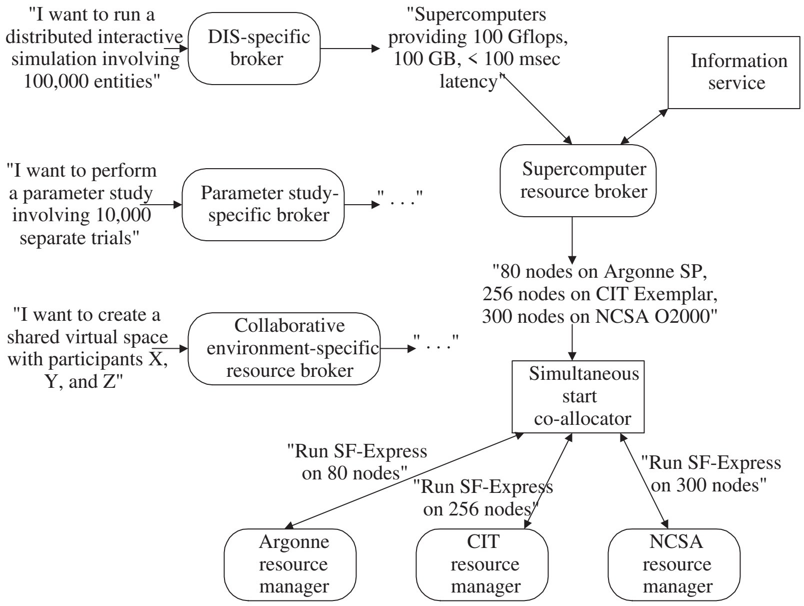

Figure 3: This view of the Globus resource management architecture shows how different types of broker can participate in a single resource request

allowing for the construction of complex communication structures.

The communication link/RSR communication model can be mapped into many different communication methods, each with potentially different performance characteristics [11]. Communication methods include not only communication protocols, but also other aspects of communication such as security, reliability, quality of service, and compression. By associating *attributes* with a specific startpoint or endpoint, an application can control the communication method used on a per-link basis. For example, an application in which some communications must be reliable while others require low latencies can establish two links between two processes, with one configured for reliable—and potentially high-latency—communication and the other for low-latency unreliable communication.

High-level selection and configuration of low-level methods is useful only if the information required to make intelligent decisions is readily available. Within Globus, MDS (discussed in Section 5) maintains a wealth of dynamic information about underlying communication networks and protocols, including network connectivity, protocols supported, and network bandwidth and latency. Applications, tools, and higher-level libraries can use this information to identify available methods and select those best suited for a particular purpose.

High-level management of low-level communication methods has many uses. For example, an MPI implementation layered on top of Nexus primitives can not only select alternative low-level protocols (e.g., message passing, IP, or shared memory) based on network topology and the location of sender and receiver [10], but can simultaneously apply selective use of encryption based on the source and destination of a message. The ability to attach network quality of service specifications to communication links is also useful.

Nexus illustrates how Globus services use translucent interfaces to allow applications to manage rather than hide heterogeneity. An application or higher-level library can express all operations in terms of a single uniform API; the resulting programs are portable across, and will execute efficiently on, a wide variety of computing platforms and networks. To this extent Nexus, like other Globus services, hides heterogeneity. However, in situations where performance is critical, properties of low-level services can be discovered. The higher-level library or application can then either adapt its behavior appropriately or use a control API to manage just how high-level behavior is implemented: for example, by specifying that it is ac-

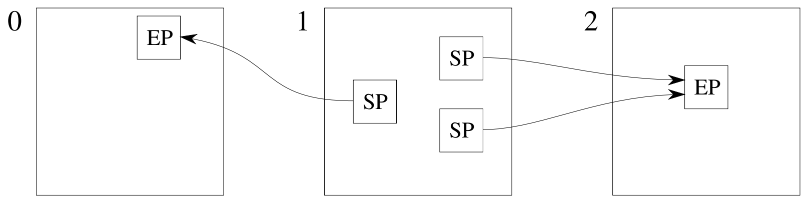

Figure 4: Nexus communication mechanisms. The figure shows three processes and three communication links. Three startpoints in process 1 reference endpoints in processes 0 and 2.

ceptable to use an unreliable communication protocol for a particular set of communications.

5 Information

The dynamic nature of grid environments means that toolkit components, programming tools, and applications must be able to adapt their behavior in response to changes in system structure and state. The Globus Metacomputing Directory Service (MDS) [9] is designed to support this type of adaptation by providing an information-rich environment in which information about system components is always available. MDS stores and makes accessible information such as the architecture type, operating system version and amount of memory on a computer, network bandwidth and latency, available communication protocols, and the mapping between IP addresses and network technology.

MDS provides a suite of tools and APIs for discovering, publishing, and accessing information about the structure and state of a computational grid. As in other Globus components, official or de facto standards are used in MDS whenever possible. In this case, the standards in question are the data representation and API defined by the Lightweight Directory Access Protocol (LDAP) [22], which together provide a uniform, extensible representation for information about grid components. LDAP defines a hierarchical, tree-structured name space called a *directory information tree* and is designed as a distributed service: arbitrary subtrees can be associated with distinct servers. Hence, the local service required to support MDS is exactly an LDAP server (or a gateway to another LDAP server, if multiple sites share a server), plus the utilities used to populate this server with up-to-date information about the structure and state of the resources within that site. The global MDS service is simply the ensemble of all these servers.

An information-rich environment is more than just mechanisms for naming and disseminating information: it also requires agents that produce useful information and components that access and use that information. Within Globus, both these roles are distributed over every system component—and potentially over every application. Every Globus service is responsible for producing information that users of that service may find useful, and for using information to enhance its flexibility and performance. For example, each local resource manager (Section 3) incorporates a component called the *GRAM reporter* responsible for collecting and publishing information about the type of resources being managed, their availability, and so forth. Resource brokers use this and other information for resource discovery.

6 Security

Security in computational grids is a multifaceted issue, encompassing authentication, authorization, privacy, and other concerns. While the basic cryptographic algorithms that form the basis of most security systems—such as public key cryptography—are relatively simple, it is a challenging task to use these algorithms to meet diverse security goals in complex, dynamic grid environments, with large and dynamic sets of users and resources and fluid relationships between users and resources.

The Globus security infrastructure developed for the initial Globus toolkit focuses on just one problem, *authentication*: the process by which one entity verifies the identity of another. We focus on authentication because it is the foundation on which other security services, such as authorization and encryption, are built; these issues will be addressed in future work.

Authentication solutions for computational grids must solve two problems not commonly addressed by standard authentication technologies. The first problem that must be addressed by a grid authentication solution is support for *local heterogeneity.* Grid resources are operated by a diverse range of entities, each defining a different *administrative domain*. Each domain will have its own requirements for authentication and authorization, and consequently, domains will have different local security solutions, mechanisms, and policies, such as one-time passwords, Kerberos [29], and Secure Shell. We will have limited ability to change these administrative decisions, and any security solution must confront this heterogeneity.

The second problem facing security solutions for computational grids is the need to support *N-way security contexts*. In traditional client-server applications, authentication involves just a single client and a single server. In contrast, a grid computation may acquire, start processes on, and release many resources dynamically during its execution. These processes will communicate by using a variety of mechanisms, including unicast and multicast. These processes form a single, fully connected logical entity, although low-level communication connections (e.g., TCP/IP sockets) may be created and deleted dynamically during program execution. A security solution for a computational grid must enable the establishment of a security relationship between any two processes in a computation.

A first important step in the design of a security architecture, often overlooked, is to define a security policy: that is, to provide a precise definition of what it means for the system in question to be secure. This policy identifies what components are to be protected and what these components are to be protected against, and defines security operations in terms of abstract algorithms. The policy defined for Globus is shaped by the need to support N-way security contexts and local heterogeneity. The policy specifies that a user authenticate just once per computation, at which time a credential is generated that allows processes created on behalf of the user to acquire resources, and so forth, without additional user intervention. Local heterogeneity is handled by mapping a user's *Globus identity* into local user identities at each resource.

One important aspect of the security policy defined by Globus is that encrypted channels are not used. Globus is intended to be used internationally, and several countries (including the United States and France) have restrictive laws with respect to encryption technology. The Globus policy relies only on digital signature mechanisms, which are more easily exportable from the United States.

The Globus security policy is implemented by the Globus security infrastructure (GSI). GSI, like other Globus components, has a modular design in which diverse global services are constructed on top of a simple local service that addresses issues of local heterogeneity. As illustrated in Figure 5, the local security service implements a security gateway that maps authenticated Globus credentials into locally recognized credentials at a particular site: for example, Kerberos tickets, or local user names and passwords. A benefit of this approach is that we do not require "group" accounts and so can preserve the integrity of local accounting and auditing mechanisms.

The internal design of GSI emphasizes the important role that standards have to play in the definition of grid services and toolkits. Several of the problems that GSI is designed to solve, namely, support for different local mechanisms and N-way security contexts, are not supported by any existing system. Nevertheless, GSI's ability to interoperate with other systems, to achieve independence from low-level mechanisms, and to leverage existing code is enhanced by coding all security algorithms in terms of the Generic Security Service (GSS) standard [24]. GSS defines a standard procedure and API for obtaining credentials (passwords or certificates), for mutual authentication (client and server), and for message-oriented signature, encryption and decryption. GSS is independent of any particular security mechanism and can be layered on top of different security methods. To promote interoperability, the GSS standard defines how GSS functionality should be implemented on top of Kerberos and public key cryptography. GSS also defines a negotiation mechanism that allows two parties to select a mutually agreeable suite of security mechanisms, should alternatives exist.

GSI currently supports two security mechanisms, both accessible through the GSS interface. The first is a plaintext password system, which basically implements Unix `rlogin` type authentication. The plaintext implementation has the advantage of being easy to develop and debug and is not encumbered by export controls. The second mechanism uses public key cryptography and is based on the authentication protocol defined by the Secure Socket Layer (SSL) [21]. This implementation has the advantages of much stronger security and interoperability with a variety of commodity services, including LDAP and HTTP. We note that GSS supports a negotiation mechanism, which allows us to support both security mechanisms simultaneously in the Globus environment.

7 Other Globus Services

We briefly describe the other three Globus services listed in Table 1: health and status monitoring, remote access to files, and executable management.

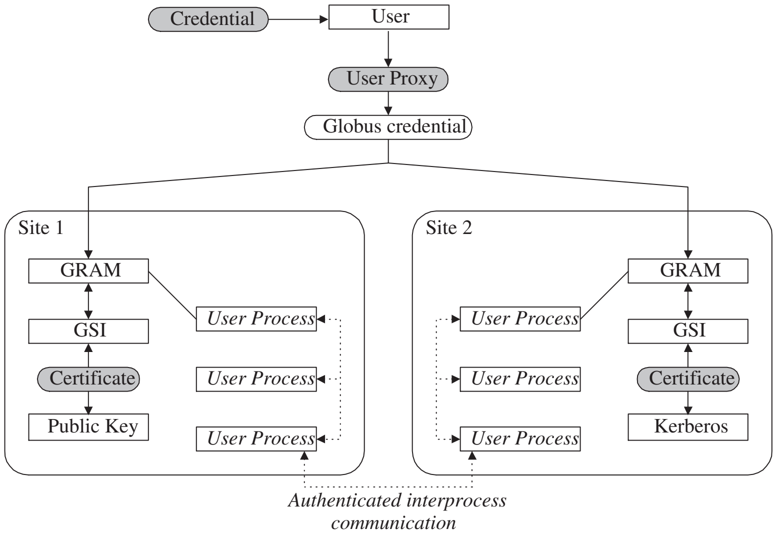

Figure 5: The Globus security infrastructure, showing its support for single sign-on and local heterogeneity

The Heartbeat Monitor (HBM) service provides simple mechanisms for monitoring the health and status of a distributed set of processes. The HBM architecture comprises a client interface and a data-collector API. The client interface allows a process to register with the HBM service, which then expects to receive regular heartbeats from the process. If a heartbeat is not received, the HBM service attempts to determine whether the process itself is faulty or whether the underlying network or computer has failed. The data-collector API allows another process to obtain information regarding the status of registered process; this information can then be used to implement a variety of fault detection and, potentially, fault recovery mechanisms. HBM mechanisms are used to monitor the status of core Globus services, such as GRAM and MDS. They can also be used to monitor distributed applications and to implement application-specific fault recovery strategies.

Access to remote files is provided by the Global Access to Secondary Storage (GASS) subsystem. This system allows programs that use the C standard I/O library to open and subsequently read and write files located on remote computers, without requiring changes to the code used to perform the reading and writing. As illustrated in Figure 6, files opened for reading are copied to a local file cache when they are opened, hence permitting subsequent read operations to proceed without communication and also avoiding repeated fetches of the same file. Reference counting is used to determine when files can be deleted from the cache. Similarly, files opened for writing are created locally and copied to their destination only when they are closed. A similar copying strategy is used in UFO [2], but our implementation does not rely on the Unix-specific `proc` file system. GASS also allows files to be opened for remote appending, in which case data is communicated to the remote file as soon as it is written; this mode is useful for log files, for example. In addition, GASS supports remote operations on caches and hence, for example, program-directed prestaging and migration of data. HTTP, FTP, and specialized GASS servers are supported.

Finally, the Globus Executable Management (GEM) service, still being designed as of January 1998, is intended to support the identification, location, and creation of executables in heterogeneous environments. GEM provides mechanisms for matching the characteristics of a computer in a computational grid with the runtime requirements of an executable or library. These mechanisms can be used in conjunction with other Globus services to implement a variety of distributed code management strategies, based for example on online executable archives and compile servers.

8 High-Level Tools

While Globus services can be used directly by application programmers, they are more commonly ac-

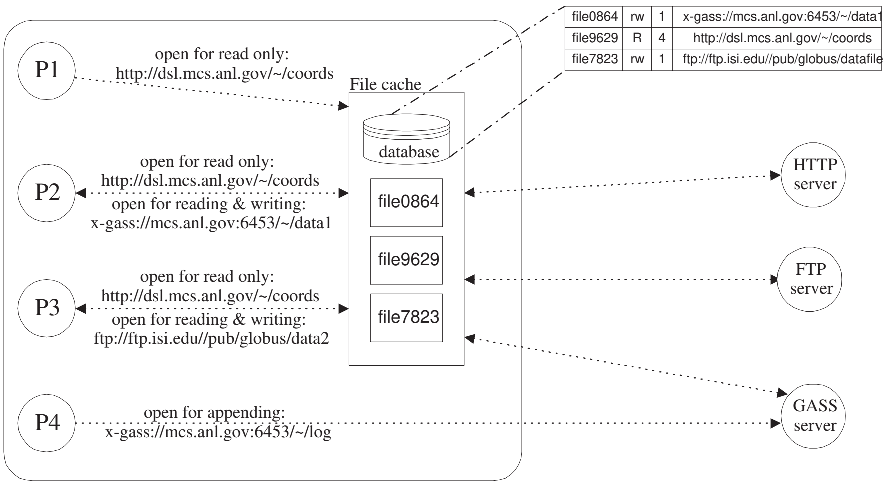

Figure 6: The Global Access to Secondary Storage (GASS) subsystem allows processes on local computers to read and write remote files. Copies of remote files opened for reading and/or writing are maintained in a local file cache. A simple database keeps track of the local file name, access mode, reference count, and remote file URL.

cessed via higher-level tools developed by tool developers. We illustrate this type of use with four examples: a message-passing library, a parallel language, a remote I/O library, and a parameter study system. Each tool uses different Globus services in a different way to support a particular programming model; in each case, availability of the Globus toolkit has allowed existing tools to be adapted for wide-area execution with relatively little effort.

The Message Passing Interface (MPI) defines a standard API for writing message-passing programs and is widely used in parallel computing. For grid applications, message passing has the advantage of providing a higher-level view of communication than TCP/IP sockets, while preserving for the programmer a high degree of control over how and when communication occurs. Globus services have been used to develop a grid-enabled MPI [10] based on the MPICH library [20], with Nexus used for communication, GRAM services for resource allocation, and GSI services for authentication. The result is a system that allows programmers to use simple, standard commands to run MPI programs in a variety of metacomputing environments (freely combining heterogeneous workstation and MPP metacomputing resources), while making efficient use of underlying networks. In future work, the developers of this system plan to use MDS information to construct communication structures—in particular, collective operations—that are optimized for wide-area execution.

Compositional C++ [5], or CC++, is a high-level parallel programming language based on C++. CC++ defines a global name space through the use of *global pointers*, dynamic resource allocation, and support for threading and remote procedure call style communication. The Globus implementation of CC++ uses the same services as the grid-enabled MPI, except that while the MPI implementation relies on Globus co-allocation services for resource allocation, the task-parallel CC++ model interfaces to GRAM directly.

The **Remote I/O** (RIO) library [16] is a tool for achieving high-speed access from parallel programs to files located on remote filesystems. RIO adopts the parallel I/O interface defined by MPI-IO [7, 27] and hence allows any program that uses MPI-IO to operate unchanged in a wide-area environment. The RIO implementation, like that of MPI, is constructed by using Globus services to adapt an existing system—the ROMIO implementation of MPI-IO—to support wide-area execution. Specifically, Nexus services are used for communication, GSI services for authentication, and MDS services for configuration.

Nimrod-G is a wide-area version of Nimrod [1], a tool that automates the creation and management of large parametric experiments. Nimrod allows a user to run a single application under a wide range of input conditions and then to aggregate the results of these different runs for interpretation. In effect, it transforms file-based programs into interactive "meta-applications" that invoke user programs much as we might call subroutines. Nimrod-G uses MDS services to locate suitable resources when a user first requests a computational experiment, and GSI and GRAM services to schedule jobs to resources identified by MDS queries. In effect, Nimrod-G implements resource brokering services specialized for a particular class of application.

9 The GUSTO Testbed

Globus technologies have been deployed in the Globus Ubiquitous Supercomputing Testbed (GUSTO), by several measures the largest computational grid testbed ever constructed as of early 1998. This testbed uses both dedicated OC3 and commodity Internet services to link (as of early 1998) 17 sites, 330 computers, and 3600 processors, providing an aggregate peak performance of 2 Tflop/s. GUSTO sites span the continental United States, Hawaii, Sweden, and Germany; additional sites are being added rapidly. We discuss briefly our experiences deploying, administering, and using this testbed.

GUSTO was created during the three months prior to the November 1997 Supercomputing conference, held in San Jose. During this time, the first version of the Globus toolkit was completed, deployed at 15 sites, and applied in 10 different application projects.

One lesson learned early during this effort was that the approach of defining simple local services (and the considerable effort put into automatic configuration and information discovery tools) was a big win: we were able to deploy Globus software at 15 sites with relative ease, admittedly with considerable help from local staff in some cases. At several sites, computer security officers reviewed and approved our code. The hardest part of the deployment process was typically the development of the GRAM interface to the local scheduler.

Once Globus was deployed, MDS and HBM proved valuable as tools for administering a complex collection of computer systems. The standard interface provided by MDS ensured that GUSTO administrators always had up-to-date information about the structure and state of the system at their fingertips. This information was accessed via both an MDS browser and various specialized Web-based tools developed to publish specific views of the testbed.

Ten different groups developed applications for GUSTO. One of these applications is discussed in the next section; others included remote visualization of scientific simulations, real-time analysis of data from scientific instruments (meteorological satellite and X-ray source), and distributed parameter studies. The tools and services used by these different applications varied tremendously, with some programming in sockets and using just the bare minimum of Globus services, and others exploiting the full range of services.

The security model used for initial GUSTO deployment was based on the plain-text GSS implementation that we have developed. While the plain-text authentication model is quite weak, it had the advantage of avoiding export control issues. However, the need for the stronger, public key implementation was universally expressed. An export license for this technology is pending, and the currently deployed system will be upgraded to this authentication mechanism once such a license is issued.

10 Application Overview

We provide a brief description of one application demonstrated on the initial GUSTO prototype. SF-Express is a distributed interactive simulation (DIS) application that harnesses multiple supercomputers to meet the computational demands of large-scale network-based simulation environments. A large simulation may involve many tens of thousands of entities and requires thousands of processors. Globus services can be used to locate, assemble, and manage those resources. For example, in one experiment in November 1997, SF-Express was run on 852 processors distributed over 6 GUSTO sites. A more detailed discussion of SF-Express and how Globus is being used to support its execution across multiple supercomputers can be found in [4]

An advantage of the Globus bag of services architecture is that an application need not be entirely rewritten before it can operate in a grid environment: services can be introduced into an application incrementally, with functionality increasing at each step. As illustrated in Table 2 and described briefly in the following, this approach is being followed as the original SF-Express is converted into a grid-enabled application.

SF-Express Startup and Configuration Prior to the use of Globus services, simply starting SF-Express on multiple supercomputers was a painful task. The user had to log in to each site in turn and recall the arcane commands needed to allocate resources and start a program. This obstacle to the use of distributed resources was overcome by encoding resource allocation requests in terms of the GRAM API. GRAM and associated GSI services are used to handle authentication, resource allocation, and process creation at each site.

Table 2: A grid-aware version of SF-Express is being constructed incrementally: Globus services are incorporated one by one to improve functionality and reduce application complexity. The Status field indicates code status as of early 1998: techniques are in use (Y), are experimental or in partial use (y), or remain to be applied in the future (blank).

Services	How used	Benefits	Status
GRAM, GSI	Start SF-Express on supercomputers	Avoid need to log in to and schedule each system	Y
+ Co-allocator	Distributed startup and management	Avoid application-level check-in and shutdown	Y
+ MDS	Use MDS information to configure computation	Performance, portability	y
+ Resource broker	Use broker to locate appropriate computers	Code reuse, portability	y
+ Nexus	Encode communication as Nexus RSRs	Uniformity of interface, access to unreliable comms	y
+ HBM	Components check in with application-level monitor	Provides degree of fault tolerance	Y
+ GASS	Use to access terrain database files etc.	Avoid need to prestage data files	
+ GEM	Use to generate and stage executables	Avoid configuration problems	

Currently, the resources used for a simulation are manually specified, using MDS tools to help locate, select, and construct RSL specifications for appropriate supercomputers. As illustrated in Figure 2, these tasks can be avoided if we have access to resource brokers that can automatically construct the required RSL, using information such as the available network bandwidth and CPU power to determine the number of nodes required from the number of entities being simulated, and the number of nodes each router can handle. Once the resource set is identified and the RSL specification generated, Globus co-allocation services are employed to coordinate startup across multiple supercomputers, ensuring that the application has started on the desired resources before allowing the simulation to proceed.

After startup, the simulation must configure itself. In order to execute efficiently on parallel computers that have nonuniform access to network interfaces or secondary storage, SF-Express is organized such that intercomputer communication and I/O activities are performed only within specialized servers. Using information contained in the MDS, SF-Express can configure itself to place these services on appropriate nodes within a parallel computer, that is, the node with the attached disk or network interface card.

Finally, SF-Express must read various files describing the simulation scenario and the terrain on which the simulation is to be performed. In the initial SF-Express prototype, these files had to be staged manually to each site at which SF-Express executed. To simplify this task, we are migrating these file operations to use the GASS service provided in the Globus toolkit.

Communication. The SF-Express demonstrated at SC'97 uses MPI for communication within a simulation group, but handwritten socket code for communication between routers. This approach leads to considerable application code complexity and hinders portability. One approach we are considering is to rewrite the inter-supercomputer communication code to use MPI. The grid-enabled MPI discussed in Section 8 can then be used, eliminating the need for application socket code.

A second approach is to rewrite SF-Express so that communication operations are expressed directly by using Nexus operations. SF-Express communication operations are concerned primarily with the remote enqueing of simulation events and, hence, are expressed more naturally as Nexus RSRs than as MPI calls. A second benefit to using Nexus is that we can then, as discussed in Section 4, select an unreliable communication protocol for the distribution of information to routers. This usage is desirable because SF-Express, unlike many other distributed simulations, does not maintain a global simulation clock. Instead, nodes simply discard incoming events with timestamps earlier than the local simulation clock. Hence, an unreliable protocol that tends to deliver most events sooner than an equivalent reliable protocol may be preferable.

11 Related Work

The primary purpose of this paper is to report on the current status of the Globus project rather than to provide detailed comparisons with related work. Hence, we provide pointers here to just a few representative efforts; the reader is referred to our other papers listed in the references for more detailed discussion.

The Legion project [19], like Globus, is investigating issues relating to software architectures and base technologies for grid environments. In contrast to the Globus bag of services architecture, Legion is organized around an object-oriented model in which every component of the system is represented by an object [23]. In principle, Globus services can be used to implement the Legion object model, so the two projects are in many respects pursuing complementary goals.

Condor [25] is a high-throughput computing environment whose goal is to deliver large amounts of computational capability over long periods of time (weeks or months), rather than peak capacity for limited time durations (hours or days). Condor addresses the needs of a limited, although important class of applications whose components are loosely coupled, often organized into a task-pool style computation. Currently, the GRAM interface to Condor enables Globus users to submit jobs to Condor pools. We are working with the Condor team to integrate other aspects of the systems, such as authentication.

A number of projects are attempting to build distributed computing environments on top of technologies and infrastructure developed for the World Wide Web. These include specialized systems such as SuperWeb [3] and WebOS [30] as well as systems leveraging basic Web technologies, such as Java Remote Method Invocation.

SNIPE [28] is a metacomputing project that builds on the resource management and communication facilities provided by the PVM message-passing library [17]. Like Globus, SNIPE recognizes the importance of information services and uses the Resource Cataloging and Distribution System to provide access to system resources and metadata.

12 Summary and Future Work

We have described the current status of the Globus project, which seeks to develop the basic technologies required to support the construction and use of computational grids. A particular focus of the Globus effort is the development of a small metacomputing toolkit providing essential services that can then be used to implement a variety of higher-level programming models, tools, and applications. As we have explained in this brief review, Globus components have been deployed in large testbeds and used to implement a variety of applications.

We have referred above to the advantages that we perceive in the Globus toolkit approach: in particular, the wide range of global services that can be supported, because of the decoupling of global and local services, and the ability to construct "grid-enabled" applications incrementally, by incorporating services one by one and/or by taking increasing advantage of translucent interfaces. Identification of the weaknesses of the approach will require the construction of larger testbeds and further experimentation with applications. One concern is that the basic techniques might not scale, perhaps because the local services defined by the Globus toolkit are too complex for broad deployment, or because the accuracy of the information provided by MDS declines below a useful level. We are investigating these issues.

We believe that the creation of large-scale testbeds must be a central part of any computational grid project. Hence, we are working with a variety of institutions around the world to create a permanent infrastructure to support experimentation with grid applications and grid software. The initial version of this GUSTO testbed already includes resources at some 17 institutions, and we expect this number to increase.

In current work, we are investigating both grid applications and more sophisticated grid services. We have started to investigate the construction of sophisticated resource brokers and robust co-allocation strategies. We are also studying how MDS can be used to support dynamic configuration and adaptation, so that applications can maintain high levels of performance in the face of dynamic changes in underlying system infrastructure. Finally, we are integrating quality of service mechanisms into the Globus framework. Our initial focus is on guaranteeing communication performance. However, we will also be studying how to integrate processor and memory scheduling into this framework.

For more information on the Globus project and toolkit, see the papers cited here and also the material at www.globus.org.

Acknowledgments

We gratefully acknowledge the numerous contributions of the Globus team, without which the accomplishments detailed here would not have been possible: in particular, Steven Tuecke, Joe Bester, Joe Insley, Nick Karonis, Gregor von Laszewski, Stuart Martin, Warren Smith, and Brian Toonen at Argonne National Laboratory; Karl Czajkowski and Steve Fitzgerald at USC/ISI; and Craig Lee and Paul Stelling at The Aerospace Corporation. SF-Express was developed by Sharon Brunett, Paul Messina, and others at Caltech and JPL. The development of GUSTO was made possible by the considerable assistance offered by staff at each participating site.

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; and by the National Science Foundation.

Author Biographies

Ian Foster received his Ph.D. in computer science from Imperial College, England. He holds a joint appointment as a scientist at Argonne National Laboratory and associate professor in the Department of Computer Science at the University of Chicago. He is the author of three books and over 100 articles and reports on various topics relating to parallel and distributed computing and computational science. In 1995, he led development of the software infrastructure for the I-WAY networking experiment. Dr. Foster co-leads the Globus project with Carl Kesselman.

Carl Kesselman is a project Leader at the Information Sciences Institute and a research associate professor in computer science at the University of Southern California. He received a Ph.D. in computer science at the University of California at Los Angeles. He co-leads the Globus project with Ian Foster. Dr. Kesselman's research interests include high-performance distributed computing, parallel computing, and parallel programming languages.

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proc. 4th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1995.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: The UFO global file system. In *1997 Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*, January 1997.

[3] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Superweb: Towards a global web-based parallel computing infrastructure. In *11th International Parallel Processing Symposium*, April 1997.

[4] S. Brunett, D. Davis, T. Gottschalk, P. Messina, and C. Kesselman. Implementing distributed synthetic forces simulations in metacomputing environments. In *Proceedings of the Heterogeneous Computing Workshop*, 1998. to appear.

[5] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*, pages 281–313. The MIT Press, 1993.

[6] D. E. Comer. *Internetworking with TCP/IP*. Prentice Hall, 3rd edition, 1995.

[7] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NAS, NASA Ames Research Center, Moffett Field, CA, January 1995. Version 0.3.

[8] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

[9] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[10] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. A wide-area implementation of the Message Passing Interface. *Parallel Computing*, 1998. to appear.

[11] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[12] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY metacomputing experiment. *Concurrency: Practice & Experience*, 1998. to appear.

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

[14] I. Foster and C. Kesselman, editors. *Computational Grids: The Future of High-Performance Distributed Computing*. Morgan Kaufmann Publishers, 1998.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[16] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proc. IOPADS'97*, pages 14–25. ACM Press, 1997.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[18] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-wide computing: Results using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, 1995.

[19] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, 1994.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[21] K. Hickman. The SSL protocol. *Internet Draft RFC*, 1995.

[22] T. Howes and M. Smith. The ldap application program interface. RFC 1823, 08/09 1995.

[23] M. Lewis and A. Grimshaw. The core Legion object model. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 562–571. IEEE Computer Society Press, 1996.

[24] J. Linn. Generic security service application program interface. *Internet RFC 1508*, 1993.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[26] P. Lyster, L. Bergman, P. Li, D. Stanfill, B. Crippe, R. Blom, C. Pardo, and D. Okaya. CASA gigabit supercomputing network: CALCRUST three-dimensional real-time multi-dataset rendering. In *Proc. Supercomputing '92*, 1992.

[27] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 1997. http://www.mpi-forum.org.

[28] K. Moore, G. Fagg, A. Geist, and J. Dongarra. Scalable networked information processing environment (SNIPE). In *Proceedings of Supercomputing '97*, 1997.

[29] J. Steiner, B. C. Neuman, and J. Schiller. Kerberos: An authentication system for open network systems. In *Usenix Conference Proceedings*, pages 191–202. 1988.

[30] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating system services for wide area applications. Technical Report UCB CSD-97-938, U.C. Berkeley, 1997.