Computing and Informatics, Vol. 21, 2002, 321–332

A REVISED ANALYSIS OF THE OPEN GRID SERVICES INFRASTRUCTURE

Dennis Gannon, Kenneth Chiu, Madhusudhan Govindaraju, Aleksander Slominski

Department of Computer Science
Indiana University
Bloomington, IN 47405, USA
e-mail: gannon@cs.indiana.edu

Revised manuscript received 18 November 2002

Abstract. This paper began its life as an unpublished technical review [20] of the proposed Open Grid Services Architecture (OGSA) as described in the papers, "The Physiology of the Grid" [1] by Ian Foster, Carl Kesselman, Jeffrey Nick and Steven Tuecke, and "The Grid Service Specification (Draft 2/15/02) [2]" by Foster, Kesselman, Tuecke and Karl Czajkowski, Jeffrey Frey and Steve Graham. However, much has changed since the publication of the original documents. The architecture has evolved substantially and the vast majority of our initial concerns have been addressed. In this paper we will describe the evolution of the specification from its original form to the current draft of 10/4/02 authored by S. Tuecke, K. Czajkowski, J. Frey, S. Graham, C. Kesselman, and P. Vanderbilt, which is now the central component of the Global Grid Forum Open Grid Service Infrastructure (OGSI) working group which is co-chaired by Steven Tuecke and David Snelling.

Keywords: OGSA, Grid services, web services, OGSI

1 INTRODUCTION

The Open Grid Service Architecture (OGSA) represents a long-overdue effort to define an "architecture" for the Grid. This is an extremely important step and truly represents an important milestone in the evolution of the Grid. From the first large scale production Grid implementation, NASA's Information Power Grid (IPG) [3],

Grids have been defined as a collection of distributed services which have been implemented using a collection of toolkits, such as Globus [4, 5, 6], or distributed operating systems like Legion [7], or distributed resource management frameworks like Condor [8] and others. There are now a number large Grid deployments underway including PPDG [9], NEES Grid [10], DOE Science Grid [11], GriPhyn [12], the U. K. e-Sciences Grid projects, Japan Science Grids and many more. Almost without exception each of these deployments have used some of the existing technology, most notably Globus, but almost all have developed specialized services which have been layered on top of, and often wedged in between, standard services such as security, remote job execution, remote data management and resource discovery and allocation. These Grids tend to be a patchwork of protocols and non-interoperable "standards" and difficult to re-use "implementations". The Global Grid Forum [13] has a dozen different working groups devoted to defining more Grid standards. More significantly, much of the work on Grid implementations has been done without sufficient regard to the evolution of distributed computing in the commercial sector.

1.1 Enter Web Services

The Web Service architecture that has emerged from five years of not-so-successful attempts to define frameworks for Business-to-Business computing is a model of simplicity. It forms the distributed computing foundation of Microsoft's .NET framework, the IBM e-business strategy and many other cooperate initiatives. It completely supports the emergence of an exciting service provider industry that has been the long sought goal for building an e-business marketplace as well as a framework for organizing the distributed information management and computing being done by large enterprises. The Web Services model is based on two simple technologies:

- The Web Services Description Language WSDL [14] that defines the XML Schema and Language used to describe a web service. Each Web Service is an entity, which is defined by ports that are service "endpoints" capable of receiving (and replying to) a set of messages defined by that port's type. Each port is, in fact, a binding of a port type and an access protocol that tells how the messages should be encoded and sent to the port. A service may have several different access points and protocols for each port type.
- The Universal Description, Discovery and Integration (UDDI [15]) and the Web Services Inspection Language (WSIL) [16] provide the mechanism needed to discover WSDL documents. UDDI is a specification for a registry that can be used by a service provider as a place to publish WSDL documents. Clients can then search the registry looking for services and then fetching the WSDL documents needed to access them. However, not all services will be listed on UDDI registries. WSIL provides a simple way to find WSDL documents on a web site. These discovery mechanisms correspond to the Grid Information Service [6] in Globus terms.

In addition, several other standards have been proposed that provide additional features. For example, IBM has proposed the BPEL4WS [18] which is a mechanism for scripting the workflow for integrating multiple services together to accomplish a complex task. A workflow engine acts as the agent that follows the BPEL4WS specification document and contacts each of the services required by the specification following the order (an directed graph) specified. Another is the Web Services Invocation Framework (WSIF) providing a way to dynamically generate service proxies as objects that may be referenced within the language of the client application. The Web Services framework defined by these technologies provides a simple way to describe, encapsulate, advertise and access a service.

2 A BRIEF OVERVIEW OF OGSA

OGSA can be seen as an extension and a refinement of the emerging web services architecture. The designers of the Web Service Description Language (WSDL) anticipated that extensions to the core language would be needed and they provided the hooks to make that possible. The original set of extensions that OGSA were designed included the concept of "service type", which allow us to describe families of services defined by a collections of ports of specified types. However, in the current version, this is no longer needed because it is a proposed standard part of WSDL 1.2. The original Grid Service Specification also provided a mechanism to specify that an instance of a service is an instance of a particular service implementation of a specified service type and a way to say that this service is compatible with others. These extensions provided a mechanism to describe service semantic evolution and versioning. These extensions have been dropped from the current version. The number of required extension to WSDL is now very small. They are

- *ServiceData*. These are descriptions of a service's state and associated metadata for the service.
- S*erviceDataDescription*. These are the formal definitions of the service data elements for the service.
- There are conventions on *PortType* names.
- Two additional concepts have been added that provide better ways to identify and access a service: the Grid Service Reference (GSR) and the Grid Service Handle (GSH).

Actually only the first two items are actual extensions to WSDL. In the sections that follow we will describe each of these concepts, as they form the central core of OGSI.

In the OGSI there are two things that a web service instance must have before it qualifies as a Grid Services instance. First, it must have one or more Grid Services Handles (GSH), which is a type of Grid URI for our service instance. The GSH is not a direct link to the service instance, but rather it may be resolved to a Grid Service Reference (GSR). The GSR can be a WSDL document for the service instance. The

idea is that the handle provides a constant way to locate the current GSR for the service instance, because the GSR may change if the service instance changes or is upgraded. It is important to note that the concept of service instance is defined by the service itself. The GSH always refers to an instance of the service, and from the client's perspective it always appears at the same instance even though its concrete instantiation may have been changed.

The second property that elevates a Grid Service above a garden variety web service is the fact that each Grid Service instance must implement a port whose type is, or is derived from, *GridService*. The service can implement other ports from the standard OGSI family and it may also implement some application specific ports.

The *GridService* port has the following operations:

- *FindServiceData*. This allows a client to discover more information about the service's state, execution environment and addition semantic details that are not available in the GSR. In general, this type of reflection is an important property for services. It can be used to allow the client a standard way to learn more about the service they will use. The exact way this information is conveyed is through *ServiceData* elements associated with the service. We will describe this in greater detail later. The input to this operation is a query, which may be a *queryByServiceDataName* or *queryByXPath* or *queryByXQuery*, where the last two are optional.
- *RequestTerminationAfter*. This allows the client to request that the service instance terminates itself no sooner than a specified time
- *RequestTerminationBefore*. This allows the client to request that the service terminates itself no later than a specified time.
- Destroy. Instruct the service instance to kill itself.

In general, the required *GridService* port is an excellent idea. We have used something similar for our work for a long time and it is very useful. OGSA also defines an additional set of standard, but not required, service ports. These ports define the standard properties required by all distributed systems: messaging, discovery, instance creation and lifetime management. Messaging is handled by the *NotificationSource*, *NotificationSink* and *NotificationSubscription* ports. The intent of this service is to provide a simple publish-subscribe system similar to Java's JMS [17], but based on XML messages. There are many unresolved issues associated with the notification part of the specification and we will return to this later in this paper.

The other OGSI standard Grid Service ports include

- *HandleResolver*. This is a service that provides the mapping between the Grid Service instance's Grid Service Handle (GSH) and the current Grid Service Reference. One can think of it as the "domain name service" for handles and references. Unfortunately what is not defined is the port type that is used to add/remove bindings to/from the service. It is possible to do this via the notification system.

- *Registration.* A registry is a Grid service that maintains a collection of Grid Service Handles, with policies associated with that collection. Clients may query the registry to discover what services are available. Information is extracted from the registry by using the *FindServiceData* method on the *GridService* port. Registries implement the *Registration* port type, which allows a client to store a Grid Service Locator (GSL) from the registry. The committee is still working on the details of how registries work, so we will not discuss it further. One interesting issue that is undergoing active debate is how Registration can be used to implement the concept of service collection.
- *Factory.* A Factory service is a service that can be used to create instances of other services. In Grid applications the factory service can create instances of transient application services. One interacts with a Factory service by providing it with creation lifetime information and an XML document that describes application specific data. This is identical to Factory services we have used and we have found the concept very valuable.

3 THE ROLE OF SERVICE DATA

The power of the World Wide Web lies in its simplicity. In [19] Fielding argues that its success is based on architectural language with few verbs (HTTP Get and Put) and a rich collections of nouns (URIs and URLs) and data which can be universally understood as a document that can be interpreted and rendered by clients. Fielding calls this model Representational State Transfer (REST).

OGSI extends the standard Web Services Model with a REST-like form of standard service introspection based on *ServiceData* elements (SDEs) and *ServiceDataDescription*s (SDDs). A service data element is used to describe service instance state or associated metadata. It is an XML fragment, which takes the form

```
<gsdl:serviceData  name="qname"
       <-- extensibility attribute -->* >
    <-- extensibility element -->*
</gsdl:serviceData>
```

The standard attributes for service data describe its lifetime, i.e. when was it created or last modified and how long it will live. The extensibility elements are called the "service data element values". SDDs describe the SDEs a service supports. SDDs take the form

```
<gsdl:serviceDataDescription
       name="NCName"
       element="qname"
       minOccurs="nonNegativeInteger"?
       maxOccurs=("nonNegativeInteger" | "unbounded")?
       instanceOnly="boolean"?
```

```
       mutability="constant"|"append"|"mutable"?>
    <wsdl:documentation .... />?
    <-- extensibility element --> *
</gsdl:serviceDataDescription>
```

These descriptions tell us what service data element types we can expect to find in instances of a particular service. It includes the name of the element, a name for a schema for the "service data element value", how many instances of that SDE values can be expected, whether it reflects service state or is constant metadata and a description of the contents. To illustrate the idea of SDEs and SDDs we consider a simple example: a Grid Service that monitors a set of task queues such as print queues or job queues on some computing resource. In this case the service would only need one port: the *GridService* port which would contain a service data element called "queueLengths". We can define a queue data element value by a simple schema like

```
xmlns:n1="http://MyResources.com/ns"

<xsd:schema ...targetNamespace=http://MyResources.com/ns
    ...>
    <xsd:element name="queue" type="n1:myQueue"/>
    <xsd:complexType name="myQueue">
       <xsd:sequence>
         <xsd:element name="queueName" type="xsd:string" minOccurs="1"
               maxOccurs="1"/>
          <xsd:element name="size" type="xsd:integer" minOccurs="1"
               maxOccurs="1"/>
       </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

The associated SDD would look like

```
<gsdl:serviceDataDescription
      name="queueLengths"
      element="n1:queue"
      minOccurs="1"
      maxOccurs="unbounded"
      mutability="mutable">
   <wsdl:documentation>
      These values is the current lengths of the monitored queues
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

Actual instances of this SDE would look like

```
<gsdl:serviceData name="tns:queueLengths">
   <n1:queue>
```

```
      <n1:QueueName> main office printer </n1:QueueName>
      <n1:size> 13 </n1:size>
   </n1:queue>
   <n1:queue>
      <n1:QueueName> cpu work queue </n1:QueueName>
      <n1:size> 4 </n1:size>
   </n1:queue>
</gsdl:serviceData>
```

For a client accessing the *GridService* port of the service instance, the way it would find the *queueLengths* is to issue the operation $findServiceData$ with the argument

```
<queryByServiceDataName name="queueLengths" />
```

The result should be to return the current service data element for the queue lengths.

The SDD for our queue length would appear in the GSDL extensions to WSDL that would be part of the GSR for our service. In our example we would have

```
<wsdl:definitions ...>
      <serviceDataDescription name="queueLengths" element="n1:queue" ...>
      ...
</wsdl:definitions>
```

Furthermore, one could list the initial value of the queue as part of the port type in the service definition.

This is a very simple and elegant way to describe service metadata and state. By adopting this model, OGSI goes a long way towards the goal of building a truly interoperable component architecture for the Grid. The process of interacting with and understanding a Grid service is completely transparent. We feel this is exactly the right approach for OGSA.

4 THE PROBLEMS WITH NOTIFICATION

There are several areas where OGSI needs much more work and the working group is currently focusing their efforts on these problems. The current OGSI draft contains the $Notification$ port types. The basic idea is that a service may be a notification sink by implementing the $NotificationSink$ port type. This has one operation, $DeliverNotification$, which is used by a notification source to deliver an XML message to the sink. There is a simple form of a notification subscription that is implemented by a notification source. The $NotificationSource$ port type has one operation *subscribe* which is used by notification sinks to tell a source that it wishes to receive notifications about changes in the sources service data.

The intent of the notification mechanisms is to allow clients services to learn about the changes in another service's state as represented by changes in its service data elements. These changes of state would trigger a notification process which

would alert all interested "subscribers". However, there is a current discussion within the working group about that limited objective.

It has been suggested that it may be better to have a more general messaging model for OGSI. For example, the current model does not provide for reliable delivery. The $DeliverNotificaton$ operation does not even require an acknowledgement. One improvement would be to allow for a point-to-point queue-style mechanism in which publishers push messages to a queue and listeners pull messages, in order they were delivered, from the queue. This is one of the standard "reliable" models for message delivery that is used in many middleware systems such as the IBM MQ series. Another approach is something that resembles a topical publish/subscribe model. In this case a message middleware layer (or grid service) is defined to allow clients subscribe by "topic". When a publisher publishes a message on that topic, the message is pushed to all known subscribers. These are both standard models used in the Java Message Service [17] and are also very closely related to CORBA notification. One thing that is very important in a realistic messaging/event system is that it must use an adaptive combination of "push" and "pull", where "push" refers to subscribers that wait for notification via the $DeliverNotification$ operation and "pull" refers to situations where a notification source must periodically consult a third party (such as an MQ series queue) to ask if there are messages waiting for delivery.

Because notification and messaging is so important to distributed systems, it seems natural to extend the current OGSI service data notification to a more general system based on current practice. The first step that has been suggested[1] is to separate the concepts of notification from that of service data elements. We feel that any sort of XML message could be a message. But to do this one must add another operation to $NotificationSource$: a $GetMessage$ operation could be used to pull messages from the source. This would allow us to build point-to-point queue style message channels. The $DeliverNotification$ operation should return an ACK so that reliability can be improved.

5 CONCLUSION

We feel that the OGSA is the critical component to making Grids work and this effort is very exciting. As part of our original analysis we addressed a number of design decisions. In all cases, we agreed with the intent of the proposed features, even if we have technical differences with many small details. The first question we asked was, "Is this really an Open architecture?" "Open" can mean three different things. First, it may mean that OGSA is defined by an open process. This is now indeed the case. The Global Grid Forum OGSI working group has weekly teleconferences and frequent meetings that have refined the original specification into the form seen here. The leadership of S. Tuecke and D. Snelling is truly excellent. Second, "open" may mean that OGSA is extensible as a definition. This certainly seems to be the

[1] Many of these ideas were recently proposed by S. Graham of IBM.

intent. Third, it may mean that OGSA can be implemented without dependence on a particular code base. Again this seems to be true. The Globus group sees this as version 3.0 Globus but it is possible to implement this completely independently of Globus if one wished to do so. Furthermore, it is likely that services from different implementations may interoperate without problem but several technical problems must be solved before we can guarantee that this will be the case.

In our original analysis, our concerns had to do with the various WSDL extensions. While we completely agreed with the intent of these extensions, we feared that they may not have achieved their goal. In fact, we feared that they may have over-achieved their goals, because it may be possible to do more with less. This turned out to be the case. The current specification has a very simple and elegant architecture.

Another question we asked is who/what are the service clients? It is our belief that most Grid users will interact with the Grid through application or discipline specific portals. The portal servers will be responsible for managing the workflow that correctly sequences the accesses to the various Grid services. The OGSA model will provide an excellent framework to support the construction of these servers. The second type of "client" can be described as Grid service programmer. This is the person who either designs the portal servers or is charged with implementing new services or applications. Our experience with previous Grid systems, tells us that OGSA will greatly improve the quality of life for these individuals. However, there are several important technical issues to be considered here. Are the proposed Grid services at the right level to be usable by clients? How hard is it to define other services or to extend a class of service? Are the WSDL extensions sufficient to be usable? Are they too limiting? It is our opinion that OGSA addresses the correct level of services. The required *GridService* port is an excellent idea. We feel that this is a major contribution to defining a service component architecture.

There is one final, large issue that must be considered because many people will ask it. Is the web service model the correct one for building a Grid service architecture? Others will suggest a CORBA framework because it is a more mature technology. CORBA has always focused on source code compatibility rather than interoperability between different implementations. However, in attempting to do so, it leaves little flexibility in the way services are implemented. Because the web service model is primarily concerned about the specification of service properties such as interface and the specification of the port-to-protocol bindings, it allows great flexibility in the way the service's hosting environment is implemented.

As we have mentioned above, a different architectural style, recently proposed by Fielding [19], has received recent attention and is worth considering as a candidate for an Open Grid Architecture. It is called the Representational State Transfer (REST) and it is based on the principles which have helped the web achieve success. These principles include statelessness, low-entry barriers, and an emphasis on a small number of verbs applied to a large number of nouns. The verbs in the web are the operations, such as GET, defined by HTTP. The nouns are the rich network of URLs that comprise the web. The REST model assigns most of the semantics of

an operation to the data, rather than to the name of the operation. It is often contrasted to the remote procedure call (RPC) model, which defines a named set of operations, each with parameters. In this model, most of the semantics is defined by the name of operation. It is our opinion, OGSA is somewhere in between a purely RPC based framework like CORBA and REST. OGSA services are not stateless, but they do rely on a relatively small set of methods. For example, rather than having many different methods for retrieving various kinds of information about a service, OGSI relies on one method *getServiceData*, which can return a large repertoire of elements. This reliance on data results in more extensible and interoperable systems. Extensions to data are relatively easy to add without impeding interoperability with existing code, especially if a format like XML is used to represent the data.

As we have observed, the OGSI working group is actively addressing several of the other concerns we have. The most significant of these is the messaging architecture. We feel that if we want to build precise software component architecture out of OGSI, we need to adopt a standard messaging architecture that will serve the widest possible collection of applications.

REFERENCES

[1] Foster, I.—Kesselman, C.—Nick, J.—Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/research/papers/ogsa.pdf, January, 2002.

[2] Tuecke, S.—Czajkowski, K.—Foster, I.—Frey, J.—Graham, S.—Kesselman, C.: Grid Service Specification February, 2002.

[3] Johnston, W.—Gannon, D.—Nitzberg, B.—Woo, A.—Thigpen, B.—Tanner, L.: Computing and Data Grids for Science and Engineering. Proceedings of SC2000.

[4] Foster, I.—Kesselman, C.—Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, Vol. 15, 2001, No. 3.

[5] The Grid: Blueprint for a New Computing Infrastructure. Ian Foster and Carl Kesselman (Eds.), Morgan-Kaufman, 1998, see also: Argonne National Lab, Math and Computer Science Division, http://www.mcs.anl.gov/globus.

[6] Czajkowski, K.—Fitzgerald, S.—Foster, I.—Kesselman, C.: Grid Information Services for Distributed Resource Sharing. Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[7] Grimshaw, A.: Legion: A Worldwide Virtual Computer. See http://www.cs.virginia.edu/ legion.

[8] Liszkow, M.—Livny, M.—Mutka, M.: Condor — A Hunter of Idle Workstations. Proceedings ICDCS, pp. 104–111, San Jose, Ca., 1988.

[9] Particle Physics Data Grid. see http://www.ppdg.net/.

[10] NeesGrid Home Page. see http://www.neesgrid.org/.

[11] DOE Science Grid. see `http://www-itg.lbl.gov/Grid/`.

[12] The Grid Physics Network, `http://www.griphyn.org/`.

[13] The Global Grid Forum. `http://www.gridforum.org`.

[14] Web Services Description Language (WSDL) 1.2. W3C, `http://www.w3.org/TR/wsdl12`.

[15] UDDI: Universal Description, Discover and Integration of Business for the Web. see `http://www.uddi.org`.

[16] Web Services Inspection Language (WSIL). see `http://xml.coverpages.org/IBM-WS-Inspection-Overview.pdf`.

[17] Monson-Haefel, R.—Chappell, D.: Java Message Service. O'Reilley & Associates, December 2000.

[18] Curbera, F. et. al.: Business Process Execution Language for Web Services, Version 1.0. see `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`.

[19] Fielding, R. T.: Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. Dissertation, University of California, Irvine, 2000.

[20] Gannon, D.—Chiu, K.—Govindaraju, M.—Slominski, A.: An Analysis of The Open Grid Services Architecture. A Report to the UK E-Sciences Core Program, `http://www.extreme.indiana.edu/ gannon/OGSAAnalsyis3.pdf`.

Dennis Gannon is a professor in the Department of Computer Science at Indiana University and he is its current chair. His previous positions include the Department of Computer Science at Purdue University and the Center for Supercomputer Research and Development, University of Illinois. His current work is on the design of software component architectures for distributed scientific applications, and the study of the architecture of Grid systems. He is one of the co-founders of the Common Component Architecture project (now supported by the DOE Center for Component Technology for Terascale Simulation Software), the Java Grande Forum, and the Global Grid Forum where he co-chairs the Open Grid Service Architecture working group and the Grid Computing Environments research group. Gannon is also the Science Director for the new Indiana Pervasive Technologies Labs. He is also the Chief Computer Scientist for the NCSA Alliance. He received his Ph.D. degrees from the University of Illinois in Computer Science in 1980 and the University of California, Davis in Mathematics in 1975. His B.S. degree is from the Univerisity of California, Davis in 1969.

Kenneth Chiu is a postdoctoral research associate at Indiana University. He received his Ph.D. in Computer Science from Indiana University in 2001. The title of his thesis was "An Architecture for Concurrent, Peer-To-Peer Components". He received his A.B. from Princeton University in 1988. He is the leading architect for the Proteus Multiprotocol communication library.

Madhusudhan Govindaraju is a Postdoctoral Scientist in the Extreme Computing Lab at Indiana University. His interests are in the areas of Grid computing, distributed object systems, high performance RMI, web services, component based technologies and problem solving environments. He has been actively involved in the design and implementation of a system named XCAT that is based on the Common Component Architecture (CCA) specification. XCAT provides a services-based architecture for building large scale distributed applications on the Grid. He is also working on the design and development of high-performance interoperable communication infrastructure for heterogeneous environments. He received a Bachelor of Engineering (B.E.) in computer science from Birla Institute of Technology, Ranchi, India, in 1992. He completed his M.S. in computer science from Indiana University in 1996. In 2002 he received his Ph.D. in computer science from Indiana University. The title of his Ph.D. thesis was: "An open framework code generation toolkit for distributed systems based on XML schemas."

Aleksander Slominski is a Ph.D. student at Indiana University. He is currently working as a Research Assistant in the Extreme Computing Lab working on projects in distributed scientific computing. He is exploring the use of XML to enable and simplify the development of frameworks based on the common component architecture, especially in the context of Grid computing. He is the creator of the XSOAP toolkit (previously SoapRMI), and XML Pull Parser (XPP). He is also the co-creator of Common API for XML Pull Parsing (`http://www.xmlpull.org`). He holds an MS degree in computer science from Indiana University and received his Master in mathematics from Nicholas Copernicus University in Torun, Poland.