

## INTRODUCTION

The term “the Grid” was coined in the mid1990s to denote a proposed distributed computing infrastructure for advanced science and engineering. Considerable progress has since been made on the construction of such an infrastructure, but the term “Grid” has also been conflated, at least in popular perception, to embrace everything from advanced networking to artificial intelligence. One might wonder whether the term has any real substance and meaning. Is there really a distinct “Grid problem” and hence a need for new “Grid technologies”? If so, what is the nature of these technologies, and what is their domain of applicability? While numerous groups have interest in Grid concepts and share, to a significant extent, a common vision of Grid architecture, we do not see consensus on the answers to these questions.

The real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resourcebrokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization (VO).

## VIRTUAL ORGANIZATIONS

The following are examples of VOs: the application service providers, storage service providers, cycle providers, and consultants engaged by a car manufacturer to perform scenario evaluation during planning for a new factory; members of an industrial consortium bidding on a new aircraft; a crisis management team and the databases and simulation systems that they use to plan a response to an emergency situation; and members of a large, international, multiyear high-energy physics collaboration. Each of these examples represents an approach to computing and problem solving based on collaboration in computation- and data-rich environments.

As these examples show, VOs vary tremendously in their purpose, scope, size, duration, structure, community, and sociology. Nevertheless, careful study of underlying technology requirements leads us to identify a broad set of common concerns and requirements. In particular, we see a need for highly flexible sharing relationships, ranging from client-server to peer-to-peer; for sophisticated and precise levels of control over how shared resources are used, including fine-grained and multi-stakeholder access control, delegation, and application of local and global policies; for sharing of varied resources, ranging from programs, files, and data to computers, sensors, and networks; and for diverse usage modes, ranging from single user to multi-user and from performance sensitive to cost-sensitive and hence embracing issues of quality of service, scheduling, co-allocation, and accounting.

Current distributed computing technologies do not address the concerns and requirements just listed. For example, current Internet technologies address communication and information exchange among computers but do not provide integrated approaches to the coordinated use of resources at multiple sites for computation. Business-to-business exchanges focus on information sharing (often via centralized servers). So do virtual enterprise technologies, although here sharing may eventually extend to applications and physical devices. Enterprise distributed computing technologies such as CORBA and Enterprise Java enable resource sharing within a single organization. The Open Group's Distributed Computing Environment (DCE) supports secure resource sharing across sites, but most VOs would find it too burdensome and inflexible. Storage service providers (SSPs) and application service providers (ASPs) allow organizations to outsource storage and computing requirements to other parties, but only in constrained ways: for example, SSP resources are typically linked to a customer via a virtual private network (VPN). Emerging "Distributed computing" companies seek to harness idle computers on an international scale but, to date, support only highly centralized access to those resources. In summary, current technology either does not accommodate the range of resource types or does not provide the flexibility and control on sharing relationships needed to establish VOs.

It is here that Grid technologies enter the picture. Over the past five years, research and development efforts within the Grid community have produced protocols, services, and tools that address precisely the challenges that arise when we seek to build scalable VOs. These

technologies include security solutions that support management of credentials and policies when computations span multiple institutions; resource management protocols and services that support secure remote access to computing and data resources and the co-allocation of multiple resources; information query protocols and services that provide configuration and status information about resources, organizations, and services; and data management services that locate and transport datasets between storage systems and applications.

Because of their focus on dynamic, cross-organizational sharing, Grid technologies complement rather than compete with existing distributed computing technologies. For example, enterprise distributed computing systems can use Grid technologies to achieve resource sharing across institutional boundaries; in the ASP/SSP space, Grid technologies can be used to establish dynamic markets for computing and storage resources, hence overcoming the limitations of current static configurations. We discuss the relationship between Grids and these technologies in more detail below.

In the rest of this paper, we expand upon each of these points in turn. Our objectives are to (1) clarify the nature of VOs and Grid computing for those unfamiliar with the area; (2) contribute to the emergence of Grid computing as a discipline by establishing a standard vocabulary and defining an overall architectural framework; and (3) define clearly how Grid technologies relate to other technologies, explaining both why emerging technologies do not yet solve the Grid computing problem and how these technologies can benefit from Grid technologies.

It is our belief that VOs have the potential to change dramatically the way we use computers to solve problems, much as the web has changed how we exchange information. As the examples presented here illustrate, the need to engage in collaborative processes is fundamental to many diverse disciplines and activities: it is not limited to science, engineering and business activities. It is because of this broad applicability of VO concepts that Grid technology is important.

## THE EMERGENCE OF VIRTUAL ORGANIZATIONS

Consider the following four scenarios:

1. A company needing to reach a decision on the placement of a new factory invokes a sophisticated financial forecasting model from an ASP, providing it with access to appropriate proprietary historical data from a corporate database on storage systems operated by an SSP. During the decision-making meeting, what-if scenarios are run collaboratively and interactively, even though the division heads participating in the decision are located in different cities. The ASP itself contracts with a cycle provider for additional “oomph” during particularly demanding scenarios, requiring of course that cycles meet desired security and performance requirements.
2. An industrial consortium formed to develop a feasibility study for a next-generation supersonic aircraft undertakes a highly accurate multidisciplinary simulation of the entire aircraft. This simulation integrates proprietary software components developed by different participants, with each component operating on that participant’s computers and having access to appropriate design databases and other data made available to the consortium by its members.
3. A crisis management team responds to a chemical spill by using local weather and soil models to estimate the spread of the spill, determining the impact based on population location as well as geographic features such as rivers and water supplies, creating a short-term mitigation plan (perhaps based on chemical reaction models), and tasking emergency response personnel by planning and coordinating evacuation, notifying hospitals, and so forth.
4. Thousands of physicists at hundreds of laboratories and universities worldwide come together to design, create, operate, and analyze the products of a major detector at CERN, the European high energy physics laboratory. During the analysis phase, they pool their computing, storage, and networking resources to create a “Data Grid” capable of analyzing petabytes of data.

These four examples differ in many respects: the number and type of participants, the types of activities, the duration and scale of the interaction, and the resources being shared. But they also have much in common, as discussed in the following (see also Figure 1).

In each case, a number of mutually distrustful participants with varying degrees of prior relationship (perhaps none at all) want to share resources in order to perform some task.

Furthermore, sharing is about more than simply document exchange (as in “virtual enterprises”): it can involve direct access to remote software, computers, data, sensors, and other resources. For example, members of a consortium may provide access to specialized software and data and/or pool their computational resources.

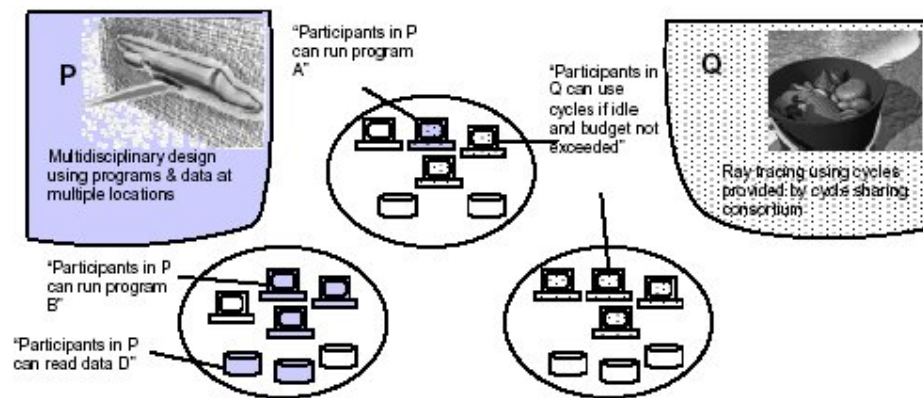


Figure 1: An actual organization can participate in one or more VOs by sharing some or all of its resources. We show three actual organizations (the ovals), and two VOs: P, which links participants in an aerospace design consortium, and Q, which links colleagues who have agreed to share spare computing cycles, for example to run ray tracing computations. The organization on the left participates in P, the one to the right participates in Q, and the third is a member of both P and Q. The policies governing access to resources (summarized in “quotes”) vary according to the actual organizations, resources, and VOs involved.

Resource sharing is conditional: each resource owner makes resources available, subject to constraints on when, where, and what can be done. For example, a participant in VO P of Figure 1 might allow VO P partners to invoke their simulation service only for “simple” problems. Resource consumers may also place constraints on properties of the resources they are prepared to work with. For example, a participant in VO Q might accept only pooled computational resources certified as “secure.” The implementation of such constraints requires mechanisms

for expressing policies, for establishing the identity of a consumer or resource (authentication), and for determining whether an operation is consistent with applicable sharing relationships (authorization).

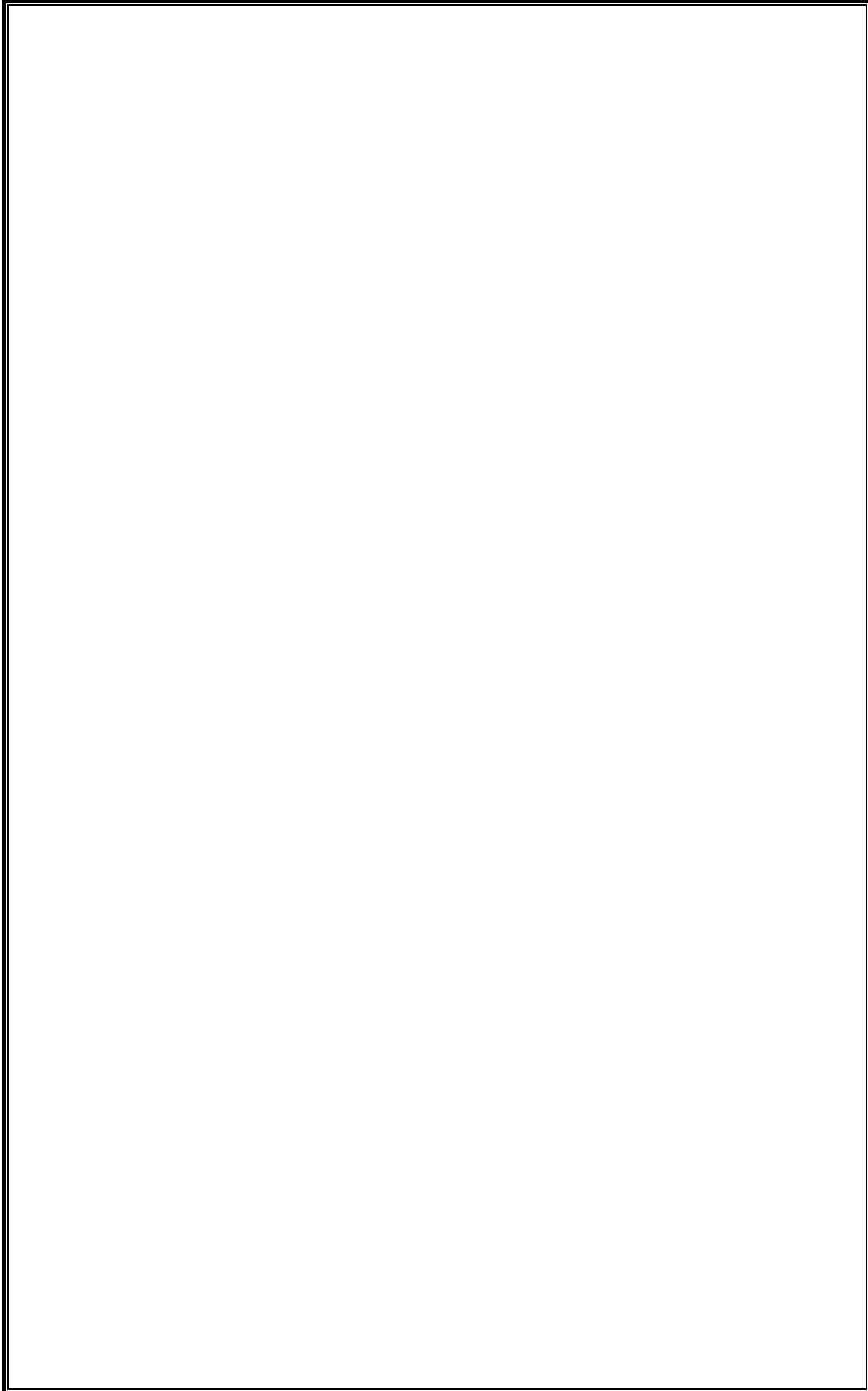
Sharing relationships can vary dynamically over time, in terms of the resources involved, the nature of the access permitted, and the participants to whom access is permitted. And these relationships do not necessarily involve an explicitly named set of individuals, but rather may be defined implicitly by the policies that govern access to resources. For example, an organization might enable access by anyone who can demonstrate that they are a “customer” or a “student.”

The dynamic nature of sharing relationships means that we require mechanisms for discovering and characterizing the nature of the relationships that exist at a particular point in time. For example, a new participant joining VO Q must be able to determine what resources it is able to access, the “quality” of these resources, and the policies that govern access.

Sharing relationships are often not simply client-server, but peer to peer: providers can be consumers, and sharing relationships can exist among any subset of participants. Sharing relationships may be combined to coordinate use across many resources, each owned by different organizations. For example, in VO Q, a computation started on one pooled computational resource may subsequently access data or initiate subcomputations elsewhere. The ability to delegate authority in controlled ways becomes important in such situations, as do mechanisms for coordinating operations across multiple resources (e.g., coscheduling).

The same resource may be used in different ways, depending on the restrictions placed on the sharing and the goal of the sharing. For example, a computer may be used only to run a specific piece of software in one sharing arrangement, while it may provide generic compute cycles in another. Because of the lack of a priori knowledge about how a resource may be used, performance metrics, expectations, and limitations (i.e., quality of service) may be part of the conditions placed on resource sharing or usage.

These characteristics and requirements define what we term a virtual organization, a concept that we believe is becoming fundamental to much of modern computing. VOs enable disparate groups of organizations and/or individuals to share resources in a controlled fashion, so that members may collaborate to achieve a shared goal.



## THE NATURE OF GRID ARCHITECTURE

The establishment, management, and exploitation of dynamic, cross-organizational VO sharing relationships require new technology. We structure our discussion of this technology in terms of a Grid architecture that identifies fundamental system components, specifies the purpose and function of these components, and indicates how these components interact with one another.

In defining a Grid architecture, we start from the perspective that effective VO operation requires that we be able to establish sharing relationships among any potential participants. Interoperability is thus the central issue to be addressed. In a networked environment, interoperability means common protocols. Hence, our Grid architecture is first and foremost a protocol architecture, with protocols defining the basic mechanisms by which VO users and resources negotiate, establish, manage, and exploit sharing relationships. A standards-based open architecture facilitates extensibility, interoperability, portability, and code sharing; standard protocols make it easy to define standard services that provide enhanced capabilities. We can also construct Application Programming Interfaces and Software Development Kits (see Appendix for definitions) to provide the programming abstractions required to create a usable Grid. Together, this technology and architecture constitute what is often termed middleware (“the services needed to support a common set of applications in a distributed network environment”), although we avoid that term here due to its vagueness. We discuss each of these points in the following.

Why is interoperability such a fundamental concern? At issue is our need to ensure that sharing relationships can be initiated among arbitrary parties, accommodating new participants dynamically, across different platforms, languages, and programming environments. In this context, mechanisms serve little purpose if they are not defined and implemented so as to be interoperable across organizational boundaries, operational policies, and resource types. Without interoperability, VO applications and participants are forced to enter into bilateral sharing arrangements, as there is no assurance that the mechanisms used between any two parties will extend to any other parties. Without such assurance, dynamic VO formation is all but impossible, and the types of VOs that can be formed are severely limited. Just as the Web revolutionized information sharing, by providing a universal protocol

and syntax (HTTP and HTML) for information exchange, so we require standard protocols and syntaxes for general resource sharing.

Why are protocols critical to interoperability? A protocol definition specifies how distributed system elements interact with one another in order to achieve a specified behavior, and the structure of the information exchanged during this interaction. This focus on externals (interactions) rather than internals (software, resource characteristics) has important pragmatic benefits. VOs tend to be fluid; hence, the mechanisms used to discover resources, establish identity, determine authorization, and initiate sharing must be flexible and lightweight, so that resource-sharing arrangements can be established and changed quickly. Because VOs complement rather than replace existing institutions, sharing mechanisms cannot require substantial changes to local policies and must allow individual institutions to maintain ultimate control over their own resources. Since protocols govern the interaction between components, and not the implementation of the components, local control is preserved.

Why are services important? A service (see Appendix) is defined solely by the protocol that it speaks and the behaviors that it implements. The definition of standard services—for access to computation, access to data, resource discovery, coscheduling, data replication, and so forth—allows us to enhance the services offered to VO participants and also to abstract away resource specific details that would otherwise hinder the development of VO applications.

Why do we also consider Application Programming Interfaces (APIs) and Software Development Kits (SDKs)? There is, of course, more to VOs than interoperability, protocols, and services. Developers must be able to develop sophisticated applications in complex and dynamic execution environments. Users must be able to operate these applications. Application robustness, correctness, development costs, and maintenance costs are all important concerns. Standard abstractions, APIs, and SDKs can accelerate code development, enable code sharing, and enhance application portability. APIs and SDKs are an adjunct to, not an alternative to, protocols. Without standard protocols, interoperability can be achieved at the API level only by using a single implementation everywhere—inevitable in many interesting VOs—or by having every implementation know the details of every other implementation. (The Jini approach of downloading protocol code to a remote site does not circumvent this requirement.)

In summary, our approach to Grid architecture emphasizes the identification and definition of protocols and services, first; and APIs and SDKs, second.

## GRID ARCHITECTURE DESCRIPTION

Our goal in describing our Grid architecture is not to provide a complete enumeration of all required protocols (and services, APIs, and SDKs) but rather to identify requirements for general classes of component. The result is an extensible, open architectural structure within which can be placed solutions to key VO requirements. Our architecture and the subsequent discussion organize components into layers, as shown in Figure 2. Components within each layer share common characteristics but can build on capabilities and behaviors provided by any lower layer.

In specifying the various layers of the Grid architecture, we follow the principles of the “hourglass model”. The narrow neck of the hourglass defines a small set of core abstractions and protocols (e.g., TCP and HTTP in the Internet), onto which many different high-level behaviors can be mapped (the top of the hourglass), and which themselves can be mapped onto many different underlying technologies (the base of the hourglass). By definition, the number of protocols defined at the neck must be small. In our architecture, the neck of the hourglass consists of Resource and Connectivity protocols, which facilitate the sharing of individual resources. Protocols at these layers are designed so that they can be implemented on top of a diverse range of resource types, defined at the Fabric layer, and can in turn be used to construct a wide range of global services and application-specific behaviors at the Collective layer—so called because they involve the coordinated (“collective”) use of multiple resources.

Our architectural description is high level and places few constraints on design and implementation. To make this abstract discussion more concrete, we also list, for illustrative purposes, the protocols defined within the Globus Toolkit, and used within such Grid projects as the NSF’s National Technology Grid, NASA’s Information Power Grid, DOE’s DISCOM, GriPhyN ([www.griphyn.org](http://www.griphyn.org)), NEESgrid ([www.neesgrid.org](http://www.neesgrid.org)), Particle Physics Data Grid ([www.ppdg.net](http://www.ppdg.net)), and the European Data Grid ([www.eu-datagrid.org](http://www.eu-datagrid.org)). More details will be provided in a subsequent paper.

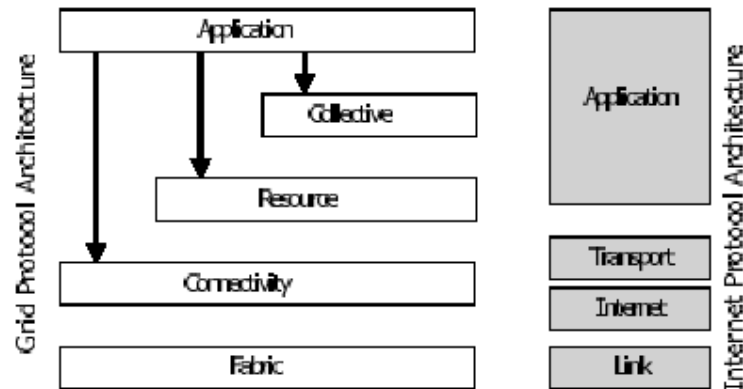


Figure 2: The layered Grid architecture and its relationship to the Internet protocol architecture. Because the Internet protocol architecture extends from network to application, there is a mapping from Grid layers into Internet layers.

## Fabric: Interfaces to Local Control

The Grid Fabric layer provides the resources to which shared access is mediated by Grid protocols: for example, computational resources, storage systems, catalogs, network resources, and sensors. A “resource” may be a logical entity, such as a distributed file system, computer cluster, or distributed computer pool; in such cases, a resource implementation may involve internal protocols (e.g., the NFS storage access protocol or a cluster resource management system’s process management protocol), but these are not the concern of Grid architecture.

Fabric components implement the local, resource-specific operations that occur on specific resources (whether physical or logical) as a result of sharing operations at higher levels. There is thus a tight and subtle interdependence between the functions implemented at the Fabric level, on the one hand, and the sharing operations supported, on the other. Richer Fabric functionality enables more sophisticated sharing operations; at the same time, if we place few demands on Fabric elements, then deployment of Grid infrastructure is simplified. For example, resource level support for advance reservations makes it possible for higher-level services to aggregate (coschedule) resources in

interesting ways that would otherwise be impossible to achieve. However, as in practice few resources support advance reservation “out of the box,” a requirement for advance reservation increases the cost of incorporating new resources into a Grid.

Issue / significance of building large, integrated systems, just-in-time by aggregation (=co-scheduling and co-management) is a significant new capability provided by these Grid services.

Experience suggests that at a minimum, resources should implement enquiry mechanisms that permit discovery of their structure, state, and capabilities (e.g., whether they support advance reservation) on the one hand, and resource management mechanisms that provide some control of delivered quality of service, on the other. The following brief and partial list provides a resource specific characterization of capabilities.

- **Computational resources:** Mechanisms are required for starting programs and for monitoring and controlling the execution of the resulting processes. Management mechanisms that allow control over the resources allocated to processes are useful, as are advance reservation mechanisms. Enquiry functions are needed for determining hardware and software characteristics as well as relevant state information such as current load and queue state in the case of scheduler-managed resources.
- **Storage resources:** Mechanisms are required for putting and getting files. Third-party and high-performance (e.g., striped) transfers are useful. So are mechanisms for reading and writing subsets of a file and/or executing remote data selection or reduction functions. Management mechanisms that allow control over the resources allocated to data transfers (space, disk bandwidth, network bandwidth, CPU) are useful, as are advance reservation mechanisms. Enquiry functions are needed for determining hardware and software characteristics as well as relevant load information such as available space and bandwidth utilization.
- **Network resources:** Management mechanisms that provide control over the resources allocated to network transfers (e.g., prioritization, reservation) can be useful. Enquiry functions should be provided to determine network characteristics and load.
- **Code repositories:** This specialized form of storage resource requires mechanisms for managing versioned source and object code: for example, a control system such as CVS.

- **Catalogs:** This specialized form of storage resource requires mechanisms for implementing catalog query and update operations: for example, a relational database.

**Globus Toolkit:** The Globus Toolkit has been designed to use (primarily) existing fabric components, including vendor-supplied protocols and interfaces. However, if a vendor does not provide the necessary Fabric-level behavior, the Globus Toolkit includes the missing functionality. For example, enquiry software is provided for discovering structure and state information for various common resource types, such as computers (e.g., OS version, hardware configuration, load, scheduler queue status), storage systems (e.g., available space), and networks (e.g., current and predicted future load), and for packaging this information in a form that facilitates the implementation of higher-level protocols, specifically at the Resource layer. Resource management, on the other hand, is generally assumed to be the domain of local resource managers. One exception is the General-purpose Architecture for Reservation and Allocation (GARA), which provides a “slot manager” that, can be used to implement advance reservation for resources that do not support this capability. Others have developed enhancements to the Portable Batch System (PBS) and Condor that support advance reservation capabilities.

### **Connectivity: Communicating Easily and Securely**

The Connectivity layer defines core communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between Fabric layer resources. Authentication protocols build on communication services to provide cryptographically secure mechanisms for verifying the identity of users and resources.

Communication requirements include transport, routing, and naming. While alternatives certainly exist, we assume here that these protocols are drawn from the TCP/IP protocol stack: specifically, the Internet (IP and ICMP), transport (TCP, UDP), and application (DNS, OSPF, RSVP, etc.) layers of the Internet layered protocol architecture. This is not to say that in the future, Grid communications will not demand new protocols that take into account particular types of network dynamics.

With respect to security aspects of the Connectivity layer, we observe that the complexity of the security problem makes it important that any solutions be based on existing standards whenever possible. As with communication, many of the security standards developed within the context of the Internet protocol suite are applicable.

Authentication solutions for VO environments should have the following characteristics:

- Single sign on. Users must be able to “log on” (authenticate) just once and then have access to multiple Grid resources defined in the Fabric layer, without further user intervention.
- Delegation. A user must be able to endow a program with the ability to run on that user’s behalf, so that the program is able to access the resources on which the user is authorized. The program should (optionally) also be able to conditionally delegate a subset of its rights to another program (sometimes referred to as restricted delegation).
- Integration with various local security solutions: Each site or resource provider may employ any of a variety of local security solutions, including Kerberos and Unix security. Grid security solutions must be able to interoperate with these various local solutions. They cannot, realistically, require wholesale replacement of local security solutions but rather must allow mapping into the local environment.
- User-based trust relationships: In order for a user to use resources from multiple providers together, the security system must not require each of the resource providers to cooperate or interact with each other in configuring the security environment. For example, if a user has the right to use sites A and B, the user should be able to use sites A and B together without requiring that A’s and B’s security administrators interact.

Grid security solutions should also provide flexible support for communication protection (e.g., control over the degree of protection, independent data unit protection for unreliable protocols, support for reliable transport protocols other than TCP) and enable stakeholder control over authorization decisions, including the ability to restrict the delegation of rights in various ways.

Globus Toolkit: The Internet protocols listed above are used for communication. The public-key based Grid Security Infrastructure (GSI) protocols are used for authentication, communication protection, and authorization. GSI builds on and extends the Transport Layer Security

(TLS) protocols to address most of the issues listed above: in particular, single sign-on, delegation, integration with various local security solutions (including Kerberos), and user-based trust relationships. X.509-format identity certificates are used. Stakeholder control of authorization is supported via an authorization toolkit that allows resource owners to integrate local policies via a Generic Authorization and Access (GAA) control interface. Rich support for restricted delegations is not provided in the current toolkit releases (v1.1.4) but has been demonstrated in prototypes.

### Resource: Sharing Single Resources

The Resource layer builds on Connectivity layer communication and authentication protocols to define protocols (and APIs and SDKs) for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer implementations of these protocols call Fabric layer functions to access and control local resources. Resource layer protocols are concerned entirely with individual resources and hence ignore issues of global state and atomic actions across distributed collections; such issues are the concern of the Collective layer discussed next.

Two primary classes of Resource layer protocols can be distinguished:

- Information protocols are used to obtain information about the structure and state of a resource, for example, its configuration, current load, and usage policy (e.g., cost).
- Management protocols are used to negotiate access to a shared resource, specifying, for example, resource requirements (including advanced reservation and quality of service) and the operation(s) to be performed, such as process creation, or data access. Since management protocols are responsible for instantiating sharing relationships, they must serve as a “policy application point,” ensuring that the requested protocol operations are consistent with the policy under which the resource is to be shared. Issues that must be considered include accounting and payment. A protocol may also support monitoring the status of an operation and controlling (for example, terminating) the operation.

While many such protocols can be imagined, the Resource (and Connectivity) protocol layers form the neck of our hourglass model, and as such should be limited to a small and focused set. These protocols must be chosen so as to capture the fundamental mechanisms of sharing

across many different resource types (for example, different local resource management systems), while not overly constraining the types or performance of higher-level protocols that may be developed.

The list of desirable Fabric functionality provided in Section 4.1 summarizes the major features required in Resource layer protocols. To this list we add the need for “exactly once” semantics for many operations, with reliable error reporting indicating when operations fail.

Globus Toolkit: A small and mostly standards-based set of protocols is adopted. In particular:

- A Grid Resource Information Protocol (GRIP, currently based on the Lightweight Directory Access Protocol: LDAP) is used to define a standard resource information protocol and associated information model. An associated soft-state resource registration protocol, the Grid Resource Registration Protocol (GRRP), is used to register resources with Grid Index Information Servers, discussed in the next section.
- The HTTP-based Grid Resource Access and Management (GRAM) protocol is used for allocation of computational resources and for monitoring and control of computation on those resources.
- An extended version of the File Transfer Protocol, GridFTP, is a management protocol for data access; extensions include use of Connectivity layer security protocols, partial file access, and management of parallelism for high-speed transfers. FTP is adopted as a base data transfer protocol because of its support for third-party transfers and because its separate control and data channels facilitate the implementation of sophisticated servers.
- LDAP is also used as a catalog access protocol.

The Globus Toolkit defines client-side C and Java APIs and SDKs for each of these protocols.

Server-side SDKs and servers are also provided for each protocol, to facilitate the integration of various resources (computational, storage, network) into the Grid. For example, the Grid Resource Information Service (GRIS) implements server-side LDAP functionality, with callouts allowing for publication of arbitrary resource information. An important server-side element of the overall Toolkit is the “gatekeeper,” which provides what is in essence a GSI-authenticated “inetd” that speaks the GRAM protocol and can be used to dispatch various local operations. The Generic Security Services (GSS) API is used to acquire, forward, and verify authentication credentials and to provide transport

layer integrity and privacy within these SDKs and servers, enabling substitution of alternative security services at the Connectivity layer.

### Collective: Coordinating Multiple Resources

While the Resource layer is focused on interactions with a single resource, the next layer in the architecture contains protocols and services (and APIs and SDKs) that are not associated with any one specific resource but rather are global in nature and capture interactions across collections of resources. For this reason, we refer to the next layer of the architecture as the Collective layer. Because Collective components build on the narrow Resource and Connectivity layer “neck” in the protocol hourglass, they can implement a wide variety of sharing behaviors without placing new requirements on the resources being shared. For example:

- Directory services allow VO participants to discover the existence and/or properties of VO resources. A directory service may allow its users to query for resources by name and/or by attributes such as type, availability, or load. Resource-level GRRP and GRIP protocols are used to construct directories.
- Co-allocation, scheduling, and brokering services allow VO participants to request the allocation of one or more resources for a specific purpose and the scheduling of tasks on the appropriate resources. Examples include AppLeS , Condor-G, Nimrod- G, and the DRM broker.
- Monitoring and diagnostics services support the monitoring of VO resources for failure, adversarial attack (“intrusion detection”), overload, and so forth.
- Data replication services support the management of VO storage (and perhaps also network and computing) resources to maximize data access performance with respect to metrics such as response time, reliability, and cost.
- Grid-enabled programming systems enable familiar programming models to be used in Grid environments, using various Grid services to address resource discovery, security, resource allocation, and other concerns. Examples include Grid-enabled implementations of the Message Passing Interface and manager-worker frameworks.

- Workload management systems and collaboration frameworks—also known as problem solving environments (“PSEs”)—provide for the description, use, and management of multi-step, asynchronous, multi-component workflows
- Software discovery services discover and select the best software implementation and execution platform based on the parameters of the problem being solved. Examples include NetSolve and Ninf.
- Community authorization servers enforce community policies governing resource access, generating capabilities that community members can use to access community resources. These servers provide a global policy enforcement service by building on resource information, and resource management protocols (in the Resource layer) and security protocols in the Connectivity layer.
- Community accounting and payment services gather resource usage information for the purpose of accounting, payment, and/or limiting of resource usage by community members.
- Collaboratory services support the coordinated exchange of information within potentially large user communities, whether synchronously or asynchronously. Examples are CAVERNsoft, Access Grid, and commodity groupware systems.

These examples illustrate the wide variety of Collective layer protocols and services that are encountered in practice. Notice that while Resource layer protocols must be general in nature and are widely deployed, Collective layer protocols span the spectrum from general purpose to highly application or domain specific, with the latter existing perhaps only within specific VOs.

Collective functions can be implemented as persistent services, with associated protocols, or as SDKs (with associated APIs) designed to be linked with applications. In both cases, their implementation can build on Resource layer (or other Collective layer) protocols and APIs. For example, Figure 3 shows a Collective co-allocation API and SDK (the middle tier) that uses a Resource layer management protocol to manipulate underlying resources. Above this, we define a co-reservation service protocol and implement a co-reservation service that speaks this protocol, calling the co-allocation API to implement co-allocation operations and perhaps providing additional functionality, such as authorization, fault tolerance, and logging. An application might then use the co-reservation service protocol to request end-to-end network reservations.

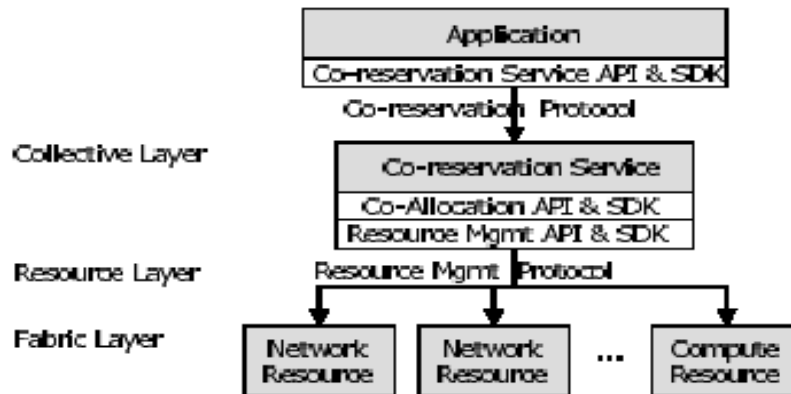


Figure 3: Collective and Resource layer protocols, services, APIs, and SDKs can be combined in a variety of ways to deliver functionality to applications.

Collective components may be tailored to the requirements of a specific user community, VO, or application domain, for example, an SDK that implements an application-specific coherency protocol, or a co-reservation service for a specific set of network resources. Other Collective components can be more general-purpose, for example, a replication service that manages an international collection of storage systems for multiple communities, or a directory service designed to enable the discovery of VOs. In general, the larger the target user community, the more important it is that a Collective component's protocol(s) and API(s) be standards based.

Globus Toolkit: In addition to the example services listed earlier in this section, many of which build on Globus Connectivity and Resource protocols, we mention the Meta Directory Service, which introduces Grid Information Index Servers (GIISs) to support arbitrary views on resource subsets, with the LDAP information protocol used to access resource-specific GRISs to obtain resource state and GRRP used for resource registration. Also replica catalog and replica management services used to support the management of dataset replicas in a Grid

[101seminartopics.com](http://101seminartopics.com)

environment. An online credential repository service (“MyProxy”) provides secure storage for proxy credentials. The DUROC co-allocation library provides an SDK and API for resource co-allocation.

## Applications

The final layer in our Grid architecture comprises the user applications that operate within a VO environment. Figure 4 illustrates an application programmer's view of Grid architecture. Applications are constructed in terms of, and by calling upon, services defined at any layer. At each layer, we have well-defined protocols that provide access to some useful service: resource management, data access, resource discovery, and so forth. At each layer, APIs may also be defined whose implementation (ideally provided by third-party SDKs) exchange protocol messages with the appropriate service(s) to perform desired actions.

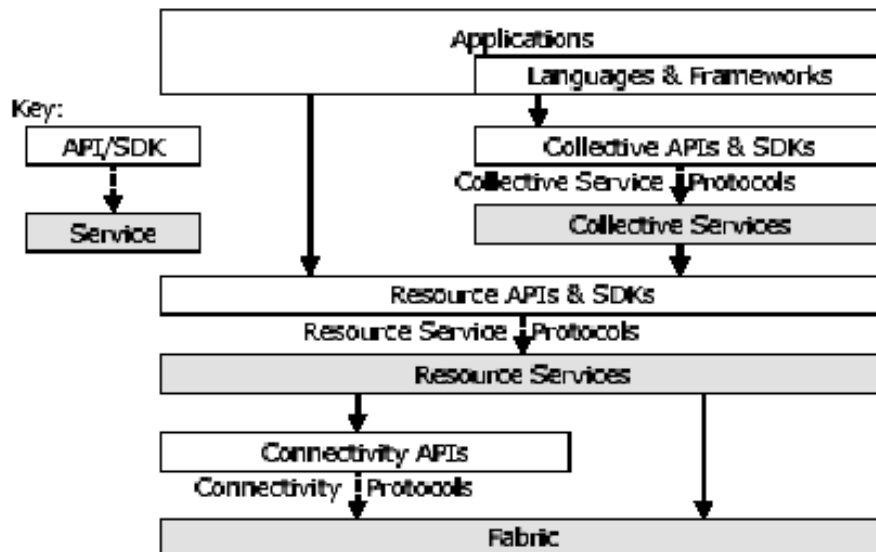


Figure 4: APIs are implemented by software development kits (SDKs), which in turn use Grid protocols to interact with network services that provide capabilities to the end user. Higher level SDKs can provide functionality that is not directly mapped to a specific protocol, but may combine protocol operations with calls to additional APIs as well as implement local functionality. Solid lines represent a direct call; dash lines protocol interactions.

We emphasize that what we label “applications” and show in a single layer in Figure 4 may in practice call upon sophisticated frameworks and libraries (e.g., the Common Component Architecture, SciRun, CORBA, Cactus, workflow systems) and feature much internal structure that would, if captured in our figure, expand it out to many

times its current size. These frameworks may themselves define protocols, services, and/or APIs. (E.g., the SimpleWorkflow Access Protocol.) However, these issues are beyond the scope of this paper, which addresses only the most fundamental protocols and services required in a Grid.

## GRID ARCHITECTURE IN PRACTICE

We use two examples to illustrate how Grid architecture functions in practice. Table 1 shows the services that might be used to implement the multidisciplinary simulation and cycle sharing (ray tracing) applications introduced in Figure 1. The basic Fabric elements are the same in each case: computers, storage systems, and networks. Furthermore, each resource speaks standard Connectivity protocols for communication and security, and Resource protocols for enquiry, allocation, and management. Above this, each application uses a mix of generic and more application-specific Collective services.

In the case of the ray tracing application, we assume that this is based on a high-throughput computing system. In order to manage the execution of large numbers of largely independent tasks in a VO environment, this system must keep track of the set of active and pending tasks, locate appropriate resources for each task, stage executables to those resources, detect and respond to various types of failure, and so forth. An implementation in the context of our Grid architecture uses both domain-specific Collective services (dynamic checkpoint, task pool management, failover) and more generic Collective services (brokering, data replication for executables and common input files), as well as standard Resource and Connectivity protocols. Condor-G represents a first step towards this goal.

Table 1: The Grid services used to construct the two example applications of Figure 1.

	<b>Multidisciplinary Simulation</b>	<b>Ray Tracing</b>
<b>Collective (application-specific)</b>	Solver coupler, distributed data archiver	Checkpointing, job management, failover, staging
<b>Collective (generic)</b>	Resource discovery, resource brokering, system monitoring, community authorization, certificate revocation	
<b>Resource</b>	Access to computation; access to data; access to information about system structure, state, performance.	
<b>Connectivity</b>	Communication (IP), service discovery (DNS), authentication, authorization, delegation	
<b>Fabric</b>	Storage systems, computers, networks, code repositories, catalogs	

In the case of the multidisciplinary simulation application, the problems are quite different at the highest level. Some application framework (e.g., CORBA, CCA) may be used to construct the application from its various components. We also require mechanisms for discovering appropriate computational resources, for reserving time on those resources, for staging executables (perhaps), for providing access to remote storage, and so forth. Again, a number of domain-specific Collective services will be used (e.g., solver coupler, distributed data archiver), but the basic underpinnings are the same as in the ray tracing example.

## THE NEED FOR INTERGRID PROTOCOLS

Our Grid architecture establishes requirements for the protocols and APIs that enable sharing of resources, services, and code. It does not otherwise constrain the technologies that might be used to implement these protocols and APIs. In fact, it is quite feasible to define multiple instantiations of key Grid architecture elements. For example, we can construct both Kerberos and PKI-based protocols at the Connectivity layer—and access these security mechanisms via the same API, thanks to GSS-API (see Appendix). However, Grids constructed with these different protocols are not interoperable and cannot share essential services—at least not without gateways. For this reason, the long-term success of Grid computing requires that we select and achieve widespread deployment of one set of protocols at the Connectivity and Resource layers—and, to a lesser extent, at the Collective layer. Much as the core Internet protocols enable different computer networks to interoperate and exchange information, these Intergrid protocols (as we might call them) enable different organizations to interoperate and exchange or share resources. Resources that speak these protocols can

be said to be “on the Grid.” Standard APIs are also highly useful if Grid code is to be shared. The identification of these Intergrid protocols and APIs is beyond the scope of this paper, although the Globus Toolkit represents an approach that has had some success to date.

## OTHER PERSPECTIVES ON GRIDS

The perspective on Grids and VOs presented in this paper is of course not the only view that can be taken. We summarize here—and critique—some alternative perspectives (given in italics). The Grid is a next-generation Internet. “The Grid” is not an alternative to “the Internet”: it is rather a set of additional protocols and services that build on Internet protocols and services to support the creation and use of computation- and data-enriched environments. Any resource that is “on the Grid” is also, by definition, “on the Net.”

The Grid is a source of free cycles. Grid computing does not imply unrestricted access to resources. Grid computing is about controlled sharing. Resource owners will typically want to enforce policies that constrain access according to group membership, ability to pay, and so forth. Hence, accounting is important, and a Grid architecture must incorporate resource and collective protocols for exchanging usage and cost information, as well as for exploiting this information when deciding whether to enable sharing.

The Grid requires a distributed operating system. In this view (e.g., see ), Grid software should define the operating system services to be installed on every participating system, with these services providing for the Grid what an operating system provides for a single computer: namely, transparency with respect to location, naming, security, and so forth. Put another way, this perspective views the role of Grid software as defining a virtual machine. However, we feel that this perspective is inconsistent with our primary goals of broad deployment and interoperability. We argue that the appropriate model is rather the Internet Protocol suite, which provides largely orthogonal services that address the unique concerns that arise in networked environments. The

tremendous physical and administrative heterogeneities encountered in Grid environments means that the traditional transparencies are unobtainable; on the other hand, it does appear feasible to obtain agreement on standard protocols. The architecture proposed here is deliberately open rather than prescriptive: it defines a compact and minimal set of protocols that a resource must speak to be “on the Grid”; beyond this, it seeks only to provide a framework within which many behaviors can be specified.

The Grid requires new programming models. Programming in Grid environments introduces challenges that are not encountered in sequential (or parallel) computers, such as multiple administrative domains, new failure modes, and large variations in performance. However, we argue that these are incidental, not central, issues and that the basic programming problem is not fundamentally different. As in other contexts, abstraction and encapsulation can reduce complexity and improve reliability. But, as in other contexts, it is desirable to allow a wide variety of higher-level abstractions to be constructed, rather than enforcing a particular approach.

So, for example, a developer who believes that a universal distributed shared memory model can simplify Grid application development should implement this model in terms of Grid protocols, extending or replacing those protocols only if they prove inadequate for this purpose. Similarly, a developer who believes that all Grid resources should be presented to users as objects needs simply to implement an object-oriented “API” in terms of Grid protocols.

The Grid makes high-performance computers superfluous. The hundreds, thousands, or even millions of processors that may be accessible within a VO represent a significant source of computational power, if they can be harnessed in a useful fashion. This does not imply, however, that traditional high-performance computers are obsolete. Many problems require tightly coupled computers, with low latencies and high communication bandwidths; Grid computing may well increase, rather than reduce, demand for such systems by making access easier.

## CONCLUSION

I have provided in this paper a concise statement of the “Grid problem,” which we define as controlled and coordinated resource sharing and resource use in dynamic, scalable virtual organizations. I have also presented both requirements and a framework for Grid architecture, identifying the principal functions required to enable sharing within VOs and defining key relationships among these different functions. Finally, we have discussed in some detail how Grid technologies relate to other important technologies. I hope that the vocabulary and structure introduced in this document will prove useful to the emerging Grid community, by improving understanding of our problem and providing a common language for describing solutions. I also hope that our analysis will help establish connections among Grid developers and proponents of related technologies.

The discussion in this paper also raises a number of important questions. What are appropriate choices for the Intergrid protocols that will enable interoperability among Grid systems? What services should be present in a persistent fashion (rather than being duplicated by each application) to create usable Grids? And what are the key APIs and SDKs that must be delivered to users in order to accelerate development and deployment of Grid applications? I have presented some opinions on these questions, but the answers clearly require further research.

## REFERENCES

- <http://www.google.com>
- <http://www.altavista.com>
- <http://www.csse.monash.edu.au/~rajkumar/papers>
- <http://www.cs.virginia.edu/~humphrey/GridComputingClass/papers>
- <http://www.howstuffworks.com>
- <http://zuni.cs.vt.edu/grid-computing>
- <http://zuni.cs.vt.edu/grid-computing>
- <http://www.gridforum.org>
- <http://www.globus.org/research/papers>
- <http://www.dhpc.adelaide.edu.au/education/dhpc/>
- <http://www.listproc.bucknell.edu/archives/tfcc-1/200003/msg00002.html>
- <http://www.buyya.com/papers/cpm.pdf>

## APPENDIX: DEFINITIONS

I define here four terms that are fundamental to the discussion in this paper but are frequently misunderstood and misused, namely, protocol, service, SDK, and API. Protocol. A protocol is a set of rules that end points in a telecommunication system use when exchanging information. For example:

- The Internet Protocol (IP) defines an unreliable packet transfer protocol.
- The Transmission Control Protocol (TCP) builds on IP to define a reliable data delivery protocol.
- The Transport Layer Security (TLS) Protocol defines a protocol to provide privacy and data integrity between two communicating applications. It is layered on top of a reliable transport protocol such as TCP.
- The Lightweight Directory Access Protocol (LDAP) builds on TCP to define a queryresponse protocol for querying the state of a remote database.

An important property of protocols is that they admit to multiple implementations: two end points need only implement the same protocol to be able to communicate. Standard protocols are thus fundamental to achieving interoperability in a distributed computing environment.

A protocol definition also says little about the behavior of an entity that speaks the protocol. For example, the FTP protocol definition indicates the format of the messages used to negotiate a file transfer but does not make clear how the receiving entity should manage its files.

As the above examples indicate, protocols may be defined in terms of other protocols.

Service: A service is a network-enabled entity that provides a specific capability, for example, the ability to move files, create processes, or verify access rights. A service is defined in terms of the protocol one uses to interact with it and the behavior expected in response to various protocol message exchanges (i.e., “service = protocol + behavior.”). A service definition may permit a variety of implementations. For example:

- An FTP server speaks the File Transfer Protocol and supports remote\_read and write access to a collection of files. One FTP server implementation may simply write to and read from the server's local disk, while another may write to and read from a mass storage system, automatically compressing and uncompressing files in the process. From a Fabric-level perspective, the behaviors of these two servers in response to a store request (or retrieve request) are very different. From the perspective of a client of this service, however, the behaviors are indistinguishable; storing a file and then retrieving the same file will yield the same results regardless of which server implementation is used.
- An LDAP server speaks the LDAP protocol and supports response to queries. One LDAP server implementation may respond to queries using a database of information, while another may respond to queries by dynamically making SNMP calls to generate the necessary information on the fly.

A service may or may not be persistent (i.e., always available), be able to detect and/or recover from certain errors; run with privileges, and/or have a distributed implementation for enhanced calability. If variants are possible, then discovery mechanisms that allow a client to determine the properties of a particular instantiation of a service are important.

Note also that one can define different services that speak the same protocol. For example, in the Globus Toolkit, both the replica catalog and information service use LDAP.

API: An Application Program Interface (API) defines a standard interface (e.g., set of subroutine calls, or objects and method invocations in the case of an object-oriented API) for invoking a specified set of functionality. For example:

- The Generic Security Service (GSS) API defines standard functions for verifying identify of communicating parties, encrypting messages, and so forth.
- The Message Passing Interface API defines standard interfaces, in several languages, to functions used to transfer data among processes in a parallel computing system.

An API may define multiple language bindings or use an Interface Definition Language. The language may be a conventional programming language such as C or Java, or it may be a shell interface. In the latter case, the API refers to particular a definition of command

line arguments to the program, the input and output of the program, and the exit status of the program. An API normally will specify a standard behavior but can admit to multiple implementations.

It is important to understand the relationship between APIs and protocols. A protocol definition says nothing about the APIs that might be called from within a program to generate protocol messages. A single protocol may have many APIs; a single API may have multiple implementations that target different protocols. In brief, standard APIs enable portability; standard protocols enable interoperability. For example, both public key and Kerberos bindings have been defined for the GSS-API. Hence, a program that uses GSS-API calls for authentication operations can operate in either a public key or a Kerberos environment without change. On the other hand, if we want a program to operate in a public key and a Kerberos environment at the same time, then we need a standard protocol that supports interoperability of these two environments. See Figure 5.



Figure 5: On the left, an API is used to develop applications that can target either Kerberos or PKI security mechanisms. On the right, protocols (the Grid security protocols provided by the Globus Toolkit) are used to enable interoperability between Kerberos and PKI domains.

SDK. The term software development kit (SDK) denotes a set of code designed to be linked with, and invoked from within, an application program to provide specified functionality. An SDK typically implements an API. If an API admits to multiple implementations, then there will be multiple SDKs for that API. Some SDKs provide access to services via a particular protocol.

For example:

- The OpenLDAP release includes an LDAP client SDK, which contains a library of functions that can be used from a C or C++ application to perform queries to an LDAP service.
- JNDI is a Java SDK, which contains functions that can be used to perform queries to an LDAP service.
- Different SDKs implement GSS-API using the TLS and Kerberos protocols, respectively.

There may be multiple SDKs, for example from multiple vendors, which implement a particular protocol. Further, for client-server oriented protocols, there may be separate client SDKs for use by applications that want to access a service, and server SDKs for use by service implementers that want to implement particular, customized service behaviors.

An SDK need not speak any protocol. For example, an SDK that provides numerical functions may act entirely locally and not need to speak to any services to perform its operations.

## ABSTRACT

“Grid” computing has emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation. First, we review the “Grid problem,” which we define as flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources—what we refer to as virtual organizations. In such settings, we encounter unique authentication, authorization, resource access, resource discovery, and other challenges. It is this class of problem that is addressed by Grid technologies. Next, we will see an extensible and open Grid architecture, in which protocols, services, application programming interfaces, and software development kits are categorized according to their roles in enabling resource sharing. We describe requirements that any such mechanisms must satisfy and we discuss the importance of defining a compact set of intergrid protocols to enable interoperability among different Grid systems. Finally, we discuss how Grid technologies relate to other contemporary technologies, including enterprise integration, application service provider, storage service provider, and peer-to-peer computing. We maintain that Grid concepts and technologies complement and have much to contribute to these other approaches.

## CONTENTS

1. Introduction
  2. Virtual organizations
  3. The Emergence of Virtual Organizations
  4. The Nature of Grid Architecture
  5. Grid Architecture Description
    - Fabric: Interfaces to Local Control
    - Connectivity: Communicating Easily and Securely
    - Resource: Sharing Single Resources
    - Collective: Coordinating Multiple Resources
    - Applications
  6. Grid Architecture in Practice
  7. The Need for Intergrid Protocols
  8. Other Perspectives on Grids
  9. Conclusion
  10. References
- Appendix: Definitions

## ACKNOWLEDGEMENT

I thank God Almighty for the successful completion of my seminar.

I express my sincere gratitude to Dr. M N Agnisharman Namboothiri, Head of the Department, Information Technology. I am deeply indebted to Staff-in-charge, Miss. Sangeetha Jose and Mr. Biju, for their valuable advice and guidance. I am also grateful to all other members of the faculty of Information Technology department for their cooperation.

Finally, I wish to thank all my dear friends, for their whole-hearted cooperation, support and encouragement.