An Overview of Service-oriented Architecture, Web Services and Grid Computing

by Latha Srinivasan and Jem Treadwell
HP Software Global Business Unit

November 3, 2005

Introduction

The use of distributed systems by enterprises and academic organizations has increased exponentially in recent years, enabled by factors such as ready access to the Internet and the World-Wide Web, the maturity and ubiquity of the HTTP protocol, and the improvement in secure communication technology. The style of distribution ranges from groups of co-located homogeneous systems operating as clusters to global-scale grids, made up of vast numbers of systems from a wide array of vendors and supporting an assortment of operating environments.

In the early days, distributed applications communicated using proprietary protocols, and system administrators used ad hoc methods to manage systems that might be across town, on another continent, or anywhere in between. Numerous standards have been developed over the years to ease the costs of deployment and maintenance, with varying degrees of success. Today, the key technologies in distributed systems are service-oriented architecture (SOA), Web services, and grid computing, all of which are seeing significant investment in standardization and increasingly rapid adoption by organizations of all types and sizes.

Discussions that involve the terms *SOA*, *Web services* and *grid* tend to reveal differences of opinion and understanding about their meanings, their relationships, and their place in the enterprise. In this paper we offer a high-level description of each of the technologies, and describe their relationships.

Service-Oriented Architecture

In considering the term *service-oriented architecture*, it is useful to review the key terms[1].

- An **architecture** is a formal description of a system, defining its purpose, functions, externally visible properties, and interfaces. It also includes the description of the system's internal components and their relationships, along with the principles governing its design, operation, and evolution.
- A **service** is a software component that can be accessed via a network to provide functionality to a service requester.
- The term **service-oriented architecture** refers to a style of building reliable distributed systems that deliver functionality as *services*, with the additional emphasis on loose coupling between interacting services.

Technically, then, the term SOA refers to the *design* of a system, not to its implementation. Although it is commonplace for the term to be used in referring to an implementation—for example, in phrases such as "building an SOA"—in this paper we avoid this use, and we use the adjective *service-oriented* in contexts such as "service-oriented environment" or "service-oriented grid."

We regard SOA as an *architectural style* that emphasizes implementation of components as modular *services* that can be discovered and used by clients. Services generally have the following characteristics:

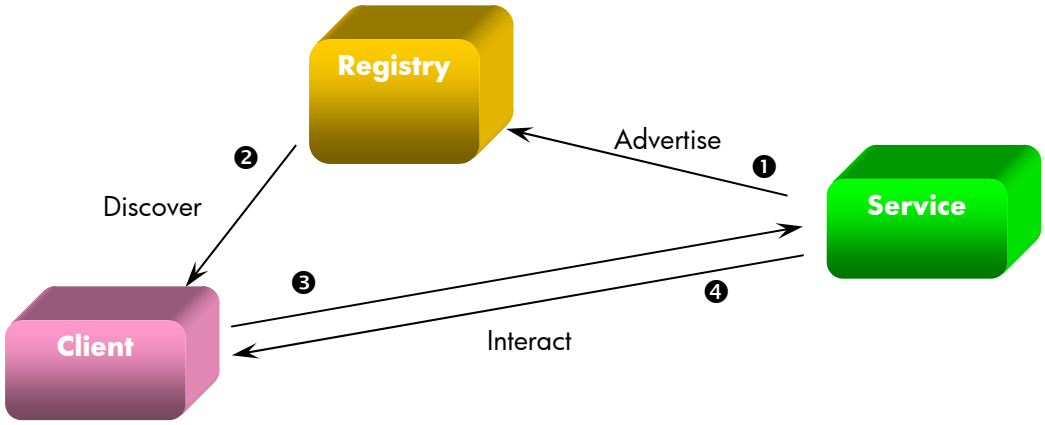
- Services may be individually useful, or they can be integrated—composed—to provide higher-level services. Among other benefits, this promotes re-use of existing functionality.
- Services communicate with their clients by exchanging messages: they are defined by the messages they can accept and the responses they can give.
- Services can participate in a workflow, where the order in which messages are sent and received affects the outcome of the operations performed by a service. This notion is defined as "service choreography."

[1] These definitions are derived from HP's Adaptive Enterprise glossary and the OGSA Glossary of Terms [3].

- Services may be completely self-contained, or they may depend on the availability of other services, or on the existence of a resource such as a database. In the simplest case, a service might perform a calculation such as computing the cube root of a supplied number without needing to refer to any external resource, or it may have pre-loaded all the data that it needs for its lifetime. Conversely, a service that performs currency conversion would need real-time access to exchange-rate information in order to yield correct values.
- Services advertise details such as their capabilities, interfaces, policies, and supported communications protocols. Implementation details such as programming language and hosting platform are of no concern to clients, and are not revealed.

Figure 1 illustrates a simple service interaction cycle, which begins with a service advertising itself through a well-known registry service (1). A potential client, which may or may not be another service, queries the registry (2) to search for a service that meets its needs. The registry returns a (possibly empty) list of suitable services, and the client selects one and passes a request message to it, using any mutually recognized protocol (3). In this example, the service responds (4) either with the result of the requested operation or with a fault message.

Figure 1: Service interaction in a service-oriented environment



The illustration shows the simplest case, but in a real-world setting such as a commercial application the process may be significantly more complex. For example, a given service may support only the HTTPS[2] protocol, be restricted to authorized users, require Kerberos authentication, offer different levels of performance to different users, or require payment for use. Services can provide such details in a variety of ways, and the client can use this information to make its selection. Some attributes, such as payment terms and guaranteed levels of service, may need to be established by a process of negotiation before the client can make use of the service it has selected. And, while this illustration shows a simple synchronous, bi-directional message exchange pattern, a variety of patterns are possible—for example, an interaction may be one-way, or the response may come not from the service to which the client sent the request, but from some other service that completed the transaction.

Loose Coupling

In our definition of SOA, we included the term *loose coupling*. This term implies that the interacting software components minimize their in-built knowledge of each other: they discover the information they need at the time they need it. For example, having learned about a service's existence, a client can discover its capabilities, its policies, its location, its interfaces and its supported protocols. Once

[2] HTTP: Hypertext Transfer Protocol. HTTPS is Secure HTTP—a variation in which message traffic is encrypted.

it has this knowledge, the client can access the service using any mutually acceptable protocol. The word "frictionless" has been used to describe the ultimate goal of loose coupling, and the word aptly conjures up a vision of components that communicate almost without contact.

The benefits of loose coupling include:

- Flexibility: A service can be located on any server, and relocated as necessary. As long as it maintains its registry entry, prospective clients will be able to find it.
- Scalability: Services can be added and removed as demand varies.
- Replaceability: Provided that the original interfaces are preserved, a new or updated implementation of a service can be introduced, and outdated implementations can be retired, without disruption to users.
- Fault tolerance: If a server, a software component, or a network segment fails, or the service becomes unavailable for any other reason, clients can query the registry for alternate services that offer the required functionality, and continue to operate without interruption.

Clearly, all these benefits have great value in a dynamic distributed environment. However, while the vision of loose coupling is appealing, it is some way from broad-based realization. For example, common Web service integrated development environments (IDEs) provide for rapid and easy development of service clients by reading the description of a service and generating a client-side "proxy" or "stub" class with methods that correspond to the service's interfaces. If the interfaces change, the proxy must be regenerated and the client code may need to be altered to invoke the changed methods. While development in this type of environment may be fast and easy, the result is far from frictionless.

Does this mean that services and clients built using such an IDE are *not* loosely coupled? Well, the word "loose" is presumably chosen because it is a relative term—we might say that a truly frictionless relationship is zero-coupled, and adding some friction simply moves it further toward the other end of the scale. The point at which it becomes tightly coupled is a subjective decision.

State and Statelessness

A key notion of loose coupling is *statelessness*—a topic that has been much-discussed and is often mentioned as a critical requirement, sometimes without a clear understanding of its significance.

Simply, the benefits of loose coupling, as listed above, are derived from the fact that a client can choose to go to *any* service that is capable of fulfilling its need. If its choice is restricted to a single service then a tight coupling exists between the client and the server, and the benefits of loose coupling are diminished.

In the simple case of a calculator or a stock-price service it is easy to see that once a client has requested and received information, the transaction is completed, and the client has no particular need to revisit the same service for its future needs. From this perspective, the client and service are loosely coupled.

For a more complex transaction that requires several steps, however, the design of the service might be such that the service retains in its local memory some information ("state") about the first step, expecting to make use of it when the client contacts it for the next step. In this case, the service is "stateful," and the client *must* return to the same service for the next step. This might result in a delay if many clients are using the same service or in a transaction failure if the node hosting the service fails between steps.

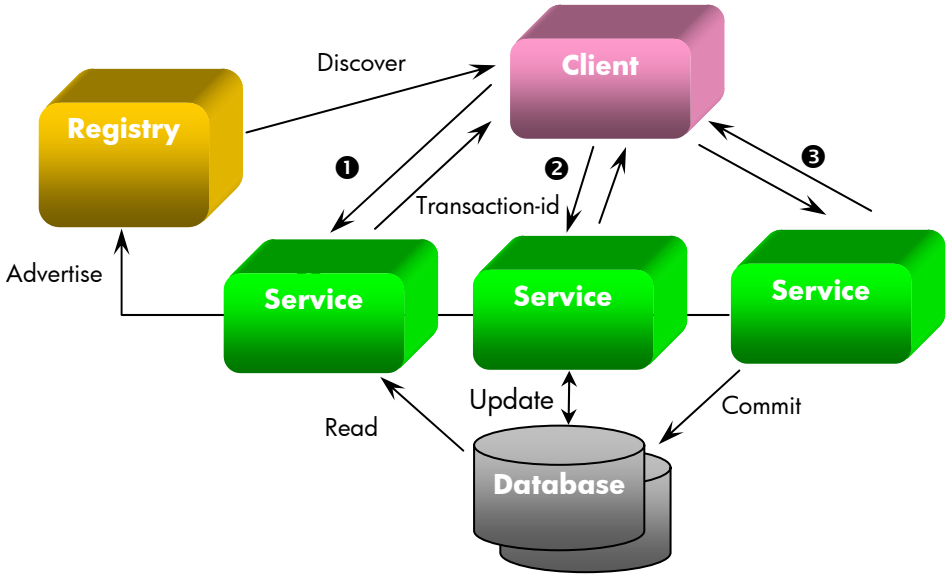
A better approach to the design of the service is for it not to retain state about the transaction, but to be "stateless." This implies that in a multi-step transaction:

- At the end of each intermediate step the service must hand back to the client sufficient state information to enable *any* qualified service to identify and continue the transaction.

- The client must hand the state information to whichever service it selects to process the next step of the transaction.
- The selected service must be able to accept and handle the state information supplied by the client, regardless of whether it processed the earlier steps itself.

Figure 2 shows a client engaged in a three-step transaction with several services, each of which might be capable of handling any part or all of the transaction. The service that handles Step 1 stores the details of the in-progress transaction in the database, and returns requested information to the client, along with a transaction identifier. The client might request confirmation from the user before passing the transaction identifier to another service, which uses it to retrieve the state information from the database and initiates Step 2. This service then updates the database and returns additional information to the client. Finally, the client passes the transaction identifier back to a third service with a request to complete the transaction.

Figure 2: A multi-step client/service interaction



Most non-trivial applications require access to some amount of state information, and the debate is not so much about *whether* state should exist as about *where* it should be stored—the approach outlined above enhances loose coupling by separating the transaction's state from the services that operate on it. In the example, both the account data and the details of the transaction can be considered to be state information, but the account data is permanent, while the transaction details only need to exist while the transaction is in progress. To minimize the amount of state that needs to be passed between the clients and the services, the critical account data and the details of the transaction are held in the database: the common requirement for all participating services is that they must be able to access the database, given a simple token such as a customer's account number, which can easily be passed between the client and the services.

Web Services

Most people are familiar with accessing the Web through a Web browser, which provides a human-oriented interface to information and user-oriented services such as on-line auctions and retail stores. When a user requests a Web page, the request is handled by a remote Web server, which returns the information in *hypertext markup language* (HTML)—a form that allows the browser to present it

using a selection of fonts, colors and pictures, all factors that make it more useful and appealing to a human.

Web services are distributed software components that provide information to *applications* rather than to humans, through an application-oriented interface. The information is structured using *eXtensible Markup Language* (XML), so that it can be parsed and processed easily rather than being formatted for display. In a Web-based retail operation, for example, Web services that may be running on widely separated servers might provide account management, inventory control, shopping cart and credit card authorization services, all of which may be invoked multiple times in the course of a single purchase.

Web services publish details of their functions and interfaces, but they keep their implementation details private; thus a client and a service that support common communication protocols can interact regardless of the platforms on which they run, or the programming languages in which they are written. This makes Web services particularly applicable to a distributed heterogeneous environment.

Although it is common to talk about Web-service "instances," this can be misleading as it implies that a service at a given address is a clone of other "instances." It may be true that at a particular moment several running services are derived from the same source code, but if the code is changed and some services continue to run as they were and others are stopped and re-started, are they still all instances of the same service? What if the change added new operations? The answer is that they are all independently running services that offer common functionality, and the word "instance" is superfluous.

The key specifications used by Web services are:

- XML (eXtensible Markup Language)—a markup language for formatting and exchanging structured data.
- SOAP (originally Simple Object Access Protocol, but technically no longer an acronym)—an XML-based protocol for specifying envelope information, contents and processing information for a message.
- WSDL (Web Services Description Language)—an XML-based language used to describe the attributes, interfaces and other properties of a Web service. A WSDL document can be read by a potential client to learn about the service.

Although a Web service can support any communication protocol, and may offer its clients a choice, the most common is SOAP over either HTTP or HTTPS. This contributes to the appeal of Web services, as HTTP and HTTPS are ubiquitous and typically do not raise issues of firewall traversal in an organization that allows bi-directional HTTP traffic.

Refer to [8, 9] for more information about Web services.

SOA and Web Services: Style vs. Implementation

We initially described SOA without mentioning Web services, and vice versa. This is because they are orthogonal: service-orientation is an architectural style, while Web services are an implementation technology. The two can be used together, and they frequently are, but they are not mutually dependent.

For example, although it is widely considered to be a distributed-computing solution, SOA can be applied to advantage in a single system, where services might be individual processes with well-defined interfaces that communicate using local channels, or in a self-contained cluster, where they might communicate across a high-speed interconnect.

Similarly, while Web services are well-suited as the basis for a service-oriented environment, there is nothing in their definition that *requires* them to embody the SOA principles. While statelessness is often held up as a key characteristic of Web services, there is no technical reason that they should be

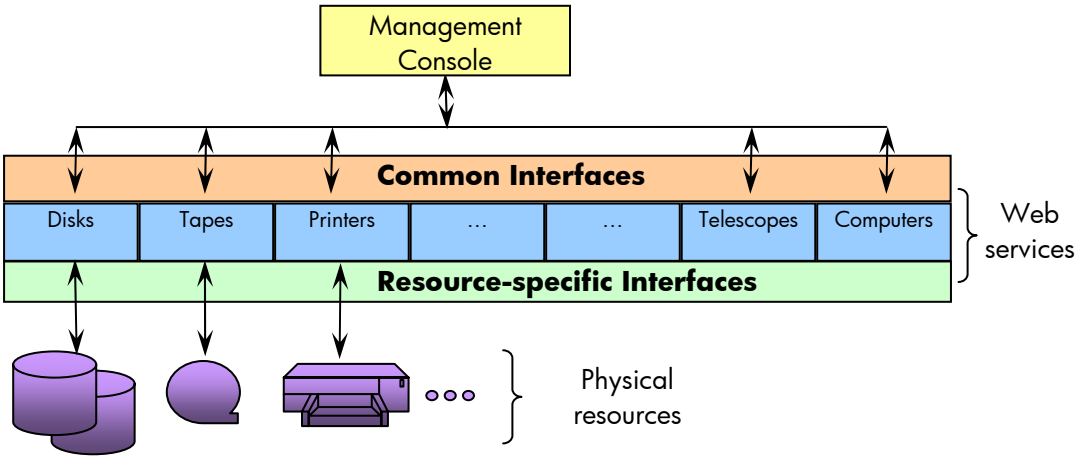
stateless—that would be a design choice of the developer, which may be dictated by the architectural style of the environment in which the service is intended to participate.

Web Services and Virtualization

The ease with which Web services can be implemented and the ability to access them from any platform, local or remote, has led to their rapid adoption by system management bodies as virtualization agents that provide common manageability interfaces to disparate resources. For example, a Web service might be designed to "represent" a particular device or a legacy application, accepting control and monitoring requests through standardized interfaces, communicating with the resource through its native interface, and returning the result to the requester in a standard format.

Figure 3 illustrates how a standards-aware management console (manager) can manage a set of dissimilar resources through a virtualization layer of Web services, each of which exposes standardized interfaces for the manager to use, while communicating with its associated resource through the resource's proprietary interfaces.

Figure 3: Access to physical resources through virtualized interfaces



The Role of Standards

The success of Web services is due in large part to the development of standards by bodies such as the World Wide Web Consortium (W3C) [4], the Organization for the Advancement of Structured Information Standards (OASIS) [5], and the Distributed Management Task Force (DMTF) [6]. These bodies continue to develop and enhance standards for some of the essential underpinnings of Web services, such as describing service interfaces, implementing security and policy management services, and, at a higher level, for implementing business processes.

Standards are critical because they make it possible to develop services that are truly interoperable, composable and interchangeable. By extension, they also promote reusability of components and allow use of standard tooling to handle low-level details such as protocol handling, enabling developers to focus on the unique aspects of their applications. This minimizes the need for new code, improves reliability, and reduces the cost of developing enterprise solutions.

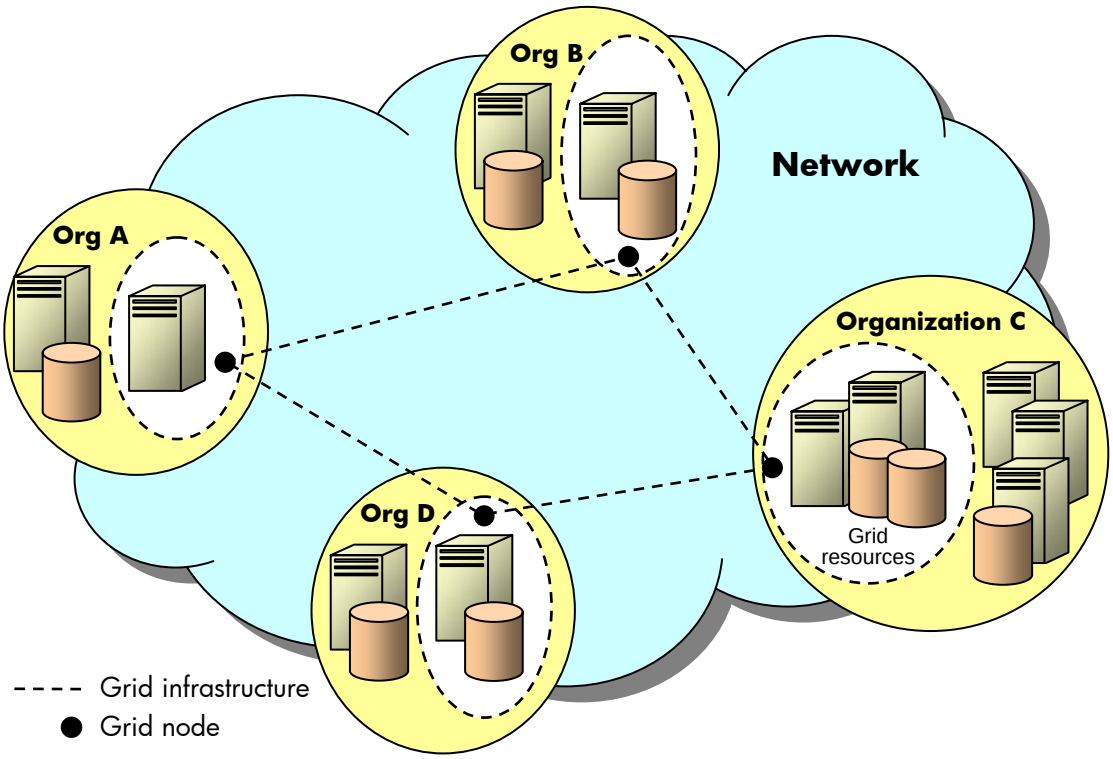
An Overview of Grid Computing

Grid computing is a form of distributed computing in which the use of disparate resources such as compute nodes, storage, applications and data, often spread across different physical locations and administrative domains, is optimized through virtualization and collective management.

Grids are often classified as either *compute* grids, which emphasize the shared use of computational resources, or *data* grids, which support federation, integration and mining of data resources. These distinctions mostly dictate the type of hardware infrastructure needed to support the grid—for example nodes on a data grid may need to provide high-bandwidth network access, while a grid whose primary use is for long-running parallel applications is more in need of high-performance computational nodes. Other classifications, such as *hallway*, *campus*, *enterprise* and *global grid*, indicate the scale or other properties of the grid. In practice, grids tend to display characteristics of various functional classifications, and their scale tends to increase over time, so the distinctions are often blurred.

Figure 4 illustrates a grid to which four connected organizations have contributed computational and storage resources. Each organization must employ security procedures to prevent unauthorized access to its private resources by grid users.

Figure 4: Grids allow resources to be shared across organizational boundaries



Although grid computing was conceived in research organizations to support compute-intensive scientific applications and to share often-massive research databases, enterprises of all types and sizes are beginning to recognize the technology as a foundation for flexible management and use of their internal IT resources, enabling them to better meet business objectives such as minimized cost and increased agility and collaboration.

Grid computing is thus the foundation for collaborative high-performance computing and data sharing, for the adaptive enterprise, and for the vision of utility computing, in which computing resources will become pay-per-use commodities. While the precise definition is debated, common characteristics of grids are that they tend to be large-scale and widely distributed, to require decentralized management, to comprise numerous heterogeneous resources, and to have a transient user population. Hence grids exemplify the need for a highly scalable, reliable, platform-independent architecture that supports secure operation and standardized interfaces for common functions.

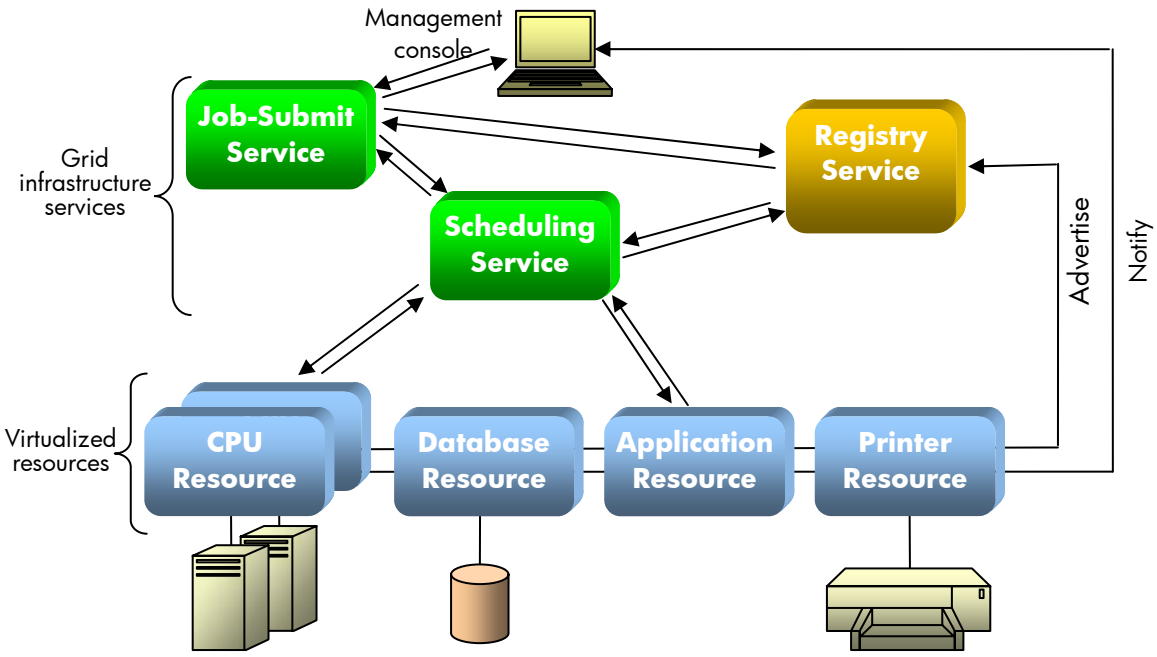
Service-Oriented Grids

Grids are facilitated by the use of grid middleware: software components and protocols that provide the required controlled access to resources. To date, grids have been built using, for the most part, either ad hoc public components or proprietary technologies. While various public and commercial solutions have been successful in their niche areas, each has its strengths and limitations, and they have limited potential as the basis for future-generation grids, which will need to be highly scalable and interoperable to meet the needs of global enterprises.

In recent years, however, it has become clear that there is considerable overlap between the goals of grid computing and the benefits of an SOA based on Web services; the rapid advances in Web services technology and standards have thus provided an evolutionary path from the "stovepipe" architecture of current grids to the standardized, service-oriented, enterprise-class grid of the future.

To illustrate how a service-oriented grid might be constructed, Figure 5 shows a simple grid in which services are used both to virtualize resources and to provide other grid functions.

Figure 5: In service-oriented grids, Web services virtualize resources and provide other functions



In the figure, a single console is used to submit jobs and to manage the grid's resources. The console's management software contacts the registry service to discover the grid resources, and then contacts each resource's representing service to request periodic performance data and to be notified of significant changes in the resource's state—for example, if it becomes unavailable or heavily loaded.

The user submits a job request to the job-submit service, which locates the grid's scheduling service and passes on the request. The scheduler locates and contacts the service that represents the requested application, and requests details of the resources that are required for the job. It then queries the registry to find all suitable resources, and contacts them individually via their associated services to determine availability. If sufficient resources are available the scheduler selects the best available set and passes their details to the application service with a request to begin execution; otherwise, it queues the job and runs it when the required resources become available. Once the job has been completed, the application service reports the result to the scheduler, which notifies the job-submit service. The job-submit service, in turn, notifies the user.

Note that this example is simplified for clarity: a real enterprise-class grid would involve much more sophisticated operation than is shown here, with the net result being a high degree of automation and optimization in the use of resources across the grid's domain.

OGSA and Grid Standards

The convergence of SOA and grid computing is embodied by the Global Grid Forum's (GGF) [7] Open Grid Services Architecture (OGSA).

The OGSA Architecture document [1] describes a service-oriented grid, at a high level, in terms of its required capabilities—including, for example, job execution, resource management, data management, and security. Other OGSA documents, some close to publication and others still in the early stages of planning or development, will define the interfaces and behaviors of a core set of services, and OGSA "profiles" will unambiguously prescribe the use of standards to ensure interoperable implementations. As the OGSA specifications are published and become mature, it is expected that grid middleware suppliers will evolve their products to provide OGSA-conformant functionality and interfaces, resulting in multi-vendor components that grid developers can combine to form grids that are scalable, interoperable, reliable and easily managed.

For information about the planned set of OGSA documents and their development schedules refer to the OGSA Roadmap [2].

It is important to note that, while GGF working groups will define new standards where they are needed, OGSA grids will make extensive use of existing and emerging standards from all segments of the Web services community, with the flexibility to support the needs of different categories of users. For example, OGSA specifications will initially define management capabilities based on the Web Services Distributed Management (WSDM) standard from OASIS, possibly followed by an alternative set of specifications based on DMTF's WS-Management, which has recently begun the standardization process. Security will be based upon WS-Security, also from OASIS. The Web Services Resource Framework (WSRF), which supports representation and manipulation of stateful resources, and WS-Notification, which provides a common publish/subscribe-based event notification mechanism, have been developed by OASIS primarily to address the needs of the grid computing community.

Conversely, many of the standards being defined by GGF, including areas such as negotiation, job submission, and application deployment, are expected to have uses in non-grid environments.

Service-oriented vs. Object-oriented

The notion of the type of coupling (tight or loose) that is advocated by object-oriented systems and service-oriented systems is debated extensively in several forums. Here we draw the distinction between the two in terms of designing large-scale IT systems.

Traditionally, distributed object-oriented systems such as CORBA [15] and DCOM [16] have been designed with the notion of expressing units of software functionality as distributed objects. In this paradigm, clients have static knowledge of the objects, object instances tend to encapsulate data,

and communication is through synchronous invocation of remote methods, using special-purpose protocols. This implies that the communicating objects are tightly coupled, with lock-step method invocations and responses. In this paradigm, an object with some data encapsulated in it is the central theme. Objects tend to be fine-grained implementations of real-life entities.

In a service-oriented environment, on the other hand, services are loosely coupled: they communicate by message exchanges that can be one-way, synchronous or asynchronous. This promotes the design of systems in which it is not necessary that requests and responses are handled by the same set of communicating entities. As an example, consider a client that communicates with service A to initiate a purchase order. Service A could then check inventory by communicating with service B, and service B could then communicate with service C, which issues a purchase order to the client. This is a simple example that illustrates that requests can be made to one service and the corresponding response is sent by another service, underscoring the notions of loose coupling. In this paradigm, the service is the central theme. Services, in general, tend to be coarser-grained than objects in implementing real-life entities.

Despite what are often seen as some significant disadvantages in the current trend towards Web-based architectures, the distributed-object paradigm benefits from factors such as maturity of implementation and efficient communication, and its model of encapsulation of state and behavior is well-suited to some applications. Consequently, object-orientation has gained general acceptance within certain industry sectors: organizations within the financial, healthcare and telecommunications industries, among others, have successfully deployed large-scale object-oriented systems [11].

On the other hand, while the combination of SOA and Web services will clearly be the focus of development for the foreseeable future, it must be recognized that they still have a number of pervasive issues to overcome, including still-immature technology, dissent and some confusion in the area of standards, and the significant performance overhead of serializing, de-serializing and parsing SOAP messages and XML documents on each message exchange. However, the examples that follow illustrate the focus that key industry players are placing on providing accessible Web services environments:

- PayPal has developed a set of APIs [12] that merchants and other users can use to automate certain functions that otherwise may require manual intervention. Examples of functionality that are exposed as Web services are refund-processing, queries against transactions, and secure withdrawal of funds from buyers' accounts. By making use of these services, merchants and retail users can integrate their tools to extend and tailor PayPal's functionality to meet their business needs.
- Google has created a set of APIs [13] that allows users to integrate Google's capabilities such as search and spell-check into their systems. Using these APIs, developers can write programs that integrate with Google services using SOAP.
- Amazon.com has provided a set of Web services [14] that developers can extend to provide inventory-management tools, pricing-configuration tools, store-building tools, etc.

It is important to note that while each architectural style may have inherent strengths and weaknesses, depending on your perspective, both provide important features such as extensibility, aggregation, composability, scalability, resilience and the ability to hide implementation details behind well-defined interfaces, and hence provide the basis for a well-architected distributed application environment.

SOA Design and Development

As we have seen, service-orientation reinforces general software architecture principles such as encapsulation, modularization and separation of the interfaces from their implementations, in addition to introducing concepts such as service choreography, service repositories and message-oriented and event-driven interactions.

To build a system that conforms to SOA principles, we would start by breaking out the different pieces of business logic and work items into *services*. Services are:

- Defined by well-published, implementation-agnostic interfaces.
- Loosely coupled, and promote concepts such as location transparency.
- Entities that encapsulate reusable business functions.

These services encapsulate the capabilities of a software system; they communicate with each other using the different message exchange patterns (one-way, synchronous and asynchronous).

The following list enumerates the steps that should be followed to implement a service-oriented system:

1. Identify the different units of business logic and units of work to be performed.
2. Describe the functionality of the various units of work in terms of services. Implement stateful or stateless services as required by the functionality.
3. Identify the core infrastructure services that each service needs to use, and decide if they are readily available or need to be implemented.
4. Identify the major pieces of common functionality, if any, between the various services, and ensure that the common functionality is captured as reusable, modular services, using subtyping and other extension techniques.
5. Capture the different pieces of functionality exposed by a service in terms of operations modeled on the message exchange pattern, and assign meaningful names to the services to convey the functionality exposed.
6. Define events of interest that the services would be producing or consuming.
7. Create orchestration scripts or workflows to enable service choreography.
8. Publish the services in a registry, or in multiple registries, so that they can be discovered by other entities.

As we discussed earlier, steps 1 through 4 are common in the design of any distributed architecture, whether it is object-oriented or service-oriented. Steps 5 and 6 are followed in object-oriented design as well, although in a different fashion. Steps 7 and 8 are the differentiators between object-oriented design and service-oriented design.

Summary

Web services and a service-oriented style of architecture are widely seen as the basis for a new generation of distributed applications and system management tools. In this paper we have introduced the key concepts, relationships and benefits of these two technologies, and indicated how they can be combined to develop highly scalable application systems that can span management and ownership domains, regardless of the hardware and software platforms deployed in each. We have also attempted to show how the careful development of key standards is increasing the appeal of service-oriented environments, and how they are being used as the basis for development of interoperable next-generation grids.

Acknowledgments

Some parts of this paper may appear in revised form in OGSA explanatory documents published by GGF, to which the authors contributed.

Thanks to Bryan Murray, William Vambenepe and Phil Vitale for their comments and suggestions.

For More Information

In addition to software downloads and technical information such as this paper, HP Dev Resource Central provides access to educational recorded and live webcasts by HP experts and partners. Visit HP's Invent Online to browse the available webcasts. In support of the technologists who work in partnership with HP to produce high-quality, interoperable software, we provide webcasts on web services development, application management, identity management, a variety of related technologies, including XML, WSDL, and SOAP, and more. Among these webcasts is Grids for the enterprise: an introduction to service-oriented grids by the authors of this paper.

Get access to more information online about SOA, standards, grid and more by signing up to receive the HP Software Partner News newsletter. Subscribers receive links to white papers, information on the latest tools and downloads, and technical tips delivered in a monthly email message. Subscribe to get the next newsletter.

Information about HP's work with grid computing is available at http://www.hp.com/go/grid.

References

1. Open Grid Services Architecture (OGSA) v1.0: http://www.gridforum.org/documents/GFD.30.pdf
2. Defining the Grid: A Roadmap for OGSA Standards v1.0: http://www.gridforum.org/documents/GFD.53.pdf
3. OGSA Glossary of Terms v1.0: http://www.gridforum.org/documents/GFD.44.pdf
4. World Wide Web Consortium (W3C): http://www.w3.org
5. Organization for the Advancement of Structured Information Standards (OASIS): http://www.oasis-open.org
6. Distributed Management Task Force (DMTF): http://www.dmtf.org
7. Global Grid Forum: http://www.ggf.org
8. Chatterjee & Webber: Developing Enterprise Web Services – An Architect's Guide (Prentice Hall)
9. WebServices.org: http://www.webservices.org
10. Microsoft® Corporation: Service Orientation and its role in your connected systems strategy: http://download.microsoft.com/download/d/2/5/d2513e64-0dcd-4ef6-89c4-c99ee117936f/serviceorientationwpa4.doc
11. Reinventing the Wheel? CORBA vs. Web Services: http://wwwconf.ecs.soton.ac.uk/archive/00000216/01
12. PayPal Developer Network: http://www.paypal.com/cgi-bin/webscr?cmd=p/pdn/about-outside
13. Google APIs: http://www.google.com/apis/
14. Amazon Web Services Network: www.amazon.com/gp/aws/landing.html
15. CORBA: http://www.corba.org
16. DCOM: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/dcom.asp

V02, 11/2005

invent