The Physiology of the Grid

An Open Grid Services Architecture for Distributed Systems Integration

Ian Foster[1,2] Carl Kesselman[3] Jeffrey M. Nick[4] Steven Tuecke[1]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

[2] Department of Computer Science, University of Chicago, Chicago, IL 60637

[3] Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292

[4] IBM Corporation, Poughkeepsie, NY 12601

foster@mcs.anl.gov carl@isi.edu jnick@us.ibm.com tuecke@mcs.anl.gov

Abstract

In both e-business and e-science, we often need to integrate services across distributed, heterogeneous, dynamic "virtual organizations" formed from the disparate resources within a single enterprise and/or from external resource sharing and service provider relationships. This integration can be technically challenging because of the need to achieve various qualities of service when running on top of different native platforms. We present an *Open Grid Services Architecture* that addresses these challenges. Building on concepts and technologies from the Grid and Web services communities, this architecture defines a uniform exposed service semantics (the *Grid service*); defines standard mechanisms for creating, naming, and discovering transient Grid service instances; provides location transparency and multiple protocol bindings for service instances; and supports integration with underlying native platform facilities. The Open Grid Services Architecture also defines, in terms of Web Services Description Language (WSDL) interfaces and associated conventions, mechanisms required for creating and composing sophisticated distributed systems, including lifetime management, change management, and notification. Service bindings can support reliable invocation, authentication, authorization, and delegation, if required. Our presentation complements an earlier foundational article, "The Anatomy of the Grid," by describing how Grid mechanisms can implement a service-oriented architecture, explaining how Grid functionality can be incorporated into a Web services framework, and illustrating how our architecture can be applied within commercial computing as a basis for distributed system integration—within and across organizational domains.

This is a DRAFT document and continues to be revised. The latest version can be found at http://www.globus.org/research/papers/ogsa.pdf. Please send comments to foster@mcs.anl.gov, carl@isi.edu, jnick@us.ibm.com, tuecke@mcs.anl.gov

Table of Contents

1 Introduction

Until recently, application developers could often assume a target environment that was (to a useful extent) homogeneous, reliable, secure, and centrally managed. Increasingly, however, computing is concerned with collaboration, data sharing, and other new modes of interaction that involve distributed resources. The result is an increased focus on the interconnection of systems both within and across enterprises, whether in the form of intelligent networks, switching devices, caching services, appliance servers, storage systems, or storage area network management systems. In addition, companies are realizing that they can achieve significant cost savings by outsourcing nonessential elements of their IT environment to various forms of service providers.

These evolutionary pressures generate new requirements for distributed application development and deployment. Today, applications and middleware are typically developed for a specific platform (e.g., Windows NT, a flavor of Unix, a mainframe, J2EE, Microsoft .NET) that provides a hosting environment for running applications. The capabilities provided by such platforms may range from integrated resource management functions to database integration, clustering services, security, workload management, and problem determination—with different implementations, semantic behaviors, and APIs for these functions on different platforms. But in spite of this diversity, the continuing decentralization and distribution of software, hardware, and human resources make it essential that we achieve desired qualities of service (QoS)—whether measured in terms of common security semantics, distributed workflow and resource management performance, coordinated fail-over, problem determination services, or other metrics—on resources assembled dynamically from enterprise systems, service provider systems, and customer systems. We require new abstractions and concepts that allow applications to access and share resources and services across distributed, wide area networks.

Such problems have been for some time a central concern of the developers of distributed systems for large-scale scientific research. Work within this community has led to the development of *Grid technologies* [30, 34], which address precisely these problems and which are seeing widespread and successful adoption for scientific and technical computing.

In an earlier article, we defined Grid technologies and infrastructures as supporting the sharing and coordinated use of diverse resources in dynamic, distributed "virtual organizations" (VOs) [34]. We defined essential properties of Grids and introduced key requirements for protocols and services, distinguishing among *connectivity* protocols concerned with communication and authentication, *resource* protocols concerned with negotiating access to individual resources, and *collective* protocols and services concerned with the coordinated use of multiple resources. We also described the Globus Toolkit™ [1] [29], an open source reference implementation of key Grid protocols that supports a wide variety of major e-science projects.

Here we extend this argument in three respects to define more precisely how a Grid functions and how Grid technologies can be implemented and applied. First, while [34] was structured in terms of the protocols required for interoperability among VO components, we focus here on the nature of the *services* that respond to protocol messages. We view a Grid as an extensible set of *Grid services* that may be aggregated in various ways to meet the needs of VOs, which themselves can be defined in part by the services that they operate and share. We then define the behaviors that such Grid services should possess in order to support distributed systems integration. By stressing functionality (i.e., "physiology"), this view of Grids complements the previous protocol-oriented ("anatomical") description.

[1] Globus Project and Globus Toolkit are trademarks of the University of Chicago.

Second, we explain how Grid technologies can be aligned with Web services technologies [40, 47] to capitalize on desirable Web services properties, such as service description and discovery; automatic generation of client and server code from service descriptions; binding of service descriptions to interoperable network protocols; compatibility with emerging higher-level open standards, services and tools; and broad commercial support. We call this alignment—and augmentation—of Grid and Web services technologies an *Open Grid Services Architecture* (OGSA), with the term *architecture* denoting here a well-defined set of basic interfaces from which can be constructed interesting systems, and *open* being used to communicate extensibility, vendor neutrality, and commitment to a community standardization process. This architecture uses the Web Services Description Language (WSDL) to achieve self-describing, discoverable services and interoperable protocols, with extensions to support multiple coordinated interfaces and change management. OGSA leverages experience gained with the Globus Toolkit to define conventions and WSDL interfaces for a *Grid service*, a (potentially transient) stateful service instance supporting reliable and secure invocation (when required), lifetime management, notification, policy management, credential management, and virtualization. OGSA also defines interfaces for the discovery of Grid service instances and for the creation of transient Grid service instances. The result is a standards-based distributed service system (we avoid the term distributed object system due to its overloaded meaning) that supports the creation of the sophisticated distributed services required in modern enterprise and interorganizational computing environments.

Third, we focus our discussion on commercial applications rather than the scientific and technical applications emphasized in [30, 34]. We believe that the same principles and mechanisms apply in both environments. However, in commercial settings we need, in particular, seamless integration with existing resources and applications, and with tools for workload, resource, security, network QoS, and availability management. OGSA's support for the discovery of service properties facilitates the mapping or *adaptation* of higher-level Grid service functions to such native platform facilities. OGSA's service orientation also allows us to *virtualize* resources at multiple levels, so that the same abstractions and mechanisms can be used both within distributed Grids supporting collaboration across organizational domains and within hosting environments spanning multiple tiers within a single IT domain. A common infrastructure means that differences (e.g., relating to visibility and accessibility) derive from policy controls associated with resource ownership, privacy, and security, rather than interaction mechanisms. Hence, as today's enterprise systems are transformed from separate computing resource islands to integrated, multitiered distributed systems, service components can be integrated dynamically and flexibly, both within and across various organizational boundaries.

The rest of this article is as follows. In Section 2, we examine the issues that motivate the use of Grid technologies in commercial settings. In Section 3, we review the Globus Toolkit and Web services, and in Section 4, we motivate and introduce our Open Grid Services Architecture. In Sections 5−8, we present an example and discuss protocol implementations and higher-level services. We discuss related work in Section 9 and summarize our discussion in Section 10.

We emphasize that the Open Grid Services Architecture and associated Grid service specifications continue to evolve as a result of both standards work within the Global Grid Forum and implementation work within the Globus Project and elsewhere. Thus the technical content in this article, and in an earlier abbreviated presentation [32], represent only a snapshot of a work in progress.

2 The Need for Grid Technologies

Grid technologies support the sharing and coordinated use of diverse resources in dynamic VOs—that is, the creation, from geographically and organizationally distributed components, of virtual computing systems that are sufficiently integrated to deliver desired QoS [34].

Grid concepts and technologies were first developed to enable resource sharing within far-flung scientific collaborations [18, 19, 28, 30, 46, 64]. Applications include collaborative visualization of large scientific datasets (pooling of expertise), distributed computing for computationally demanding data analyses (pooling of compute power and storage), and coupling of scientific instruments with remote computers and archives (increasing functionality as well as availability) [45]. We expect similar applications to become important in commercial settings, initially for scientific and technical computing applications (where we can already point to success stories) and then for commercial distributed computing applications, including enterprise application integration and business to business (B2B) partner collaboration over the Internet. Just as the World Wide Web began as a technology for scientific collaboration and was adopted for e-business, we expect a similar trajectory for Grid technologies.

Nevertheless, we argue that Grid concepts are critically important for commercial computing not primarily as a means of enhancing capability, but rather as a solution to new challenges relating to the construction of reliable, scalable, and secure distributed systems. These challenges derive from the current rush, driven by technology trends and commercial pressures, to decompose and distribute through the network previously monolithic host-centric services, as we now discuss.

2.1 The Evolution of Enterprise Computing

In the past, computing typically was performed within highly integrated host-centric enterprise computing centers. While sophisticated distributed systems (e.g., command and control systems, reservation systems, the Internet Domain Name System [52]) existed, these have remained specialized, niche entities [9, 54].

The rise of the Internet and the emergence of e-business have, however, led to a growing awareness that an enterprise's IT infrastructure also encompasses external networks, resources, and services. Initially, this new source of complexity was treated as a network-centric phenomenon and attempts were made to construct "intelligent networks" that intersect with traditional enterprise IT data centers only at "edge servers": for example, an enterprise's Web point of presence, or the virtual private network server that connects an enterprise network to service provider resources. The assumption was that the impact of e-business and the Internet on an enterprise's core IT infrastructure could thus be managed and circumscribed.

This attempt has, in general, failed because IT services decomposition is also occurring *inside* enterprise IT facilities. New applications are being developed to programming models (such as the Enterprise Java Beans component model [65]) that insulate the application from the underlying computing platform and support portable deployment across multiple platforms. This portability in turn allows platforms to be selected on the basis of price/performance and QoS requirements, rather than operating system supported. Thus, for example, Web serving and caching applications target commodity servers rather than traditional mainframe computing platforms. The resulting proliferation of Unix and NT servers necessitates distributed connections to legacy mainframe application and data assets. Increased load on those assets has caused companies to off-load nonessential functions (such as query processing) from back-end transaction processing systems to mid-tier servers. Meanwhile, Web access to enterprise resources requires ever-faster request servicing, further driving the need to distribute and cache content closer to the edge of the network. The overall result is a decomposition of highly integrated internal IT infrastructure into a collection of heterogeneous and fragmented systems.

Enterprises must then reintegrate (with QoS) these distributed servers and data resources, addressing issues of navigation, distributed security, and content distribution inside the enterprise, much as on external networks.

In parallel with these developments, enterprises are engaging ever more aggressively in e-business and are realizing that a highly robust IT infrastructure is required to handle the associated unpredictability and rapid growth. Enterprises are also now expanding the scope and scale of their enterprise resource planning projects as they try to provide better integration with customer relationship management, integrated supply chain, and existing core systems. These developments are adding to the significant pressures on the enterprise IT infrastructure.

The aggregate effect is that *qualities of service traditionally associated with mainframe host-centric computing [56] are now essential to the effective conduct of e-business across distributed compute resources, inside as well as outside the enterprise*. For example, enterprises must provide consistent response times to customers, despite workloads with significant deviations between average and peak utilization. Thus, they require flexible resource allocation in accordance with workload demands and priorities. Enterprises must also provide a secure and reliable environment for distributed transactions flowing across a collection of dissimilar servers, must deliver continuous availability as seen by end-users, and must support disaster recovery for business workflow across a distributed network of application and data servers. Yet the current paradigm for delivering QoS to applications via the vertical integration of platform-specific components and services just does not work in today's distributed environment: the decomposition of monolithic IT infrastructures is not consistent with the delivery of QoS through vertical integration of services on a given platform. Nor are distributed resource management capabilities effective, being limited by their proprietary nature, inaccessibility to platform resources, and inconsistencies between similar resources across a distributed environment.

The result of these trends is that IT systems integrators take on the burden of re-integrating distributed compute resources with respect to overall QoS. However, without appropriate infrastructure tools, the management of distributed computing workflow becomes increasingly labor-intensive, complex, and fragile as platform-specific operations staff watch for "fires" in overall availability and performance and verbally collaborate on corrective actions across different platforms. This situation is not scalable, cost-effective, or tenable in the face of changes to the computing environment and application portfolio.

2.2 Service Providers and Business-to-Business Computing

Another key trend is the emergence of service providers (SPs) of various types, such as web-hosting SPs, content distribution SPs, applications SPs, and storage SPs. By exploiting economies of scale, SPs aim to take standard e-business processes, such as creation of a web-portal presence, and provide them to multiple customers with superior price/performance. Even traditional enterprises with their own IT infrastructures are offloading such processes because they are viewed as commodity functions.

Such emerging "eUtilities" (a term used to refer to service providers offering continuous, on-demand access) are beginning to offer a model for carrier-grade IT resource delivery through metered usage and subscription services. Unlike the computing services companies of the past, which tended to provide offline batch-oriented processes, resources provided by eUtilities are often tightly integrated with of enterprise computing infrastructures and used for business processes that span both in-house and outsourced resources. Thus, a price of exploiting the economies of scale that are enabled by eUtility structures is a further decomposition and distribution of enterprise computing functions. EUtilities providers face their own technical challenges. To achieve economies of scale, eUtility providers require server infrastructures that can be easily customized on demand to meet specific customer needs. Thus, there is a demand for

IT infrastructure that (1) supports dynamic resource allocation in accordance with service-level agreement policies, efficient sharing and reuse of IT infrastructure at high utilization levels, and distributed security from edge of network to application and data servers and (2) delivers consistent response times and high levels of availability—which in turn drives a need for end-to-end performance monitoring and real-time reconfiguration.

Still another key IT industry trend is cross-enterprise business-to-business (B2B) collaboration such as multi-organization supply chain management, virtual web malls, and electronic market auctions. B2B relationships are, in effect, virtual organizations, as defined above—albeit with particularly stringent requirements for security, auditability, availability, service level agreements, and complex transaction processing flows. Thus, B2B computing represents another source of demand for distributed systems integration, characterized by often large differences among the information technologies deployed within different organizations.

3 Background

We review two technologies on which we build to define the Open Grid Services Architecture: the Globus Toolkit, which has been widely adopted as a Grid technology solution for scientific and technical computing, and Web services, which have emerged as a popular standards-based framework for accessing network applications.

3.1 The Globus Toolkit

The Globus Toolkit [29, 34] is a community-based, open-architecture, open-source set of services and software libraries that support Grids and Grid applications. The toolkit addresses issues of security, information discovery, resource management, data management, communication, fault detection, and portability. Globus Toolkit mechanisms are in use at hundreds of sites and by dozens of major Grid projects worldwide.

The toolkit components that are most relevant to OGSA are the Grid Resource Allocation and Management (GRAM) protocol and its "gatekeeper" service, which provides for secure, reliable, service creation and management [25]; the Meta Directory Service (MDS-2) [24], which provides for information discovery through soft state registration [59, 69], data modeling, and a local registry ("GRAM reporter" [25]); and the Grid Security Infrastructure (GSI), which supports single sign on, delegation, and credential mapping. As illustrated in Figure 1, these components provide the essential elements of a service-oriented architecture, but with less generality than is achieved in OGSA.

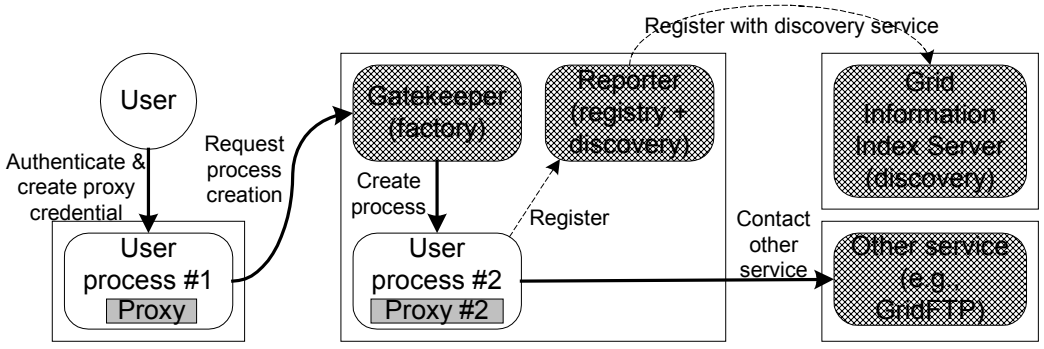

Figure 1: Selected Globus Toolkit mechanisms, showing initial creation of a proxy credential and subsequent authenticated requests to a remote gatekeeper service, resulting in the creation of user process #2, with associated (potentially restricted) proxy credential, followed by a request to another remote service. Also shown is soft-state service registration via MDS-2.

The GRAM protocol provides for the reliable, secure remote creation and management of arbitrary computations: what we term in this article *transient service instances*. GSI mechanisms are used for authentication, authorization, and credential delegation [38] to remote computations. A two-phase commit protocol is used for reliable invocation, based on techniques used in the Condor system [50]. Service creation is handled by a small, trusted "gatekeeper" process (termed a *factory* in this article), while a GRAM reporter monitors and publishes information about the identity and state of local computations (*registry*).

MDS-2 [24] provides a uniform framework for discovering and accessing system configuration and status information such as compute server configuration, network status, or the locations of replicated datasets (what we term a *discovery* interface in this article). MDS-2 uses a soft-state protocol, the Grid Notification Protocol [44], for lifetime management of published information.

The public-key-based Grid Security Infrastructure (GSI) protocol [33] provides single sign-on authentication, communication protection, and some initial support for restricted delegation. In brief, *single sign-on* allows a user to authenticate once and thus create a proxy credential that a program can use to authenticate with any remote service on the user's behalf. *Delegation* allows for the creation and communication to a remote service of delegated proxy credentials that the remote service can use to act on the user's behalf, perhaps with various restrictions; this capability is important for nested operations. (Similar mechanisms can be implemented within the context of other security technologies, such as Kerberos [63], although with potentially different characteristics.)

GSI uses X.509 certificates, a widely employed standard for PKI certificates, as the basis for user authentication. GSI defines an X.509 proxy certificate [67] to leverage X.509 for support of single sign-on and delegation. (This proxy certificate is similar in concept to a Kerberos forwardable ticket but is based purely on public key cryptographic techniques.) GSI typically uses the Transport Layer Security (TLS) protocol (the follow-on to SSL) for authentication, although other public key-based authentication protocols could be used with X.509 proxy certificates. A remote delegation protocol of X.509 proxy certificates is layered on top of TLS. An Internet Engineering Task Force draft defines the X.509 Proxy Certificate extensions [67]. Global Grid Forum drafts define the delegation protocol for remote creation of an X.509 Proxy Certificate [67] and GSS-API extensions that allow this API to be used effectively for Grid programming.

Rich support for restricted delegation has been demonstrated in prototypes and is a critical part of the proposed X.509 Proxy Certificate Profile [67]. Restricted delegation allows one entity to delegate just a subset of its total privileges to another entity. Such restriction is important to reduce the adverse effects of either intentional or accidental misuse of the delegated credential.

3.2 Web Services

The term *Web services* describes an important emerging distributed computing paradigm that differs from other approaches such as DCE, CORBA, and Java RMI in its focus on simple, Internet-based standards (e.g., eXtensible Markup Language: XML [14, 27]) to address heterogeneous distributed computing. Web services define a technique for describing software components to be accessed, methods for accessing these components, and discovery methods that enable the identification of relevant service providers. Web services are programming language-, programming model-, and system software-neutral.

Web services standards are being defined within the W3C and other standards bodies and form the basis for major new industry initiatives such as Microsoft (.NET), IBM (Dynamic e-Business), and Sun (Sun ONE). We are particularly concerned with three of these standards: SOAP, WSDL, and WS-Inspection.

- The *Simple Object Access Protocol* (SOAP) [4] provides a means of messaging between a service provider and a service requestor. SOAP is a simple enveloping mechanism for XML payloads that defines a remote procedure call (RPC) convention and a messaging convention. SOAP is independent of the underlying transport protocol; SOAP payloads can be carried on HTTP, FTP, Java Messaging Service (JMS), and the like. We emphasize that Web services can describe multiple access mechanisms to the underlying software component. SOAP is just one means of formatting a Web service invocation.
- The *Web Services Description Language* (WSDL) [22] is an XML document for describing Web services as a set of *endpoints* operating on messages containing either document-oriented (messaging) or RPC payloads. Service interfaces are defined abstractly in terms of message structures and sequences of simple message exchanges (or operations, in WSDL terminology) and then bound to a concrete network protocol and data-encoding format to define an endpoint. Related concrete endpoints are bundled to define abstract endpoints (services). WSDL is extensible to allow description of endpoints and the concrete representation of their messages for a variety of different message formats and network protocols. Several standardized binding conventions are defined describing how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST, and MIME.
- *WS-Inspection* [15] comprises a simple XML language and related conventions for locating service descriptions published by a service provider. A WS-Inspection language (WSIL) document can contain a collection of service descriptions and links to other sources of service descriptions. A service description is usually a URL to a WSDL document; occasionally, a service description can be a reference to an entry within a Universal Description, Discovery, and Integration (UDDI) [5] registry. A link is usually a URL to another WS-Inspection document; occasionally, a link is a reference to a UDDI entry. With WS-Inspection, a service provider creates a WSIL document and makes the document network accessible. Service requestors use standard Web-based access mechanisms (e.g., HTTP GET) to retrieve this document and discover what services the service provider advertises. WSIL documents can also be organized in different forms of index.

Various other Web services standards have been or are being defined. For example, Web Services Flow Language (WSFL) [6] addresses Web services *orchestration*, that is, the building of sophisticated Web services by composing simpler Web services.

The Web services framework has two advantages for our purposes. First, our need to support the dynamic discovery and composition of services in heterogeneous environments necessitates mechanisms for registering and discovering interface definitions and endpoint implementation descriptions and for dynamically generating proxies based on (potentially multiple) bindings for specific interfaces. WSDL supports this requirement by providing a standard mechanism for defining interface definitions separately from their embodiment within a particular binding (transport protocol and data encoding format). Second, the widespread adoption of Web services mechanisms means that a framework based on Web services can exploit numerous tools and extant services, such as WSDL processors that can generate language bindings for a variety of languages (e.g., Web Services Invocation Framework: WSIF [53]), workflow systems that sit on top of WSDL, and hosting environments for Web services (e.g., Microsoft .NET and Apache Axis). We emphasize that the use of Web services does not imply the use of SOAP for all communications. If needed, alternative transports can be used, for example to achieve higher performance or to run over specialized network protocols.

4 An Open Grid Services Architecture

We have argued that within internal enterprise IT infrastructures, SP-enhanced IT infrastructures, and multi-organizational Grids, computing is increasingly concerned with the creation, management, and application of dynamic ensembles of resources and services (and people)—what we call *virtual organizations* [34]. Depending on context, these ensembles can be small or large, short-lived or long-lived, single institutional or multi-institutional, and homogeneous or heterogeneous. Individual ensembles may be structured hierarchically from smaller systems and may overlap in membership.

We assert that regardless of these differences, developers of applications for VOs face common requirements as they seek to deliver QoS—whether measured in terms of common security semantics, distributed workflow and resource management, coordinated fail-over, problem determination services, or other metrics—across a collection of resources with heterogeneous and often dynamic characteristics.

We now turn to the nature of these requirements and the mechanisms required to address them in practical settings. Extending our analysis in [34], we introduce an Open Grid Services Architecture that supports the creation, maintenance, and application of ensembles of services maintained by VOs.

We start our discussion with some general remarks concerning the utility of a service-oriented Grid architecture, the importance of being able to virtualize Grid services, and essential service characteristics. Then, we introduce the specific aspects that we standardize in our definition of what we call a *Grid service*. We present more technical details in Section 6 (and in [66]).

4.1 Service Orientation and Virtualization

When describing VOs, we can focus on the physical resources being shared (as in [34]) or on the services supported by these resources. (A *service* is a network-enabled entity that provides some capability. The term object could arguably also be used, but we avoid that term due to its overloaded meaning.) In OGSA, we focus on *services*: computational resources, storage resources, networks, programs, databases, and the like are all represented as services.

Regardless of our perspective, a critical requirement in a distributed, multiorganizational Grid environment is for mechanisms that enable interoperability [34]. In a service-oriented view, we can partition the interoperability problem into two subproblems, namely the definition of service interfaces and the identification of the protocol(s) that can be used to invoke a particular interface—and, ideally, agreement on a standard set of such protocols.

A service-oriented view allows us to address the need for standard interface definition mechanisms, local/remote transparency, adaptation to local OS services, and uniform service semantics. A service-oriented view also simplifies virtualization—that is, the encapsulation behind a common interface of diverse implementations. Virtualization allows for consistent resource access across multiple heterogeneous platforms with local or remote location transparency, and enables mapping of multiple logical resource instances onto the same physical resource and management of resources within a VO based on composition from lower-level resources. Virtualization allows the composition of services to form more sophisticated services—without regard for how the services being composed are implemented. Virtualization of Grid services also underpins the ability to map common service semantic behavior seamlessly onto native platform facilities.

Virtualization is easier if service functions can be expressed in a standard form, so that any implementation of a service is invoked in the same manner. WSDL, which we adopt for this purpose, supports a service interface *definition* that is distinct from the protocol bindings used for

service *invocation*. WSDL allows for multiple bindings for a single interface, including distributed communication protocol(s) (e.g., HTTP) as well as locally optimized binding(s) (e.g., local IPC) for interactions between request and service processes on the same host. Other binding properties may include reliability (and other forms of QoS) as well as authentication and delegation of credentials. The choice of binding should always be transparent to the requestor with respect to service invocation semantics—but not with respect to other things: for example, a requestor should be able to choose a particular binding for performance reasons.

The service interface definition and access binding are also distinct from the *implementation* of the functionality of the service. A service can support multiple implementations on different platforms, facilitating seamless overlay not only to native platform facilities but also, via the nesting of service implementations, to virtual ensembles of resources. Depending on the platform and context, we might use the following implementation approaches.

1. We can use a reference implementation constructed for full portability across multiple platforms to support the execution environment (container) for hosting a service.
2. On a platform possessing specialized native facilities for delivering service functionality, we might map from the service interface definition to the native platform facilities.
3. We can also apply these mechanisms recursively so that a higher-level service is constructed by the composition of multiple lower-level services, which themselves may either map to native facilities or decompose further. The service implementation then dispatches operations to lower-level services (see also Section 4.4)

As an example, consider a distributed trace facility that records trace records to a repository. On a platform that does not support a robust trace facility, a reference implementation can be created and hosted in a service execution environment for storing and retrieving trace records on demand. On a platform already possessing a robust trace facility, however, we can integrate the distributed trace service capability with the native platform trace mechanism, thus leveraging existing operational trace management tools, auxiliary offload, dump/restore, and the like, while semantically preserving the logical trace stream through the distributed trace service. Finally, in the case of a higher-level service, trace records obtained from lower-level services would be combined and presented as the integrated trace facility for the service.

Central to this virtualization of resource behaviors is the ability to adapt to operating system functions on specific hosts. A significant challenge when developing these *mappings* is to enable exploitation of native capabilities—whether concerned with performance monitoring, workload management, problem determination, or enforcement of native platform security policy—so that the Grid environment does not become the least common denominator of its constituent pieces. Grid service discovery mechanisms are important in this regard, allowing higher-level services to discover what capabilities are supported by a particular implementation of an interface. For example, if a native platform supports reservation capabilities, an implementation of a resource management interface (e.g., GRAM [25, 31]) can exploit those capabilities.

Thus, our service architecture supports *local and remote transparency with respect to service location and invocation*. It also provides for *multiple protocol bindings* to facilitate localized optimization of services invocation when the service is hosted locally with the service requestor, as well as to enable protocol negotiation for network flows across organizational boundaries where we may wish to choose between several interGrid protocols, each optimized for a different purpose. Finally, we note that an implementation of a particular Grid service interface may map to native, nondistributed, platform functions and capabilities.

4.2 Service Semantics: The Grid Service

Our ability to virtualize and compose services depends on more than standard interface definitions. We also require standard semantics for service interactions so that, for example, different services follow the same conventions for error notification. To this end, OGSA defines what we call a *Grid service*: a Web service that provides a set of well-defined interfaces and that follows specific conventions. The interfaces address discovery, dynamic service creation, lifetime management, notification, and manageability; the conventions address naming and upgradeability. We expect also to address authorization and concurrency control as OGSA evolves. Two other important issues, authentication and reliable invocation, are viewed as service protocol bindings and are thus external to the core Grid service definition, but must be addressed within a complete OGSA implementation. This separation of concerns increases the generality of the architecture without compromising functionality.

The interfaces and conventions that define a Grid service are concerned, in particular, with behaviors related to the management of *transient service instances*. VO participants typically maintain not merely a static set of persistent services that handle complex activity requests from clients. They often need to instantiate new transient service instances dynamically, which then handle the management and interactions associated with the state of particular requested activities. When the activity's state is no longer needed, the service can be destroyed. For example, in a videoconferencing system, the establishment of a videoconferencing session might involve the creation of service instances at intermediate points to manage end-to-end data flows according to QoS constraints. Or, in a Web serving environment, service instances might be instantiated dynamically to provide for consistent user response time by managing application workload through dynamically added capacity. Other examples of transient service instances might be a query against a database, a data mining operation, a network bandwidth allocation, a running data transfer, and an advance reservation for processing capability. (These examples emphasize that service instances can be extremely lightweight entities, created to manage even short-lived activities.) Transience has significant implications for how services are managed, named, discovered, and used.

4.2.1 Upgradeability Conventions and Transport Protocols

Services within a complex distributed system must be independently *upgradeable*. Hence, versioning and compatibility between services must be managed and expressed so that clients can discover not only specific service versions but also compatible services. Further, services (and the hosting environments in which they run) must be upgradeable without disrupting the operation of their clients. For example, an upgrade to the hosting environment may change the set of network protocols that can be used to communicate with the service, and an upgrade to the service itself may correct errors or even enhance the interface. Hence, OGSA defines conventions that allow us to identify when a service changes and when those changes are backwardly compatible with respect to interface and semantics (but not necessarily network protocol). OGSA also defines mechanisms for refreshing a client's knowledge of a service, such as what operations it supports or what network protocols can be used to communicate with the service. A service's description indicates the protocol binding(s) that can be used to communicate with the service. Two properties will often be desirable in such bindings.

- *Reliable service invocation*. Services interact with one another by the exchange of messages. In distributed systems prone to component failure, however, one can never guarantee that a message has been delivered. The existence of internal state makes it important to be able to guarantee that a service has received a message either once or not at all. From this foundation one can build a broad range of higher-level per-operation semantics, such as transactions.

- *Authentication*. Authentication mechanisms allow the identity of individuals and services to be established for policy enforcement. Thus, one will often desire a transport protocol that provides for mutual authentication of client and service instance, as well as the delegation of proxy credentials. From this foundation one can build a broad range of higher-level authorization mechanisms.

4.2.2 Standard Interfaces

The interfaces (in WSDL terms, portTypes) that define a Grid service are listed in Table 1, introduced here, and described in more detail in Section 6 (and in [66]). Note that while OGSA defines a variety of behaviors and associated interfaces, all but one of these interfaces (*GridService*) are optional.

Table 1: Proposed OGSA Grid service interfaces (see text for details). The names provided here will likely change in the future. Interfaces for authorization, policy management, manageability, and likely other purposes remain to be defined.

PortType	Operation	Description
GridService	FindServiceData	Query a variety of information about the Grid service instance, including basic introspection information (handle, reference, primary key, home handleMap: terms to be defined), richer per-interface information, and service-specific information (e.g., service instances known to a registry). Extensible support for various query languages.
	SetTerminationTime	Set (and get) termination time for Grid service instance
	Destroy	Terminate Grid service instance
Notification-Source	SubscribeTo-NotificationTopic	Subscribe to notifications of service-related events, based on message type and interest statement. Allows for delivery via third party messaging services.
Notification-Sink	DeliverNotification	Carry out asynchronous delivery of notification messages
Registry	RegisterService	Conduct soft-state registration of Grid service handles
	UnregisterService	Deregister a Grid service handle
Factory	CreateService	Create new Grid service instance
HandleMap	FindByHandle	Return Grid Service Reference currently associated with supplied Grid Service Handle

Discovery. Applications require mechanisms for discovering available services and for determining the characteristics of those services so that they can configure themselves and their requests to those services appropriately. We address this requirement by defining

- a standard representation for *service data*, that is, information about Grid service instances, which we structure as a set of named and typed XML elements called *service data elements*, encapsulated in a standard container format;

- a standard operation, FindServiceData (within the required *GridService* interface), for retrieving service data from individual Grid service instances ("pull" mode access; see the *NotificationSource* interface below for "push" mode access); and
- standard interfaces for registering information about Grid service instances with registry services (*Registry*) and for mapping from "handles" to "references" (*HandleMap*—to be explained in Section 6, when we discuss naming).

Dynamic service creation. The ability to dynamically create and manage new service instances is a basic tenet of the OGSA model and necessitates the existence of service creation services. The OGSA model defines a standard interface (*Factory*) and semantics that any service creation service must provide.

Lifetime management. Any distributed system must be able to deal with inevitable failures. In a system that incorporates transient, stateful service instances, mechanisms must be provided for *reclaiming services and state associated with failed operations*. For example, termination of a videoconferencing session might also require the termination of services created at intermediate points to manage the flow. We address this requirement by defining two standard operations: *Destroy* and *SetTerminationTime* (within the required *GridService* interface), for explicit destruction and soft state lifetime management of Grid service instances, respectively. (*Soft state protocols* [59, 69] allow state established at a remote location to be discarded eventually, unless refreshed by a stream of subsequent "keepalive" messages. Such protocols have the advantages of being both resilient to failure—a single lost message need not cause irretrievable harm—and simple: no reliable "discard" protocol message is required.)

Notification. A collection of dynamic, distributed services must be able to notify each other asynchronously of interesting changes to their state. OGSA defines common abstractions and service interfaces for subscription to (*NotificationSource*) and delivery of (*NotificationSink*) such *notifications*, so that services constructed by the composition of simpler services can deal with notifications (e.g., for errors) in standard ways. The *NotificationSource* interface is integrated with service data, so that a notification request is expressed as a request for subsequent "push" mode delivery of service data. (We might refer to the capabilities provided by these interfaces as an event service [10], but we avoid that term due to its overloaded meaning.)

Other interfaces. We expect to define additional standard interfaces in the near future, to address issues such as authorization, policy management, concurrency control, and the monitoring and management of potentially large sets of Grid service instances.

4.3 The Role of Hosting Environments

OGSA defines the semantics of a Grid service instance: how it is created, how it is named, how its lifetime is determined, how to communicate with it, and so on. However, while OGSA is prescriptive on matters of basic behavior, it does not place requirements on what a service does or how it performs that service. In other words, OGSA does not address issues of implementation programming model, programming language, implementation tools, or execution environment.

In practice, Grid services are instantiated within a specific execution environment or *hosting environment*. A particular hosting environment defines not only implementation programming model, programming language, development tools, and debugging tools, but also how an implementation of a Grid service meets its obligations with respect to Grid service semantics.

Today's e-science Grid applications typically rely on *native operating system processes* as their hosting environment, with for example creation of a new service instance involving the creation of a new process. In such environments, a service itself may be implemented in a variety of languages such as C, C++, Java, or Fortran. Grid semantics may be implemented directly as part

of the service, or provided via a linked library [39]. Typically semantics are not provided via external services, beyond those provided by the operating system. Thus, for example, lifetime management functions must addressed within the application itself, if required.

Web services, on the other hand, may be implemented on more sophisticated *container or component-based* hosting environments such as J2EE, Websphere, .NET, and Sun One. Such environments define a framework (container) within which components adhering to environment-defined interface standards can be instantiated and composed to build complex applications. Compared with the low levels of functionality provided by native hosting environments, container/component hosting environments tend to offer superior programmability, manageability, flexibility, and safety. Consequently, component/container based hosting environments are seeing widespread use for building e-business services. In the OGSA context, the container (hosting environment) has primary responsibility for ensuring that the services it supports adhere to Grid service semantics, and thus OGSA may motivate modifications or additions to the container/component interface.

By defining service semantics, OGSA specifies interactions between services in a manner independent of any hosting environment. However, as the above discussion highlights, successful implementation of Grid services can be facilitated by specifying baseline characteristics that all hosting environments must possess, defining the "internal" interface from the service implementation to the global Grid environment. These characteristics would then be rendered into different implementation technologies (e.g., J2EE or shared libraries).

A detailed discussion of hosting environment characteristics is beyond the scope of this article. However, we can expect a hosting environment to address mapping of Grid-wide names (i.e., Grid service handles) into implementation-specific entities (C pointers, Java object references, etc.); dispatch of Grid invocations and notification events into implementation-specific actions (events, procedure calls); protocol processing and the formatting of data for network transmission; lifetime management of Grid service instances; and inter-service authentication.

4.4 Using OGSA Mechanisms to Build VO Structures

Applications and users must be able to create transient services and to discover and determine the properties of available services. The OGSA *Factory*, *Registry*, *GridService*, and *HandleMap* interfaces support the creation of transient service instances and the discovery and characterization of the service instances associated with a VO. (In effect, a registry service—a service instance that supports the *Registry* interface for registration and the *GridService* interface's FindServiceData operation, with appropriate service data, for discovery—defines the service set associated with a VO.) These interfaces can be used to construct a variety of VO service structures, as illustrated in Figure 2 and described in the following.

Simple hosting environment. A simple execution environment is a set of resources located within a single administrative domain and supporting native facilities for service management: for example, a J2EE application server, Microsoft .NET system, or Linux cluster. In OGSA, the user interface to such an environment will typically be structured as a registry, one or more factories, and a handleMap service. Each factory is recorded in the registry, to enable clients to discover available factories. When a factory receives a client request to create a Grid service instance, the factory invokes hosting-environment-specific capabilities to create the new instance, assigns it a handle, registers the instance with the registry, and makes the handle available to the handleMap service. The implementations of these various services map directly into local operations.

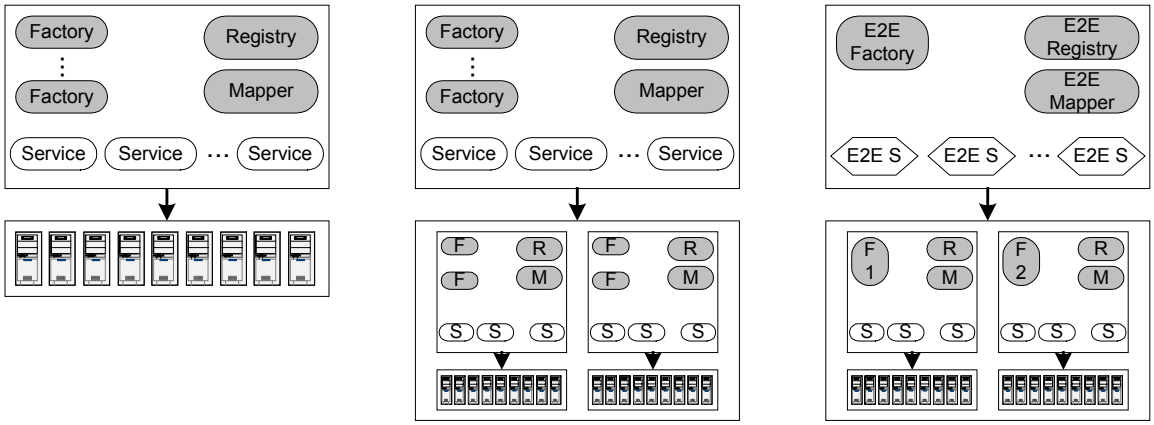

Figure 2: Three different VO structures, as described in the text. From left to right: simple hosting environment, virtual hosting environment, and collective services.

Virtual hosting environment. In more complex environments, the resources associated with a VO will span heterogeneous, geographically distributed "hosting environments." (For example, in Figure 2, these resources span two simple hosting environments.) Nevertheless, this "virtual hosting environment" (which corresponds, perhaps, to the set of resources associated with a B2B partnership) can be made accessible to a client via exactly the same interfaces as were used for the hosting environment just described. We create one or more "higher-level" factories that delegate creation requests to lower-level factories. Similarly, we create a higher-level registry that knows about the higher-level factories and the service instances that they have created, as well any VO-specific policies that govern the use of VO services. Clients can then use the VO registry to find factories and other service instances associated with the VO, and then use the handles returned by the registry to talk directly to those service instances. The higher-level factories and registry implement standard interfaces and so, from the perspective of the user, are indistinguishable from any other factory or registry.

Note that here, as in the previous example, the registry handle can be used as a globally unique name for the service set maintained by the VO. Resource management policies can be defined and enforced on the platforms hosting VO services, targeting the VO by this unique name.

Collective operations. We can also construct a "virtual hosting environment" that provides VO participants with more sophisticated, virtual, "collective" or "end-to-end" services. In this case, the registry keeps track of and advertises factories that create higher-level service instances. Such instances are implemented by asking lower-level factories to create multiple service instances and by composing the behaviors of those multiple lower-level service instances into that single, higher-level service instance.

These three examples, and the preceding discussion, illustrate how Grid service mechanisms can be used to integrate distributed resources both across virtual multi-organizational boundaries and within internal commercial IT infrastructures. In both cases, a collection of Grid services registered with appropriate discovery services can support functional capabilities delivering QoS interactions across distributed resource pools. Applications and middleware can exploit these services for distributed resource management across heterogeneous platforms with local and remote transparency and locally optimized flows.

Implementations of Grid services that map to native platform resources and APIs enable seamless integration of higher-level Grid services such as those just described with underlying platform components. Furthermore, service sets associated with multiple VOs can map to the same

underlying physical resources, with those services represented as logically distinct at one level but sharing physical resource systems at lower levels.

5 Application Example

We illustrate in Figure 3 the following stages in the life of a data mining computation, which we use to illustrate the working of basic remote service invocation, lifetime management, and notification functions.

1. The environment initially comprises (from left to right) four simple hosting environments: one that runs the user application; one that encapsulates computing and storage resources (and that supports two factory services, one for creating storage reservations and the other for creating mining services); and two that encapsulate database services. The "R"s represent local registry services; an additional VO registry service presumably provides information about the location of all depicted services.
2. The user application invokes "create Grid service" requests on the two factories in the second hosting environment, requesting the creation of a "data mining service" that will perform the data mining operation on its behalf, and an allocation of temporary storage for use by that computation. Each request involves mutual authentication of the user and the relevant factory (using an authentication mechanism described in the factory's service description) followed by authorization of the request. Each request is successful and results in the creation of a Grid service instance with some initial lifetime. The new data mining service instance is also provided with delegated proxy credentials that allow it to perform further remote operations on behalf of the user.
3. The newly created data mining service uses its proxy credentials to start requesting data from the two database services, placing intermediate results in local storage. The data mining service also uses notification mechanisms to provide the user application with periodic updates on its status. Meanwhile, the user application generates periodic "keepalive" requests to the two Grid service instances that it has created.
4. The user application fails for some reason. The data mining computation continues for now, but as no other party has an interest in its results, no further keepalive messages are generated.
5. (Not shown in figure) Due to the application failure, keepalive messages cease, and so the two Grid service instances eventually time out and are terminated, freeing the storage and computing resources that they were consuming.

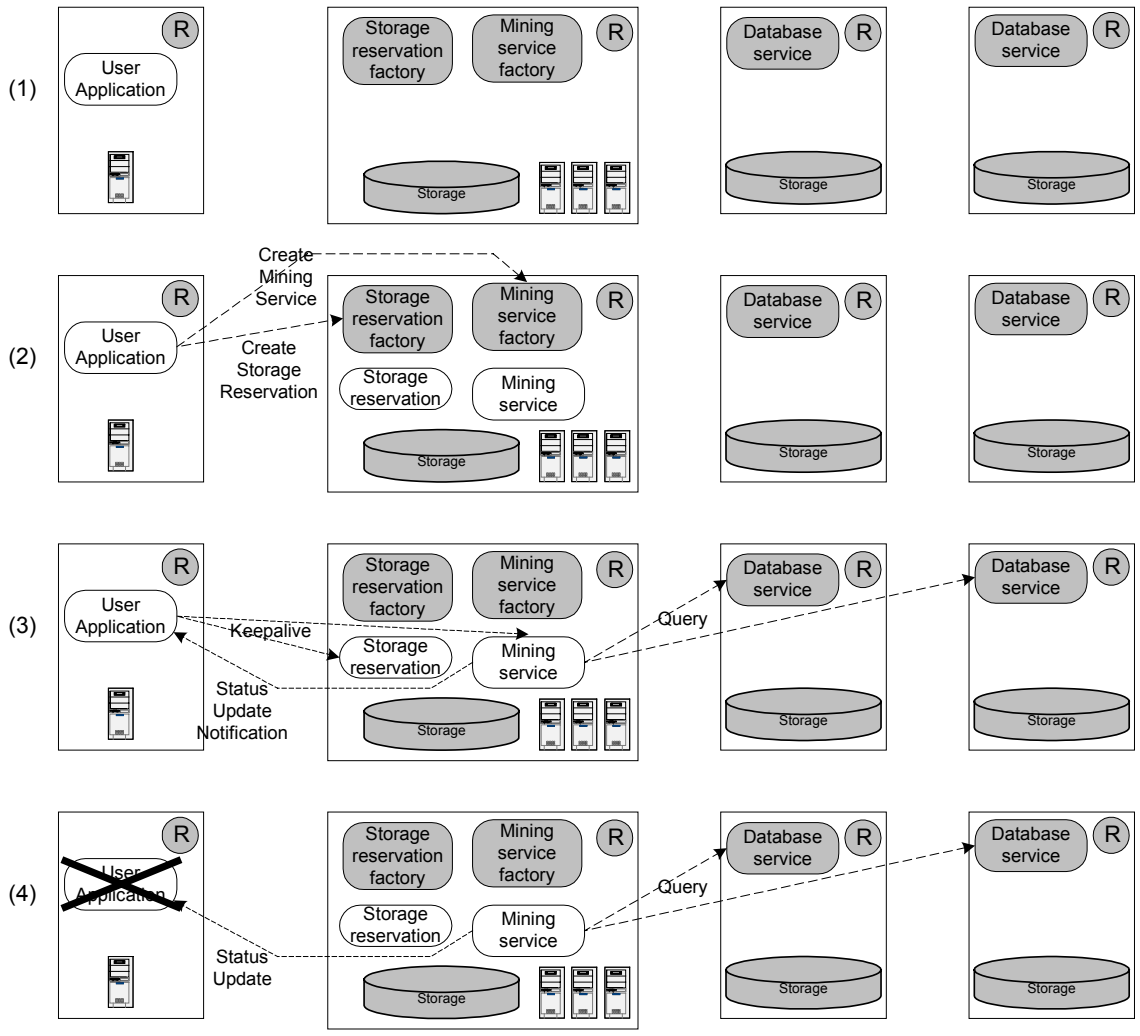

Figure 3: An example of Grid services at work. See text for details.

6 Technical Details

We now present a more detailed description of the Grid service abstraction and associated interfaces and conventions.

6.1 The OGSA Service Model

A basic premise of OGSA is that everything is represented by a *service*: a network enabled entity that provides some capability through the exchange of messages. Computational resources, storage resources, networks, programs, databases, and so forth are all services. This adoption of a uniform service-oriented model means that all components of the environment are virtual.

More specifically, OGSA represents everything as a *Grid service*: a Web service that conforms to a set of conventions and supports standard interfaces for such purposes as lifetime management. This core set of consistent interfaces, from which all Grid services are implemented, facilitates the construction of higher-order services that can be treated in a uniform way across layers of abstraction.

Grid services are characterized (*typed*) by the capabilities that they offer. A Grid service implements one or more *interfaces*, where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages. Grid service interfaces correspond to portTypes in WSDL. The set of portTypes supported by a Grid service, along with some additional information relating to versioning, are specified in the Grid service's *serviceType*, a WSDL extensibility element defined by OGSA.

Grid services can maintain internal state for the lifetime of the service. The existence of state distinguishes one *instance* of a service from another that provides the same interface. We use the term *Grid service instance* to refer to a particular instantiation of a Grid service.

The protocol binding associated with a service interface can define a delivery semantics that addresses, for example, reliability. Services interact with one another by the exchange of messages. In distributed systems prone to component failure, however, one can never guarantee that a message that is sent has been delivered. The existence of internal state can make it important to be able to guarantee that a service has received a message once or not at all, even if failure recovery mechanisms such as retry are in use. In such situations, we may wish to use a protocol that guarantees exactly-once delivery or some similar semantics. Another frequently desirable protocol binding behavior is mutual authentication during communication.

OGSA services can be created and destroyed dynamically. Services may be destroyed explicitly, or may be destroyed or become inaccessible as a result of some system failure such as operating system crash or network partition. Interfaces are defined for managing service lifetime.

Because Grid services are dynamic and stateful, we need a way to distinguish one dynamically created service instance from another. Thus, every Grid service instance is assigned a globally unique name, the *Grid service handle (GSH),* that distinguishes a specific Grid service instance from all other Grid service instances that have existed, exist now, or will exist in the future. (If a Grid service fails and is restarted in such as way as to preserve its state, then it is essentially the same instance, and the same GSH can be used.)

Grid services may be upgraded during their lifetime, for example to support new protocol versions or to add alternative protocols. Thus, the GSH carries no protocol- or instance-specific information such as network address and supported protocol bindings. Instead, this information is encapsulated, along with all other instance-specific information required to interact with a specific service instance, into a single abstraction called a *Grid service reference* (GSR). Unlike a GSH, which is invariant, the GSR(s) for a Grid service instance can change over that service's lifetime. A GSR has an explicit expiration time, or may become invalid at any time during a service's lifetime, and OGSA defines mapping mechanisms, described below, for obtaining an updated GSR.

The result of using a GSR whose lifetime has expired is undefined. Note that holding a valid GSR does not guarantee access to a Grid service instance: local policy or access control constraints (for example maximum number of current requests) may prohibit servicing a request. In addition, the referenced Grid service instance may have failed, preventing the use of the GSR.

As everything in OGSA is a Grid service, there must be Grid services that manipulate the Grid service, handle, and reference abstractions that define the OGSA model. Defining a specific set of services would result in a specific rendering of the OGSA service model. We therefore take a more flexible approach and define a set of basic OGSA interfaces (i.e., WSDL portTypes) for manipulating service model abstractions. These interfaces can then be combined in different ways to produce a rich range of Grid services. Table 1 presents names and descriptions for the Grid service interfaces defined to date. Note that *only the* GridService *interface must be supported by all Grid services.*

6.2 Creating Transient Services: Factories

OGSA defines a class of Grid services that implement an interface that creates new Grid service instances. We call this the *Factory* interface and a service that implements this interface a *factory*. The *Factory* interface's CreateService operation creates a requested Grid service and returns the GSH and initial GSR for the new service instance.

The *Factory* interface does not specify how the service instance is created. One common scenario is for the factory interface to be implemented in some form of hosting environment (such as .NET or J2EE) that provides standard mechanisms for creating (and subsequently managing) new service instances. The hosting environment may define how services are implemented (e.g., language), but this is transparent to service requestors in OGSA, which see only the factory interface. Alternatively, one can construct "higher-level" factories that create services by delegating the request to other factory services (see Section 4.4). For example, in a Web serving environment, a new computer might be integrated into the active pool by asking an appropriate factory service to instantiate a "Web serving" service on an idle computer.

6.3 Service Lifetime Management

The introduction of transient service instances raises the issue of determining the service's lifetime: that is, determining when a service can or should be terminated so that associated resources can be recovered. In normal operating conditions, a transient service instance is created to perform a specific task and either terminates on completion of this task or via an explicit request from the requestor or from another service designated by the requestor. In distributed systems, however, components may fail and messages may be lost. One result is that a service may never see an expected explicit termination request, thus causing it to consume resources indefinitely.

OGSA addresses this problem through a soft state approach [23, 69] in which Grid service instances are created with a specified lifetime. The initial lifetime can be extended by a specified time period by explicit request of the client or another Grid service acting on the client's behalf (subject of course to the policy of the service). If that time period expires without having received a re-affirmation of interest from a client, either the hosting environment or the service instance itself is at liberty to terminate the service instance and release any associated resources.

Our approach to Grid service lifetime management has two desirable properties:

- A client knows, or can determine, when a Grid service instance will terminate. This knowledge allows the client to determine reliably when a service instance has terminated and hence its resources have been recovered, even in the face of system faults (e.g., failures of servers, networks, clients). The client knows exactly how long it has in order to request a final status from the service instance or to request an extension to the service's lifetime. Moreover, it also knows that if system faults occur, it need not continue attempting to contact a service after a known termination time, and that any resources associated with that service would be released after that time—unless another client succeeded in extending the lifetime. In brief, lifetime management enables robust termination and failure detection, by clearly defining the lifetime semantics of a service instance.
- A hosting environment is guaranteed that resource consumption is bounded, even in the face of system failures outside of its control. If the termination time of a service is reached, the hosting environment can reclaim all associated resources.

We implement soft state lifetime management via the SetTerminationTime operation within the required *GridService* interface, which defines operations for negotiating an initial lifetime for a

new service instance, for requesting a lifetime extension, and for harvesting a service instance when its lifetime has expired. We describe each of these mechanisms in turn.

Negotiating an initial lifetime. When requesting the creation of a new Grid service instance through a factory, a client indicates minimum and maximum acceptable initial lifetimes. The factory selects an initial lifetime and returns this to the client.

Requesting a lifetime extension. A client requests a lifetime extension via a SetTerminationTime message to the Grid service instance, which specifies a minimum and maximum acceptable new lifetime. The service instance selects a new lifetime and returns this to the client. Note that a keepalive message is effectively idempotent: the result of a sequence of requests is the same, even if intermediate requests are lost or reordered, as long as not so many requests are lost that the service instance's lifetime expires.

The periodicity of keepalive messages can be determined by the client based on the initial lifetime negotiated with the service instance (and perhaps renegotiated via subsequent keepalive messages) and knowledge about network reliability. The interval size allows tradeoffs between currency of information and overhead.

We note that this approach to lifetime management provides a service with considerable autonomy. Lifetime extension requests from clients are not mandatory: the service can apply its own policies on granting such request. A service can decide at any time to extend its lifetime, either in response to a lifetime extension request by a client or any other reason. A service instance can also cancel itself at any time, for example if resource constraints and priorities dictate that it relinquishes its resources. Subsequent client requests that refer to this service will fail.

The use of absolute time in the *SetTerminationTime* operation—and, for that matter, in Grid service information elements, and commonly in security credentials—implies the existence of a global clock that is sufficiently well synchronized. The Network Time Protocol (NTP) provides standardized mechanisms for clock synchronization and can typically synchronize clocks within at most tens of milliseconds, which is more than adequate for the purposes of lifetime management. Note that we are not implying by these statements a requirement for ordering of events, although we expect to introduce some such mechanisms in future revisions.

6.4 Managing Handles and References

As discussed above, the result of a factory request is a GSH and a GSR. While the GSH is guaranteed to reference the created Grid service instance in perpetuity, the GSR is created with a finite lifetime and may change during the service's lifetime. While this strategy has the advantage of increased flexibility from the perspective of the Grid service provider, it introduces the problem of obtaining a valid GSR once the GSR returned by the service creation operation expires. At its core, this is a bootstrapping problem: how does one establish communication with a Grid service given only its GSH? We describe here how these issues are addressed in the Grid service specification as of June 2002, but note that this part of the specification is likely to evolve in the future, at a minimum to support multiple handle representations and handle mapping services.

The approach taken in OGSA is to define a handle-to-reference mapper interface (*HandleMap*). The operations provided by this interface take a GSH and return a valid GSR. Mapping operations can be access controlled and thus a mapping request may be denied. An implementation of the *HandleMap* interface may wish to keep track of what Grid service instances are actually in existence and not return references to instances that it knows have terminated. However, possession of a valid GSR does not assure that a Grid service instance can be contacted: the service may have failed or been explicitly terminated between the time the GSR

was given out and the time that it was used. (Obviously, if termination of a service is scheduled, it is desirable to represent this in the GSR lifetime, but it is not required.)

By introducing the *HandleMap* interface, we partition the general problem of obtaining a GSR for an arbitrary service into two more specific subproblems:

1) Identifying a handleMap service that contains the mapping for the specified GSH, and
2) Contacting that handleMap to obtain the desired GSR.

We address these two subproblems in turn. To ensure that we can always map a GSH to a GSR, we require that every Grid service instance be registered with at least one handleMap, which we call the *home* handleMap. By structuring the GSH to include the home handleMap's identity, we can easily and scalably determine which handleMap to contact to obtain a GSR for a given GSH. Hence, unique names can be determined locally, thus avoiding scalability problems associated with centralized name allocation services—although relying on the Domain Name System [52]. Note that GSH mappings can also live in other handleMaps. However, every GSH must have exactly one home handleMap.

How do we identify the home handleMap within a GSH? Any service that implements the *HandleMap* interface is a Grid service, and as such will have a GSH. If we use this name in constructing a GSH, however, then we are back in the same position of trying to obtain a GSR from the handleMap service's GSH. To resolve this bootstrapping problem, we need a way to obtain the GSR for the handleMap without requiring a handleMap! We accomplish this by requiring that all home handleMap services be identified by a URL and support a bootstrapping operation that is bound to a single, well-known protocol, namely, HTTP (or HTTPS). Hence, instead of using a GSR to describe what protocols should be used to contact the handleMap service, an HTTP GET operation is used on the URL that points to the home handleMap, and the GSR for the handleMap, in WSDL form, is returned.

Note that a relationship exists between services that implement the *HandleMap* and *Factory* interfaces. Specifically, the GSH returned by a factory request must contain the URL of the home handleMap, and the GSH/GSR mapping must be entered and updated into the handleMap service. The implementation of a factory must decide what service to use as the home handleMap. Indeed a single service may implement both the *Factory* and *HandleMap* interfaces.

Current work within GGF is revising this Grid service component to allow for other forms of handles and mappers/resolvers and/or to simplify the current handle and resolver.

6.5 Service Data and Service Discovery

Associated with each Grid service instance is a set of *service data*, a collection of XML elements encapsulated as service data elements. The packaging of each element includes a name that is unique to the Grid service instance, a type, and time-to-live information that a recipient can use for lifetime management.

The obligatory *GridService* interface defines a standard WSDL operation, FindServiceData, for querying and retrieving service data. This operation requires a simple "by name" query language, and is extensible to allow for the specification of the query language used, which may be for example Xquery [20].

The Grid service specification defines for each Grid service interface a set of zero or more service data elements that must be supported by any Grid service instance that supports that interface. Associated with the *GridService* interface, and thus obligatory for any Grid service instance, are a set of elements containing basic information about a Grid service instance, such as its GSH, GSR, primary key, and home handleMap.

One application of the *GridService* interface's FindServiceData operation is service discovery. Our discussion above assumed that one has a GSH that represents a desired service. But how does one obtain the GSH in the first place? This is the essence of *service discovery*, which we define here as the process of identifying a subset of GSHs from a specified set based on GSH attributes such as the interfaces provided, the number of requests that have been serviced, the load on the service, or policy statements such as the number of outstanding requests allowed.

A Grid service that supports service discovery is called a *registry*. A registry service is defined by two things: the *Registry* interface, which provides operations by which GSHs can be registered with the registry service, and an associated service data element used to contain information about registered GSHs. Thus, the *Registry* interface is used to register a GSH and the *GridService* interface's FindServiceData operation is used to retrieve information about registered GSHs.

The *Registry* interface allows a GSH to register with a registry service to augment the set of GSHs that are considered for subsetting. As in MDS-2 [24], a service (or VO) can use this operation to notify interested parties within a VO of its existence and the service(s) that it provides. These interested parties typically include various forms of service discovery services, which collect and structure service information in order to respond efficiently to service discovery requests. As with other stateful interfaces in OGSA, GSH registration is a soft state operation and must be periodically refreshed, thus allowing discovery services to deal naturally with dynamic service availability.

We note that specification of the attributes associated with a GSH is not tied to the registration of a GSH to a service implementing the *GridService* interface. This feature is important because attribute values may be dynamic and there may be a variety of ways in which attribute values may be obtained, including consulting another service implementing the *GridService* interface.

6.6 Notification

The OGSA notification framework allows clients to register interest in being notified of particular messages (the *NotificationSource* interface) and supports asynchronous, one-way delivery of such notifications (*NotificationSink*). If a particular service wishes to support subscription of notification messages, it must support the *NotificationSource* interface to manage the subscriptions. A service that wishes to receive notification messages must implement the *NotificationSink* interface, which is used to deliver notification messages. To start notification from a particular service, one invokes the subscribe operation on the notification source interface, giving it the service GSH of the notification sink. A stream of notification messages then flow from the source to the sink, while the sink sends periodic keepalive messages to notify the source that it is still interested in receiving notifications. If reliable delivery is desired, this behavior can be implemented by defining an appropriate protocol binding for this service.

An important aspect of this notification model is a close integration with service data: a subscription operation is just a request for subsequent "push" delivery of service data that meet specified conditions. (Recall that the FindServiceData operation provides a "pull" model.)

The framework allows both for direct service-to-service notification message delivery, and for integration with various third-party services, such as messaging services commonly used in the commercial world, or custom services that filter, transform, or specially deliver notification messages on behalf of the notification source. Notification semantics are a property of the protocol binding used to deliver the message. For example, a SOAP/HTTP protocol or direct UDP binding would provide point-to-point, best-effort, notification, while other bindings (e.g., some proprietary message service) would provide better than best-effort delivery. A multicast protocol binding would support multiple receivers.

6.7 Change Management

In order to support *discovery* and *change management* of Grid services, Grid service interfaces must be globally and uniquely named. In WSDL, an interface is defined by a portType and is globally and uniquely named by the portType's qname (i.e., an XML namespace as defined by the targetNamespace attribute in the WSDL document's definitions element, and a local name defined by the portType element's name attribute). Any changes made to the definition of a Grid service, either by changing its interface or by making semantically significant implementation changes to the operations, must be reflected through new interface names (i.e., new portTypes and/or serviceTypes). This feature allows clients that require Grid Services with particular properties (either particular interfaces or implementation semantics) to discover compatible services.

6.8 Other Interfaces

We expect in the future to define an optional *Manageability* interface that supports a set of manageability operations. Such operations allow potentially large sets of Grid service instances to be monitored and managed from management consoles, automation tools, and the like. An optional *Concurrency* interface will provide concurrency control operations.

7 Network Protocol Bindings

The Web services framework can be instantiated on a variety of different protocol bindings. SOAP+HTTP with TLS for security is one example, but others can and have been defined. Here we discuss some issues that arise in the OGSA context.

In selecting network protocol bindings within an OGSA context, we must address four primary requirements:

- *Reliable transport*. As discussed above, the Grid services abstraction can require support for reliable service invocation. One way to address this requirement is to incorporate appropriate support within the network protocol binding, as for example in HTTP-R.
- *Authentication and delegation*. As discussed above, the Grid services abstraction can require support for communication of proxy credentials to remote sites. One way to address this requirement is to incorporate appropriate support within the network protocol binding, as for example in TLS extended with proxy credential support.
- *Ubiquity*. The Grid goal of enabling the dynamic formation of VOs from distributed resources means that, in principle, it must be possible for any arbitrary pair of services to interact.
- *GSR Format*. Recall that the Grid Service Reference can take a binding-specific format. One possible GSR format is a WSDL document; CORBA IOR is another.

The successful deployment of large-scale interoperable OGSA implementations would benefit from the definition of a small number of standard protocol bindings for Grid service discovery and invocation. Just as the ubiquitous deployment of the Internet Protocol allows essentially any two entities to communicate, so ubiquitous deployment of such "InterGrid" protocols will allow any two services to communicate. Hence, clients can be particularly simple, since they need to know about only one set of protocols. (Notice that the definition of such standard protocols does not prevent a pair of services from using an alternative protocol, if both support it.) Whether or not such InterGrid protocols can be defined and gain widespread acceptance remains to be seen. In any case, their definition is beyond the scope of this article.

8 Higher-Level Services

The abstractions and services described in this article provide building blocks that can be used to implement a variety of higher-level Grid services. We intend to work closely with the community to define and implement a wide variety of such services that will, collectively, address the diverse requirements of e-business and e-science applications. These are likely to include the following:

- *Distributed data management services*, supporting access to and manipulation of distributed data, whether in databases or files [58]. Services of interest include database access, data translation, replica management, replica location, and transactions.
- *Workflow services*, supporting the coordinated execution of multiple application tasks on multiple distributed Grid resources.
- *Auditing services*, supporting the recording of usage data, secure storage of that data, analysis of that data for purposes of fraud and intrusion detection, and so forth.
- *Instrumentation and monitoring services*, supporting the discovery of "sensors" in a distributed environment, the collection and analysis of information from these sensors, the generation of alerts when unusual conditions are detected, and so forth.
- *Problem determination services for distributed computing,* including dump, trace, and log mechanisms with event tagging and correlation capabilities.
- *Security protocol mapping services,* enabling distributed security protocols to be transparently mapped onto native platform security services for participation by platform resource managers not implemented to support the distributed security authentication and access control mechanism.

The flexibility of our framework means that such services can be implemented and composed in a variety of different ways. For example, a coordination service that supports the simultaneous allocation and use of multiple computational resources can be instantiated as a service instance, linked with an application as a library, or incorporated into yet higher-level services.

It appears straightforward to re-engineer the resource management, data transfer, and information service protocols used within the current Globus Toolkit to build on these common mechanisms (see Figure 4). In effect, we can *refactor* the design of those protocols, extracting similar elements to exploit commonalities. In the process, we enhance the capabilities of the current protocols and arrive at a common service infrastructure. This process will produce Globus Toolkit 3.0.

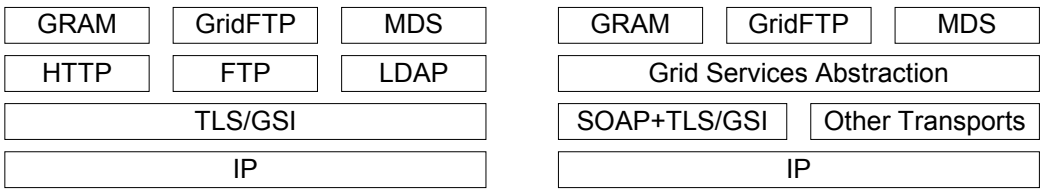

Figure 4: On the left, some current Globus Toolkit protocols; on the right, a potential refactoring to exploit OGSA mechanisms.

9 Related Work

We note briefly some relevant prior and other related work, focusing in particular on issues relating to the secure and reliable remote creation and management of transient, stateful services.

As discussed in Section 3.1, many OGSA mechanisms derive from the Globus Toolkit v2.0: in particular, the factory (GRAM gatekeeper [25]), registry (GRAM reporter [25] and MDS-2 [24]),

use of soft-state registration (MDS-2 [24]), secure remote invocation with delegation (GSI [33]), and reliable remote invocation (GRAM [25]). The primary differences relate to how these different mechanisms are integrated, with OGSA refactoring key design elements so that, for example, common notification mechanisms are used for service registration and service state.

OGSA can be viewed as a distributed object system [21], in the sense that each Grid service instance has a unique identity with respect to the other instances in the system, and each instance can be characterized as state coupled with behavior published through type-specific operations. In this respect, OGSA exploits ideas developed previously in systems such as Eden [7], Argus [48], CORBA [1], SOS [60], Spring [51], Globe [62], Mentat [42], and Legion [41, 43], among others. In contrast to CORBA, OGSA like Web services addresses directly issues of secure interoperability and provides a richer interface definition language. In Grid computing, the Legion group has promoted the use of object models, and we can draw parallels between certain OGSA and Legion constructs, in particular the factory ("Class Object"), handleMap ("Binding Agent"), and timeouts on bindings. However, we also note that OGSA is nonprescriptive on several issues that are often viewed as central to distributed object systems, such as the use of object technologies in implementations, the exposure of inheritance mechanisms at the interface level, and hosting technology.

Soft state mechanisms have been used for management of specific state in network entities within Internet protocols [23, 61, 69] and (under the name "leases") in RMI and Jini [57]. In OGSA, all services and information are open to soft state management. We prefer soft state techniques to alternatives such as distributed reference counting [12] because of their relative simplicity.

Our reliable invocation mechanisms are inspired by those used in Condor [36, 49, 50], which in turn build on much prior work in distributed systems.

As noted in Section 4.3, core OGSA service behaviors will, in general, be supported via some form of hosting environment that simplifies the development of individual components by managing persistence, security, lifecycle management, and so forth. The notion of a hosting environment appears in various operating systems and object systems.

The application of Web services mechanisms to Grid computing has also been investigated and advocated by others (e.g., [35, 37]), with a recent workshop providing overviews of a number of relevant efforts [2]. Gannon et al. [37] discuss the application of various contemporary technologies to e-science applications and propose "application factories" (with WSDL interfaces) as a means of creating application services dynamically. De Roure et al. [26] propose a "Semantic Grid," by analogy to the Semantic Web [11], and propose a range of higher-level services. Work on service-oriented interfaces to numerical software in NetSolve [16, 17] and Ninf [55] is also relevant.

Sun Microsystems' JXTA system [3] addresses several important issues encountered in Grids, including discovery of, and membership in, virtual organizations—what JXTA calls "peer groups." We believe that these abstractions can be implemented within the OGSA framework.

There are connections to be made with component models for distributed and high-performance computing [8, 13, 68], some implementations of which build on Globus Toolkit mechanisms.

10 Summary

We have defined an Open Grid Services Architecture (OGSA) that supports, via standard interfaces and conventions, the creation, termination, management, and invocation of *stateful, transient services as named, managed entities with dynamic, managed lifetime*.

Within OGSA, everything is represented as a *Grid service*, that is, a (potentially transient) service that conforms to a set of conventions (expressed using WSDL) for such purposes as lifetime management, discovery of characteristics, notification, and so on. Grid service implementations can target native platform facilities for integration with, and of, existing IT infrastructures. Standard interfaces for creating, registering, and discovering Grid services can be configured to create various forms of VO structure.

The merits of this service-oriented model are as follows. All components of the environment are virtualized. By providing a core set of consistent interfaces from which all Grid services are implemented, we facilitate the construction of hierarchal, higher-order services that can be treated in a uniform way across layers of abstraction. Virtualization also enables mapping of multiple logical resource instances onto the same physical resource, composition of services regardless of implementation, and management of resources within a VO based on composition from lower-level resources. It is virtualization of Grid services that underpins the ability for mapping common service semantic behavior seamlessly onto native platform facilities.

The development of OGSA represents a natural evolution of the Globus Toolkit 2.0, in which the key concepts of factory, registry, reliable and secure invocation, etc., exist, but in a less general and flexible form than here, and without the benefits of a uniform interface definition language. In effect, OGSA refactors key design elements so that, for example, common notification mechanisms are used for service registration and service state. OSGA also further abstracts these elements so that they can be applied at any level to virtualize VO resources. The Globus Toolkit provides the basis for an open source OGSA implementation, Globus Toolkit 3.0, that supports existing Globus APIs as well as WSDL interfaces, as described at www.globus.org/ogsa.

The development of OGSA also represents a natural evolution of Web services. By integrating support for transient, stateful service instances with existing Web services technologies, OGSA extends significantly the power of the Web services framework, while requiring only minor extensions to existing technologies.

Acknowledgments

We are pleased to acknowledge the many contributions to the Open Grid Services Architecture of Karl Czajkowski, Jeffrey Frey, and Steve Graham. We are also grateful to numerous colleagues for discussions on the topics covered here and/or for helpful comments on versions of this article, in particular Malcolm Atkinson, Brian Carpenter, David De Roure, Andrew Grimshaw, Marty Humphrey, Keith Jackson, Bill Johnston, Kate Keahey, Gregor von Laszewski, Lee Liming, Miron Livny, Norman Paton, Jean-Pierre Prost, Thomas Sandholm, Peter Vanderbilt, and Von Welch.

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Science Foundation; by the NASA Information Power Grid program; and by IBM.

Bibliography

1. Common Object Request Broker: Architecture and Specification, Revision 2.2. Object Management Group Document 96.03.04, 1998.
2. Grid Web Services Workshop. 2001, https://gridport.npaci.edu/workshop/webserv01/agenda.html.
3. JXTA. www.jxta.org.

4. Simple Object Access Protocol (SOAP) 1.1. W3C, Note 8, 2000.
5. UDDI: Universal Description, Discovery and Integration. www.uddi.org.
6. Web Services Flow Language. www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
7. Almes, G.T., Black, A.P., Lazowska, E.D. and Noe, J.D. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, *SE-11* (1). 43--59. 1985.
8. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L. and Parker, S. Toward a Common Component Architecture for High Performance Scientific Computing. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*, 1999.
9. Bal, H.E., Steiner, J.G. and Tanenbaum, A.S. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, *21* (3). 261--322. 1989.
10. Barrett, D.J., Clarke, L.A., Tarr, P.L. and Wise, A.E. A Framework for Event-based Software Integration. *ACM Transactions on Software Engineering and Methodology*, *5* (4). 378-421. 1996.
11. Berners-Lee, T., Hendler, J. and Lassila, O. The Semantic Web. *Scientific American*. 2001.
12. Bevan, D.I., Distributed Garbage Collection Using Reference Counting. In *PARLE Parallel Architectures and Languages Europe*, (1987), Springer Verlag, LNCS 259, 176-187
13. Bramley, R., Gannon, D., Stuckey, T., Villacis, J., Balasubramanian, J., E. Akman, Breg, F., Diwan, S. and Govindaraju, M. Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science and Engineering*, *5* (2). 50-63. 1998.
14. Bray, T., Paoli, J. and Sperberg-McQueen, C.M. The Extensible Markup Language (XML) 1.0. 1998.
15. Brittenham, P. An Overview of the Web Services Inspection Language. 2001, www.ibm.com/developerworks/webservices/library/ws-wsilover.
16. Casanova, H. and Dongarra, J. NetSolve: A Network Server for Solving Computational Science Problems. *International Journal of Supercomputer Applications and High Performance Computing*, *11* (3). 212-223. 1997.
17. Casanova, H., Dongarra, J., Johnson, C. and Miller, M. Application-Specific Tools. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 159-180.
18. Catlett, C. In Search of Gigabit Applications. *IEEE Communications Magazine* (April). 42-51. 1992.
19. Catlett, C. and Smarr, L. Metacomputing. *Communications of the ACM*, *35* (6). 44--52. 1992.
20. Chamberlin, D. Xquery 1.0: An XML Query Language. W3C Working Draft 07, 2001.
21. Chin, R.S. and Chanson, S.T. Distributed Object-based Programming Systems. *ACM Computing Surveys*, *23* (1). 91-124. 1991.
22. Christensen, E., Curbera, F., Meredith, G. and Weerawarana., S. Web Services Description Language (WSDL) 1.1. W3C, Note 15, 2001, www.w3.org/TR/wsdl.
23. Clark, D.D., The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, (1988), ACM Press, 106-114
24. Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C., Grid Information Services for Distributed Resource Sharing. In *10th IEEE International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 181-184
25. Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W. and Tuecke, S. A Resource Management Architecture for Metacomputing Systems. In *4th Workshop on Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, 1998, 62-82.

26. De Roure, D., Jennings, N. and Shadbolt, N. Research Agenda for the Semantic Grid: A Future e-Science Infrastructure. UK National eScience Center, 2002, www.semanticgrid.org.
27. Fallside, D.C. XML Schema Part 0: Primer. W3C, Recommendation, 2001, http://www.w3.org/TR/xmlschema-0/.
28. Foster, I. The Grid: A New Infrastructure for 21st Century Science. *Physics Today*, *55* (2). 42-47. 2002.
29. Foster, I. and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 259-278.
30. Foster, I. and Kesselman, C. (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
31. Foster, I., Kesselman, C., Lee, C., Lindell, R., Nahrstedt, K. and Roy, A., A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proc. International Workshop on Quality of Service*, (1999), 27-36
32. Foster, I., Kesselman, C., Nick, J.M. and Tuecke, S. Grid Services for Distributed Systems Integration. *IEEE Computer*, *35* (6). 2002.
33. Foster, I., Kesselman, C., Tsudik, G. and Tuecke, S. A Security Architecture for Computational Grids. In *ACM Conference on Computers and Security*, 1998, 83-91.
34. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, *15* (3). 200-222. 2001. www.globus.org/research/papers/anatomy.pdf.
35. Fox, G., Balsoy, O., Pallickara, S., Uyar, A., Gannon, D. and Slominski, A. Community Grids. Community Grid Computing Laboratory, Indiana University, 2002.
36. Frey, J., Tannenbaum, T., Foster, I., Livny, M. and Tuecke, S., Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *10th International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 55-66
37. Gannon, D., Bramley, R., Fox, G., Smallen, S., Rossi, A., Ananthakrisnan, R., Bertrand, F., Chiu, K., Farrellee, M., Govindaraju, M., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C. and Rey-Cenvaz, N., Programming the Grid: Distributed Software Components, P2P, and Grid Web Services for Scientific Applications. In *Grid 2001*, (2001)
38. Gasser, M. and McDermott, E., An Architecture for Practical Delegation in a Distributed System. In *Proc. 1990 IEEE Symposium on Research in Security and Privacy*, (1990), IEEE Press, 20-30
39. Getov, V., Laszewski, G.v., Philippsen, M. and Foster, I. Multiparadigm Communications in Java for Grid Computing. *Communications of the ACM*, *44* (10). 118-125. 2001.
40. Graham, S., Simeonov, S., Boubez, T., Daniels, G., Davis, D., Nakamura, Y. and Neyama, R. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Sams, 2001.
41. Grimshaw, A.S., Ferrari, A., Knabe, F.C. and Humphrey, M. Wide-Area Computing: Resource Sharing on a Large Scale. *IEEE Computer*, *32* (5). 29-37. 1999.
42. Grimshaw, A.S., Ferrari, A.J. and West, E.A. Mentat. In *Parallel Programming Using C++*, MIT Press, 1997, 383-427.
43. Grimshaw, A.S. and Wulf, W.A. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, *40* (1). 39-45. 1997.
44. Gullapalli, S., Czajkowski, K., Kesselman, C. and Fitzgerald, S. The Grid Notification Framework. Global Grid Forum, Draft GWD-GIS-019, 2001.

45. Johnston, W. Realtime Widely Distributed Instrumentation Systems. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 75-103.
46. Johnston, W.E., Gannon, D. and Nitzberg, B., Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid. In *Proc. 8th IEEE Symposium on High Performance Distributed Computing*, (1999), IEEE Press
47. Kreger, H. Web Services Conceptual Architecture. IBMTechnical Report WCSA 1.0, 2001.
48. Liskov, B. Distributed Programming in Argus. *Communications of the ACM*, *31* (3). 300-312. 1988.
49. Litzkow, M. and Livny, M. Experience With The Condor Distributed Batch System. In *IEEE Workshop on Experimental Distributed Systems*, 1990.
50. Livny, M. High-Throughput Resource Management. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 311-337.
51. Mitchell, J.G., Gibbons, J., Hamilton, G., Kessler, P.B., Khalidi, Y.Y.A., Kougiouris, P., Madany, P., Nelson, M.N., Powell, M.L. and Radia, S.R., An Overview of the Spring System. In *COMPCON*, (1994), 122-131
52. Mockapetris, P.V. and Dunlap, K., Development of the Domain Name System. In *SIGCOMM*, (1988), ACM, 123-133
53. Mukhi, N. Web Service Invocation Sans SOAP. 2001, http://www.ibm.com/developerworks/library/ws-wsif.html.
54. Mullender, S. (ed.), *Distributed Systems*, 1989.
55. Nakada, H., Sato, M. and Sekiguchi, S. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems*, *15* (5-6). 649-658. 1999.
56. Nick, J.M., Moore, B.B., Chung, J.-Y. and Bowen, N.S. S/390 Cluster Technology: Parallel Sysplex. *IBM Systems Journal*, *36* (2). 172-201. 1997.
57. Oaks, S. and Wong, H. *Jini in a Nutshell*. O'Reilly, 2000.
58. Paton, N.W., Atkinson, M.P., Dialani, V., Pearson, D., Storey, T. and Watson, P. Database Access and Integration Services on the Grid. Manchester University, 2002.
59. Raman, S. and McCanne, S. A Model, Analysis, and Protocol Framework for Soft State-based Communication. *Computer Communication Review*, *29* (4). 1999.
60. Shapiro, M. SOS: An Object Oriented Operating System—Assessment and Perspectives. *Computing Systems*, *2* (4). 287-337. 1989.
61. Sharma, P., Estrin, D., Floyd, S. and Jacobson, V., Scalable Timers for Soft State Protocols. In *IEEE Infocom '97*, (1997), IEEE Press
62. Steen, M.v., Homburg, P., Doorn, L.v., Tanenbaum, A. and Jonge, W.d. Towards Object-based Wide Area Distributed Systems. In Carbrera, L.-F. and Theimer, M. eds. *International Workshop on Object Orientation in Operating Systems*, 1995, 224-227.
63. Steiner, J., Neuman, B.C. and Schiller, J., Kerberos: An Authentication System for Open Network Systems. In *Proc. Usenix Conference*, (1988), 191-202
64. Stevens, R., Woodward, P., DeFanti, T. and Catlett, C. From the I-WAY to the National Technology Grid. *Communications of the ACM*, *40* (11). 50-61. 1997.
65. Thomas, A. Enterprise Java Beans Technology: Server Component Model for the Java Platform. 1998, http://java.sun.com/products/ejb/white_paper.html.
66. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S. and Kesselman, C. Grid Services Specification. 2002, www.globus.org/research/papers/gsspec.pdf.
67. Tuecke, S., Engert, D., Foster, I., Thompson, M., Pearlman, L. and Kesselman, C. Internet X.509 Public Key Infrastructure Proxy Certificate Profile. IETF, Draft draft-ietf-pkix-proxy-01.txt, 2001.

68. Villacis, J., M.Govindaraju, Stern, D., Whitaker, A., Breg, F., Deuskar, P., Temko, B., Gannon, D. and Bramley, R., CAT: A High Performance, Distributed Component Architecture Toolkit for the Grid. In *IEEE Intl Symp. on High Performance Distributed Computing*, (1999)
69. Zhang, L., Braden, B., Estrin, D., Herzog, S. and Jamin, S., RSVP: A new Resource ReSerVation Protocol. In *IEEE Network*, (1993), 8-18