

An Analysis of The Open Grid Services Architecture

Dennis Gannon, Kenneth Chiu, Madhusudhan Govindaraju, Aleksander Slominski
Department of Computer Science
Indiana University, Bloomington, IN

Commissioned by the UK e-Science Core Program

1.0 Introduction and Executive Summary

This paper provides a brief technical review of the Open Grid Services Architecture (OGSA) that has been proposed in the paper, "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, Karl Czajkowski, Jeffrey Fry and Steve Graham.

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 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. A number of large Grid deployments are now 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 Services. 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 fetch the WSDL documents needed to access them. However, not all services will be listed in 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 WSFL [17], which is a mechanism for scripting the workflow for integrating multiple services together to accomplish a complex task. A workflow engine reads a WSFL instance document, and functions as an agent by contacting each involved service according to the order represented as a directed graph. Another is the Web Services Invocation Framework (WSIF), which provides 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.

1.2 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 anticipated the need for extensions to the core language and provided the requisite hooks to make that possible. The extensions used by the OGSA have designed include the concept of "service type", which allow us to describe families of services defined by a collections of ports of specified types. OGSA also provides 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 provide a mechanism to describe service semantic evolution and versioning. A basic WSDL instance document can only state that a service implements a port with the given interface. It cannot convey any information about what the service does with that port. The OGSA extensions allow us to name families of services that have identical semantics and to assert that a particular service implements these semantics. Clients of the service will have a clue as to what behavior to expect from the service.

OGSA specifies three things that a web service must have before it qualifies as a Grid Services. First it must be an instance of a service implementation of some service type as described above. Second, it must have a Grid Services Handle (GSH), which is a type of Grid URI for the service instance. The GSH is not a direct link to the service instance, but rather it is bound to a Grid Service Reference (GSR). The GSR might be (the OGSA allows for other representations) the WSDL document for the service instance with the required "instanceOf" and other OGSA extensions. 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.

The third property that elevates a Grid Service above a garden variety web service, is the fact that each Grid Service instance must implement a port called "GridService" which has three operations:

- **FindServiceData.** This allows a client to discover more information about the service's state, execution environment and additional 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 by the client as a standard way to learn more about the service.
- **Destroy.** This allows an authorized client to kill the instance.
- **SetTerminationTime.** A method to extend the lifetime of a service.

In general, the required GridService port is an excellent idea. We have long used something similar for our work 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 and NotificationSink ports. The intent of this service is to provide a simple publish-subscribe system similar to Java's JMS, but based on XML messages. It is noted that messaging may be implemented on top of many different existing research and commercial systems. This provides important leveraging of existing technology.

The OGSA standard Grid Service ports include

- **HandleMap.** This is a service that provides the mapping between the Grid Service Handle 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.
- **Registry.** This is a service that allows service instance metadata to be bound to a registry. The Registry port also allows services to be unregistered. There seems to be only one suggested way to extract information from the registry service and this is by using its FindServiceData method on the GridService port.
- **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 currently use and we have found the concept very valuable.

1.4 Summary of Conclusions

We feel that the OGSA is the critical component to making Grids work and this effort is very exciting. As part of our analysis we have addressed a number of design decisions. In all cases, we agree with the intent of the proposed features, even if we have technical differences with many small details.

The first question that has often been asked, "Is this really an Open architecture?" "Open" can mean three different things. First, it may mean that OGSA is defined by an open process. To date, this has not been true, but a Grid Forum working group has now been established, so the process

is there. Second, it may mean that OGSA is extensible as a definition. This certainly seems to be the intent. However, the process by which extensions can be discovered or made it still unclear. 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 Globus 3.0 but it is possible to implement this completely independent 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.

Complete answers to "openness" questions will be known as soon as some of the Global Grid Forum working groups attempt to define additional standard interfaces. We feel that only a few technical details in the design may inhibit this. In fact, the adoption of this framework should greatly accelerate progress. Our specific concerns are with the WSDL extensions. While we completely agree with the intent of these extensions, we fear that they may not have achieved their goal. In fact, they may have overachieved them because it may be possible to do more with less. Our detailed comments on this topic are in the following section.

Another important question to ask 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 correct 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. While they have not yet fully specified the schema for the Grid Service Data elements that will be returned by this port, we can already see that this is a major contribution. The other areas where service port types have been introduced, registry, notification and factories are also very good. In each case, we have some technical quibbles with the details, but these are exactly the right core features of usable Grid services 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.

A different architectural style, recently proposed by Fielding [18], 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 that 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.

In 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, OGSA relies on one method, 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. Adding a parameter to a method, on the other hand generally requires changes to existing code.

There are other technical problems with the Web Services model that may make it less suitable to Grid computing, and we will discuss those in the sections ahead. However, our conclusion is that OGSA is on the right track. If they allow the specification to evolve and change, discarding ideas that fail and always allowing room for experimentation, it should be successful.

2.0 An Analysis of the details -Assessment and Recommendations

In this section we analyze OGSA features we consider to be critical to its success, but where we see technical problems. In some cases, we suggest changes that we believe will improve OGSA. The reader must keep in mind that we have focused on the negative because that is where the technical problems lie. Unfortunately, this may give the incorrect impression that we have an overall negative view of OGSA. In fact, as we have stated in section 1.4 we are very excited by what OGSA represents. That is why we have gone to the trouble to prepare this analysis.

2.1 WSDL Extensions

WSDL extensions are a prominent part of OGSA. Is this a good idea?

Although WSDL is an extensible language, we believe that extensions need to be approached very carefully to preserve interoperability with existing WSDL clients and services. Preserving interoperability with non-OGSA WSDL clients will speed adoption. We note that even though WSDL provides the **required** attribute to denote whether or not a client is required to understand an extensibility element, this specification does not mention it. Unfortunately, some toolkits do not properly ignore non-required extensibility elements that are not understood. We do not believe this warrants abandoning extensibility elements, especially for an emerging technology such as Grid computing. However, we wish OGSA provided a standard mechanism to obtain a WSDL description devoid of OGSA extensibility elements.

We agree that use of WSDL extensions to specify additional semantics for OGSA services is good if done with great care. However, it must be clearly stated in the specification which WSDL extensions are required and which are not (and, in the required cases, we must use the `wsdl:required='true'` attribute). Unfortunately, the correct value of the required attribute is also unclear. If we define it to be true, then non-OGSA implementations will be unable to process the OGSA extensions. If we define it to be false (the default), does this then mean that an OGSA compliant client is free to ignore the extensions?

One possible solution would be for OGSA to add an attribute named 'mustRecognize' to extensibility elements. The value of this identifies an OGSA version. If the properties of an OGSA service require that it's OGSA clients recognize elements belonging to a specific service OGSA version, then those clients must recognize the element of the form.

```

<gsdl:someOGSAExtension mustRecognize="qname">
    ...
</gsdl:someOGSAExtension>

```

where “qname” is the fully qualified name of that version. Any OGSA client that does not recognize this service version should not attempt to use this service. Non-OGSA clients can safely ignore the extension.

The OGSA extensions provide a means for adding certain semantics. In any WSDL environment these semantics must be assumed or obtained anyway, but through non-OGSA means. For example, if the WSDL does not tell a client that the supplied port type is compatible with the desired port type, the client must either simply assume that it is, or obtain the information through other means. If you remove the extensibility elements, and you need those semantics, then you will have to use other means to obtain those semantics. Fortunately, there is another way to define the needed semantic, and we will return to this point later.

Which OGSA extensions are necessary? Which are good ideas? Which are almost right?

We will now assess each extension, as described in the Table 1 of the draft specification.

Interface Naming

This is simply a requirement that portTypes are immutable. We agree with this requirement. This is arguably a restriction rather than an extension, so should not have any negative affect on interoperability.

Compatibility Assertion

WSDL extensions for compatibility assertions are defined as "a mechanism to associate equivalent interface elements and implementations " in Table 1 from Section 4.1. Although we agree with the intent, we believe that an alternative mechanism such as RDF, which has more complete metadata framework including support for queries, should be considered. We therefore believe that they should be optional (wsdl:required should be never set to 'true' on them).

If they are to be kept as is, however, we recommend a few changes to improve their usability. First, we note that, contrary to their assertion, compatibility as defined is provably transitive. However, as a relation, it is clearly not symmetric, but this is o.k. In fact, ServiceImplementation compatibility, which requires full semantic and signature compatibility, is the only OGSA mechanism to state that one service “extends” the capabilities of another. To make compatibility more broadly useful, we recommend that port type compatibility be redefined to include both semantic and signature compatibility. We also recommend that the compatibility assertions include a mapping from old operation name to new operation name. This can be done by extending the 'type' attribute of the compatibility assertion down to the WSDL operation element as follows.

```

<gsdl:compatibilityAssertion name="example">
  <gsdl:compatible name="ns1:old_op1" withName="ns2:new_op1"
    type="operation">
</gsdl:compatibilityAssertion>

```

Without this, two port types may be compatible because there is a compatibilityAssertion, but there is no way to map the operation names from the old to the new. The type attribute in

compatible element should be QName instead of nmtoken to allow one to specify relation types in future without worrying about conflicts (the namespace from QName will protect against conflicts) and will make it possible to add new types of compatibility assertions by other OGSA users without fear of collision in name space.

ServiceData

One of the things that is missing in Web Services is discovery mechanism similar to introspection in programming languages. Such mechanisms will allow one to examine web service to discover their properties (also called service metadata). Using agreed upon standard names for metadata elements one can discover for example: service type(s), if it is a service instance, location of service factory, its compatibility with other services implementations etc. This allows an OGSA or even non-OGSA Web Service client to discover more information about the service's state, execution environment and additional semantic details that may not be available in the GSR and the client may prefer to do introspection instead of parsing 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.

OGSA authors propose serviceData elements and an operation FindServiceData to provide introspection in OGSA Web Services. In general, this seems like a good way to discover information about a running OGSA web service. Unfortunately, in the current specification, Service metadata is not very well specified. Some of it corresponds directly to serviceData elements, while other items are buried in the service description (defined at top page 14 in spec). These items can still be obtained indirectly from the service data, since it is possible to access the service description as a serviceData element, but this inconsistency is confusing and not necessary.

In general, keeping all metadata information inside serviceData elements will allow for easier discovery of service metadata than using WSDL extensibility elements to represent service description. However our main concern is the service data naming. The names are currently not namespace qualified, which means that some naming policy is necessary to prevent name collisions between different metadata efforts.

We feel that all of the OGSA proposed extensions should be changed to be serviceData elements and encapsulated inside serviceData elements unless they are critical extensions (and require wsd:required='true'). The extensions are still part of WSDL document but are contained in WSDL serviceData extensibility elements and this way they can be easily discovered by introspection on service instance.). We will explore this in much greater detail in the next subsection. We also recommend that a serviceData name be a QName (instead of nmtoken) - this will allow global names (XML namespaces) for service data elements.

ServiceType, ServiceImplementation and Service Instance.

The OGSA architecture currently has 'service implementation', 'service type' and 'service instance' concepts, which are represented grammatically with a somewhat confusing mixture of WSDL extensibility elements. We further note that although these concepts are defined in the documents, they are never made use of in the rest of the specification, which deals only in traditional port types. Surely, Registry must qualify as a service type and not just a port type?

Though, in principle, we agree with the ideas that motivate these concepts, we believe that a simpler yet more flexible ontology and architecture will result if we reformulate the ideas in a different way. Hence, we recommend dropping all three extensions in favor of the following approach that better reflects current web services practice.

Consider grouping the service type, service implementation, service description, and other service interface concepts into one service interface. This service interface is represented by what is termed an abstract *interface WSDL* document, which is simply a WSDL document with no service element (and perhaps no bindings). The service type is defined implicitly by the list of port types in the interface WSDL document, which eliminates the need for the serviceType element. The name of the service type is given by the target namespace of the WSDL.

A service instance is then essentially an interface WSDL document combined with a **service** element, which is termed an *instance WSDL* document. The WSDL spec already provides for this kind of aggregation with the *import* element, and any number of instance WSDL documents can import the same interface WSDL document. Using this approach, we can also make it possible for one service to implement multiple service Types specified by interface WSDLs that cannot be achieved by the current OGSA serviceType.

If we follow this approach, we can move `gsdl:serviceType` WSDL extensibility element into WSDL `serviceData` extensibility element, so it becomes part of service instance metadata. This element will simply contain a string with namespace of WSDL interface file. A service *instance* is then essentially a WSDL document that combines a number of interface WSDL documents (one or more) with `serviceType serviceData` element. The WSDL spec already provides for this kind of aggregation with the *import* element, and any number of instance WSDL documents can import the same interface WSDL document. (We note that the same solution also works for the compatibility assertions and `serviceImplementation`. We can also remove `gsdl:instanceOf` because the service instance handle is already part of service metadata.) We will illustrate this with a simple example. Suppose we have a service with two service interface documents defined in ‘someServiceType.wsdl’ and ‘someOtherServiceType.wsdl’. We can then create an instance document as follows.

```
<wsdl:definitions name="ServiceTypeUseExample"
  targetNamespace="http://gridExample/ServiceTypeUseExample.wsdl"
  xmlns:tns="http://gridExample/ServiceTypeUseExample.wsdl"
  xmlns:ns1="http://gridExample/someServiceNamespace"
  xmlns:ns2="http://gridExample/someOtherServiceNamespace"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  >

  <wsdl:import namespace="http://gridExample/someServiceNamespace"
    location="http://gridExample/someServiceType.wsdl"/>

  <wsdl:import namespace="http://gridExample/someOtherServienamespace"
    location="http://gridExample/someOtherNamespaceServiceType.wsdl"/>

  <wsdl:service name="StevesFirstService">

    <gsdl:serviceData name="gsdl:gridServiceHandle">
      http://somehost:2322/gridservice/examples/instance-434334
    </gsdl:serviceData>

    <gsdl:serviceData name="gsdl:gridServiceFactoryHandle">
      http://someotherhost:2322/gridservice/examples/factory-38787
    </gsdl:serviceData>
```

```

...

<gSDL:serviceData name="gSDL:serviceType">
  http://gridExample/someServiceNamespace
</gSDL:serviceData>

<gSDL:serviceData name="gSDL:serviceType">
  http://gridExample/someOtherServiceNamespace
</gSDL:serviceData>

<gSDL:serviceData name="gSDL:serviceImplementation">
  http://gridExample/ServiceExample/2/0
</gSDL:serviceData>

<gSDL:serviceData name="gSDL:compatibilityRef">
  http://gridExample/myCompatibilities.wsdl
</gSDL:serviceData>

...

</wsdl:service>
</wsdl:definitions>

```

Note that this separation of the WSDL document into an interface part and an instance part is a logical separation that may or may not be reflected in concrete structures. Most GSRs will probably not include the interface WSDL, because it will have already been accessed in a pre-processing phase to generate a stub class. Upon receipt of the GSR, the client will create a stub object of the class corresponding to the service type, as indicated by the namespace attribute of the import element.

If a dynamic invocation framework of some kind is used, then the client may need to retrieve the interface WSDL from the network. This situation may be undesirable if the connectivity to the interface WSDL location is poor, so we also propose an XML-encoding of GSRs to include all required WSDL parts as new WSDL extensibility elements that can be ignored (`wsdl:required='false'`).

```

<wsdl:definitions ...>
  <gSDL:container contains="URI1">
    <!-- WSDL doc for serviceType 1 identified by namespace URI1-->
  </gSDL:container>
  <gSDL:container contains="URI2">
    <!-- WSDL doc for serviceType 2 identified by namespace URI2-->
  </gSDL:container>
  <wsdl:message ...>
  </wsdl:message>
  <!-- other WSDL elements -->
</wsdl:definition>

```

The WSDL import element refers to a WSDL document through a URI. The "contains" attribute gives the URI to which the enclosed WSDL corresponds. This allows the complete WSDL description to be constructed without requiring the retrieval of the separate WSDL pieces, and without modifying the original separate WSDL documents.

Obviously we have not worked through all the semantic implications of shifting from WSDL extensions to something specified purely with metadata expressed in serviceData elements, but we do feel that this provides a simpler and more usable approach.

The Grid Service Handle (GSH)

A service instance does not necessarily correspond to an operating system entity, but is rather defined by its behavior. That is, any manifestation (process, etc.) of a service is the same instance as long as it *acts* like its the same instance, as seen through its interfaces. From [1], Sec. 6.1 (emphasis original):

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.

Also, on page 19 of [1], it states, "If a Grid service fails and is restarted in such a way as to preserve its state, then it is essentially the same instance, and the same GSH can be used."

So the scenario for upgrading an instance would be something like a registry or factory that keeps shareable state on disk. To replace an old registry implementation with a new registry implementation, first start the new registry process, and have it take over the GSH by directing all new mappings to its GSR. Because the old and the new share state on disk, the old registry process can keep fulfilling requests made on its GSR. Eventually, all the old GSRs expire, and the old registry process can be shutdown. From the clients viewpoint, there was ever only one instance.

A GSH for any Grid Service is required to be a URL that contains the GSHHomeHandleMapID. This GSHHomeHandleMapID is globally unique for mapping the GSH to one or more valid Grid Service References (GSRs). The GSHHomeHandlerMapID includes a hostname at which the HandleMap service resides. The HandleMap provides a mapping from a GSH to a GSR. This may present several problems

- Since the location of the HandleMapper is included in the GSH, it is not possible to move the HandleMap service to another host. For the lifetime of the GSR, the Mapper service has to reside on a fixed machine. However, it is reasonable to expect that over a period of time there will be a need to move the HandleMap service to a different machine or administrative domain. In such cases the binding from a GSH to a GSR may be lost.
- In the absence of a well-defined security policy, it is not possible for distrustful organizations to communicate and obtain the mapping from a GSH to a GSR.

There is one solution that should be considered. The Secure Grid Naming Protocol (SGNP) has been proposed to the Grid Forum to alleviate the location-dependency and security problems that arise with GSH and HandleMap service of OGSA. The SGNP model can be considered as an alternative way of defining the specification for the HandleMapper PortType in OGSA. SGNP provides a mechanism for "Naming" of Grid resources. It defines a scheme that assigns "logical" and thus location-independent names to Grid resources. The scheme obviates the need for authentication of two Grid resources via a trusted third party. The logical name is a combination of the identity and security information of a Grid resource. The security information can be (1) nothing (2) RSA public Key (3) X.509 certificates (4) Open PGP certificate. An SGNP name is a Location Independent Object Identifier (LOID), which is globally unique and immutable for the lifetime of the Grid resource. The actual location of a Grid resource can be determined by associating the LOID with one or more communication protocols and network endpoints. This two level naming scheme of SGNP (LOID and its mapping) allows a Grid resource to be

migrated both spatially and temporally. The binding of a LOID can be represented as a WSDL document. SGNP defines a Grid Naming Service (GNS), with a well-known binding that provides clients access to SGNP naming services. This naming service provides access to authoritative LOID-to-Binding mappings and a Resolver Hierarchy Service that resolves increasingly specific portions of LOIDS. We feel this is an idea of considerable merit and it is worth evaluating.

2.2 The Proposed Service Ports.

GridService

We completely agree that the GridService portType should be a required element in each Grid Service. However, the GridService portType defines both discovery and lifecycle management - those are separate services and combining them in one port type hinders experimentation with creation of independent enhancements and extensions for those very important capabilities. Many Grid research groups are already working on those issues and may want to extend existing features or add new ones - keeping those two capabilities separate will make it easier. We recommend that the GridService portType be broken into two portTypes

- GridMetadataDiscovery containing just the FindServiceData operation, and
- GridLifecycleManagement with the SetTermination and the Destroy operation

This allows for separation of concerns that may be very important. For example GridLifecycleManagement can be bound on SSL endpoint and GridMetadataDiscovery on a freely available HTTP location.

HandleMap

The handleMap port does not allow mappings to be deleted or added. However, this is deliberate: 'Upon creation by a factory, the Grid service instance MUST be registered with, and receive a GSH from, a home handleMap service (see Section 7). The method by which this registration is accomplished is specific to the hosting environment, and is therefore outside the scope of this specification. The Grid service instance additionally MAY be registered with other handleMaps.' Though the OGSA believes that handle map registration is outside the scope of the OGSA, we feel that adding new HandleMapManagement portType that will contains *add*, *delete*, *change* or a more sophisticated handle that uses metadata. This serviceType should be optional and may require authorization to use. We recommend adding it as a separate serviceType with its own portType to allow setup in which access to HandleMap::FindByHandle is unlimited but access to HandleMapManagement operations requires authorization. That is because one may want to let most people map GSH to GSR, but a small set of people add new mappings.

Notification

We feel that notification is an extremely important service that OGSA must support. However, we have several technical quibbles with what is proposed here.

.NotificationSourceTopicNames should return list of QNames and not nmtokens. The sink in NotificationSource::SubscribeToNotificationTopic assumes that GSH can be always mapped to a GSR, which may not always be the case. We feel that returning actual GSR would be better. By examining GSR, the client can find out how long the service instance is valid and also if a GSH mapping is supported.

The specification also requires a notification sink to expose a network accessible endpoint when calling `SubscribeToNotificationTopic` because the notification sink must have a GSR that identifies network accessible location to deliver notifications. That will not work for services or clients that are behind a firewall - not only are their endpoints inaccessible, the GSH to GSR mapping service may also be inaccessible. This is related to bigger problem of participation in Grid Web Services by small devices that may not have permanent IP address and for OGSA clients that reside behind firewalls or NAT and need to access notification services. We suggest augmenting the notification source interface to allow a client to subscribe for information pulling. In this way, clients behind firewall can request and pull notification data.

Using `GridPort::FindServiceData` to execute notification queries for large and changing quantities of data does not seem like a good idea. Events do not necessarily describe metadata and they are transient. Even though service data may be used to represent dynamically changing state information, this is still inherently different from an event. An event represents a *transition* from one state to another, as opposed to the state itself. This fundamental difference makes it difficult to unify a metadata change model with generic event notification. For example, a service may choose to make available notification about its memory utilization since start of computation. This should be reported for every second of the computation. Memory utilization data for every second that the service is run, is not really part of service metadata but this capability to do this kind of notifications can be represented as `serviceData` that describes how to register for notifications.

Factory and PrimaryKey

Introducing a Factory model into OGSA is also a very good idea. However, we fail to see the purpose of Grid Service Primary Key (GSPK). What is the added functionality when compared to GSH and GSR? As defined in OGSA factory service input, the primary key is required to be globally unique? How will multiple clients make sure they are providing unique keys? It is possible that we have failed to catch an important point here.

Registry

The requirement "Update of an existing handle MUST only be allowed by a client that authenticates with the same subject as the client that registered this handle previously." in section 11.2.2.1 seems to be very stringent and. We recommend that authentication and authorization be required but how it is implemented should be left to the implementation (it may indeed present the same credentials for renewal but there is no inherent reason why service clients should not be allowed to use different credentials, if accepted by a Registry). There seems to be only one suggested way to extract information from the registry service and this is by using its `FindServiceData` method on the `GridService` port. While there is nothing inherently wrong with this, it seems a misuse of a feature that should be best used to discover metadata and not service state.

3.0 Final Comments.

There were three issues that occupied much of our discussion while we evaluated the OGSA design. They are

Interoperability. An ideal specification proscribes a set of rules and promises a set of guarantees. If an implementation conforms to the rules, then the guarantees hold. Within this context, interoperability can be loosely defined as the preservation of the guarantees between clients and servers from possibly different implementations.

The presence of a rule set and a guarantee set suggests two corresponding dimensions to interoperability. In one dimension, interoperability is simply an exercise in specification. Testing reveals hidden assumptions, which are then eliminated by revising the set of rules. Considering the guarantee set, however, reveals another dimension. For any requirement outside the guarantee set, each implementation will choose its own solution. Depending on the overall goals, the result might be a specification that achieves interoperability in theory but not in practice.

For example, suppose the DOE wishes to increase collaboration between laboratories by mandating that all projects use WSDL with SOAP bindings. Each laboratory, however, implements its own service for transmitting notification events within a distributed application. The WSDL documents detail how to invoke the other implementation's operations, but say nothing about the semantics of the operations. Aspects such as the names of the operations, or the delivery model (push vs. pull) differ. If notification between implementations is critical to collaboration, the higher goal will not be achieved, even though WSDL performed as specified.

Thus, the second dimension is really a usability issue. If OGSA does not specify functionality to be usable by most grid systems, each implementation may use different mechanisms to obtain the missing functionality. This will of course impede interoperability. On other hand, specifying too much will hinder research and innovation, because anything different will violate the specification. Finding the right balance is critical to the success of OGSA.

We believe OGSA should be a lightweight specification and should not adapt the all-in-one approach of CORBA. Instead the spec should be broken in number of parts. OGSA Core should define elements required to create OGSA compatible services. To achieve good interoperability, the minimal number of properties should be specified, but it must be a sufficient number that behavior guarantees are clear and well defined. Finally, number of standard OGSA services such as Factory should be defined with the intention that they provide basic building blocks for new services. This is a very difficult task.

Extensibility. Grid computing and web services are still developing rapidly. An extensible architecture provides a leverageable base for new ideas, thus allowing them to be tested more quickly. Furthermore, once an idea gains wide acceptance, an extensible architecture can incorporate it as part of the standard. Extensibility also requires standard mechanisms to discover what extensions have been implemented. Without this, each group will implement their own means, which will result in multiple mechanisms to discover extensions. The resulting duplication and complexity would hinder the testing and subsequent standardization of good extensions.

Though OGSA was designed with extensibility in mind, we feel that current mechanisms are inadequate. No community process exists for adding extensions in an orderly manner. They also do not provide mechanisms for the robust discovery of new extensions. Since OGSA is currently incomplete, and Grid computing is an active field of research, we feel that this is an important issue.

Usability. Usability is closely related to the second dimension of interoperability, but we list it separately because it considers a broad range of issues. For example, an otherwise good specification may never gain wide acceptance if it does not provide security. We are certain that this will be a major issue of debate; hence we have not gone into great detail on this point.

There is one final issue that concerns the suitability of Web Services as the mechanism to describe Grid Services. The WSDL presents the service as the sole entity possessing identity. Ports are not separately addressable, but merely a different interface to the service. Thus, multiple ports with different names but the same port type are defined to possess "semantically equivalent

behavior." This serves most business transactions well. If for example, you need to buy a book through a web service, the notion of giving ports identity is meaningless.

In scientific computing, however, we may wish to provide access to actual instruments through web services, and perhaps through multiple protocols, in which case the phrase "semantically equivalent" is ambiguous. Consider controlling an electron microscope possessing two electron guns through a web service. The two guns have identical physical characteristics, so they naturally map to two ports with the same port type. Also suppose that we provide access to the service via two different protocols. The OGSA document for this service will now have four ports with the same port type. No distinction can be made, however, between two ports that control the same electron gun via different protocols, or two ports that control different electron guns. We feel that this problem can be solved but we defer that discussion to another paper.

References

- [1] The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke; January, 2002.
- [2] Grid Service Specification. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman; February, 2002.
- [3] W. Johnston, D. Gannon, B. Nitzberg, A. Woo, B. Thigpen, L. Tanner, "Computing and Data Grids for Science and Engineering," Proceedings of SC2000.
- [4] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations." *International J. Supercomputer Applications*, 15(3), 2001.
- [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] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "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] Andrew Grimshaw. "Legion: A Worldwide Virtual Computer." See <http://www.cs.virginia.edu/~legion>.
- [8] M. Liskow, M Livny and M. Mutka. "Condor – A Hunter of Idle Workstations, Proceedings ICDCS, pages 104-111, San Jose, Ca. 1988. IEEE.
- [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) 1.1", W3C,
<http://www.w3.org/TR/wsdl>

[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] "Web Services Flow Language (WSFL)", see <http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>.

[18] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D. Dissertation, University of California, Irvine, 2000.