

Grid Service Specification

Steven Tuecke¹ Karl Czajkowski³ Ian Foster^{1,2}
Jeffrey Frey⁴ Steve Graham⁵ Carl Kesselman³

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

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

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

⁴ IBM Corporation, Poughkeepsie, NY 12601

⁵ IBM Corporation, Research Triangle Park, NC 27713

Abstract

Building on both Grid and Web services technologies, the Open Grid Services Architecture (OGSA) defines mechanisms for creating, managing, and exchanging information among entities called *Grid services*. Succinctly, a Grid service is a Web service that conforms to a set of conventions (interfaces and behaviors) that define how a client interacts with a Grid service. These conventions, and other OGSA mechanisms associated with Grid service creation and discovery, provide for the controlled, fault resilient, and secure management of the distributed and often long-lived state that is commonly required in advanced distributed applications. In a separate document, we have presented in detail the motivation, requirements, structure, and applications that underlie OGSA. Here we focus on technical details, providing a full specification of the behaviors and Web Service Definition Language (WSDL) interfaces that define a Grid service.

This is a DRAFT document and continues to be revised.
The latest version can be found at <http://www.gridforum.org/ogsi-wg>.
Please send comments to the authors (see Section 14 for contact information).

Table Of Contents

1	Introduction	4
2	Terminology and Abbreviations.....	4
3	Setting the Context.....	5
3.1	Relationship to Distributed Object Systems.....	5
3.2	Client-Side Programming Patterns.....	6
3.3	Relationship to Hosting Environment.....	7
4	The Grid Service	8
4.1	WSDL Extensions and Conventions.....	8
4.2	Service Description and Service Instance.....	9
4.3	Service Data	10
4.3.1	XML Element Lifetime Declaration Attributes	11
4.3.2	serviceDataDescription	13
4.4	Defining Service Types: ServiceType and ServiceImplementation	14
4.4.1	ServiceType.....	14
4.4.2	ServiceImplementation.....	15
4.5	Interface Naming and Change Management.....	15
4.5.1	The Change Management Problem.....	15
4.5.2	Naming Conventions for Grid Service Descriptions.....	16
4.5.3	Compatibility Assertions.....	16
4.5.3.1	PortType Compatibility.....	17
4.5.3.2	ServiceType Compatibility	17
4.5.3.3	ServiceImplementation Compatibility	18
4.5.3.4	CompatibilityAssertion Mutability	18
4.6	Naming Grid Service Instances: Handles and References	18
4.6.1	Grid Service Handle (GSH)	19
4.6.2	Grid Service Reference (GSR).....	20
4.6.2.1	WSDL Encoding of a GSR	21
4.7	Grid Service WSDL Examples	22
4.8	Grid Service Lifecycle	24
4.9	Common Handling of Operation Faults.....	25
4.10	Summary of PortTypes Defined in this Document	25
5	The GridService PortType	25
5.1	GridService PortType: ServiceData Elements	26
5.2	GridService PortType: Operations and Messages.....	27
5.2.1	GridService :: FindServiceData	27
5.2.2	queryByServiceDataName.....	28
5.2.3	GridService :: SetTerminationTime	28
5.2.4	GridService :: Destroy.....	29
6	The HandleMap PortType.....	29
6.1	HandleMap PortType: ServiceData Elements.....	29
6.2	HandleMap PortType: Operations and Messages	29
6.2.1	HandleMap :: FindByHandle	29
7	Notification.....	30
7.1	The NotificationSource PortType	30

7.1.1	Notification Topics.....	31
7.1.2	NotificationSource PortType: ServiceData Elements.....	31
7.1.3	NotificationSource PortType: Operations and Messages	32
7.1.3.1	NotificationSource :: SubscribeToNotificationTopic	32
7.1.3.2	NotificationSource :: UnsubscribeFromNotificationTopic.....	33
7.2	The NotificationSink PortType.....	33
7.2.1	NotificationSink PortType: ServiceData Elements.....	33
7.2.2	NotificationSink PortType: Operations and Messages	33
7.2.2.1	NotificationSink :: DeliverNotification.....	33
7.3	Integration With Notification Intermediaries.....	33
8	The Factory PortType.....	34
8.1	Factory PortType: ServiceData Elements	35
8.2	Factory PortType: Operations and Messages.....	35
8.2.1	Factory :: CreateService.....	35
9	Registry	36
9.1	WS-Inspection Document.....	36
9.2	The Registry portType	37
9.2.1	Registry PortType: ServiceData Elements.....	37
9.2.2	Registry PortType: Operations and Messages	37
9.2.2.1	Registry :: RegisterService.....	37
9.2.2.2	Registry :: UnregisterService	38
9.3	Existence and Change Notification	38
10	Other Properties of OGSA To Be Addressed	38
11	Change Log	39
11.1	Draft 1 (2/15/2002) → Draft 2 (6/13/2002)	39
12	Acknowledgements	40
13	References	40
14	Contact Information	40
15	XML and WSDL Specifications	41

1 Introduction

The *Open Grid Services Architecture* (OGSA) [4] integrates key Grid technologies [3, 5] (including the Globus Toolkit [2]) with Web services mechanisms [6] to create a distributed system framework based around the Grid service. A *Grid service instance* is a (potentially transient) service that conforms to a set of conventions (expressed as WSDL interfaces, extensions, and behaviors) for such purposes as lifetime management, discovery of characteristics, notification, and so forth. Grid services provide for the controlled management of the distributed and often long-lived state that is commonly required in sophisticated distributed applications. OGSA also introduces standard factory and registry interfaces for creating and discovering Grid services.

In this document, we propose detailed specifications for the conventions that govern how clients create, discover, and interact with a Grid service. That is, we specify (a) how Grid service instances are named and referenced, (b) the interfaces (and associated behaviors) that define any Grid service and (c) the additional (optional) interfaces and behaviors associated with factories and registries. We do *not* address how Grid services are created, managed, and destroyed within any particular hosting environment. Thus, services that conform to this specification are not necessarily portable to various hosting environments, but they can be invoked by any client that conforms to this specification

Our presentation here is deliberately terse, in order to avoid overlap with [4]. The reader is referred to [4] for discussion of motivation, requirements, architecture, relationship to Grid and Web services technologies, other related work, and applications.

This document is a work in progress and feedback is encouraged. Future versions will incorporate additional pedagogical text and examples. We also draw the reader's attention to various "Notes" that indicates areas of particular uncertainty.

2 Terminology and Abbreviations

The term *hosting environment* is used in this document to denote the server in which one or more Grid service implementations run. Such servers are typically language and/or platform specific. Examples include native Unix and Windows processes, J2EE application servers, and Microsoft .NET.

- The terms Web services, XML, SOAP, WSDL, and WS-Inspection are as defined in [4].

We also discuss several extensions to Web services, such as **serviceType**, **serviceData**, **compatibilityAssertion**, in WSDL, etc. These extensions are summarized in Section 4. In this document we use **bold** font face to emphasize WSDL defined elements and names of WSDL extensions.

The following abbreviations are used—and the corresponding terms defined—in this document:

- *GSH*: Grid Service Handle.
- *GSR*: Grid Service Reference.
- *SDE*: Service Data Element.

The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" are to be interpreted as described in RFC-2119 [cite: RFC 2119].

3 Setting the Context

Although [4] describes overall motivation for the Open Grid Services Architecture, this document describes the architecture at a more detailed level. Correspondingly, there are several details we examine in this section that help put the remainder of the document in context. Specifically, we discuss the relationship between OGSA and distributed object systems, and also the relationship that we expect to exist between OGSA and the existing Web services framework, examining both the client-side programming patterns and a conceptual hosting environment for Grid services.

We emphasize that the patterns described in this section are enabled but not *required* by OGSA. We discuss these patterns in this section to help put into context certain details described in the other parts of this document.

3.1 Relationship to Distributed Object Systems

As we describe in much more detail below, a given Grid service implementation is an addressable, and potentially stateful, instance that implements one or more interfaces described by WSDL **portTypes** within the context of an aggregate **serviceType** (see Section 4.2). Grid service factories (Section 8) can be used to create instances of a given **serviceType**. Each Grid service instance has a unique identity with respect to the other instances in the system. Each instance can be characterized as state coupled with behavior published through type-specific operations. The architecture also supports introspection in that a client application can ask a Grid service instance to return information describing itself, such as its **serviceType** and the collection of WSDL **portTypes** that it implements.

Grid service instances are made accessible to (potentially remote) client applications through the use of a Grid Service Handle (Section 4.6.1) and a Grid Service Reference (Section 4.6.2). These constructs are basically network-wide pointers to specific Grid service instances hosted in (potentially remote) execution environments. A client application can use a Grid Service Reference to send requests (represented by the operations defined in the WSDL **portType**(s) of the target service) directly to the specific instance at the specified network-attached service endpoint identified by the Grid Service Reference.

Each of the characteristics introduced above (stateful instances, typed interfaces, unique global names, etc.) is frequently also cited as a fundamental characteristic of so-called *distributed object-based systems*. However, there are also various other aspects of distributed object models (as traditionally defined) that are specifically *not* required or prescribed by the Grid service architecture. For this reason, we do not adopt the term distributed object model or distributed object system when describing this work, but instead use the term Open Grid Services Architecture, thus emphasizing the connections that we establish with both Web services and Grid technologies.

Among the object-related issues that are not addressed within OGSA are inheritance, development approach, and hosting technology. The Grid service specification does not require, nor does it prevent, implementations based upon object technologies that support inheritance at either the interface or the implementation level. There is no requirement in the architecture to expose the notion of inheritance either at the client side or the service provider side of the usage contract. In addition, the Grid service specification does not prescribe, dictate, or prevent the use of any particular development approach or hosting technology for the Grid service. Grid service providers are free to implement the semantic contract of the service in any technology and hosting architecture of their choosing. We envision implementations in J2EE, .NET, traditional commercial transaction management servers, traditional procedural UNIX servers, etc. We also envision service implementations in a wide variety of programming languages that would include both object-oriented and non-object-oriented alternatives.

3.2 Client-Side Programming Patterns

Another important issue that we feel requires some explanation, particularly for readers not familiar with Web services, is how OGSA interfaces are likely to be invoked from client applications. OGSA incorporates an important component of the Web services framework: the use of WSDL to describe multiple protocol bindings, encoding styles, messaging styles (RPC vs. document-oriented), and so on, for a given Web service.

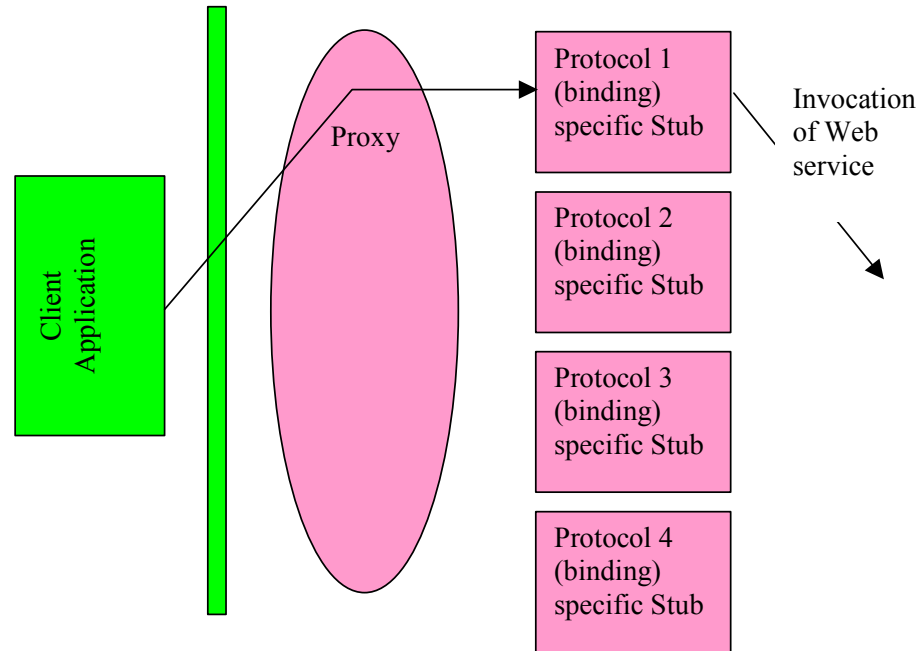


Figure 1: A possible client-side runtime architecture

Figure 1 depicts a possible (but not required) client-side architecture for OGSA. In this approach, there is a clear separation between the client application and the client-side representation of the Web service (proxy), including components for marshalling the invocation of a Web service over a chosen binding. In particular, the client application is insulated from the details of the Web service invocation by a higher-level abstraction: the client-side interface. Various runtime tools can take the WSDL description of the Web service and generate interface definitions in a wide-range of programming language specific constructs (e.g. Java interfaces). This interface is a front-end to specific parameter marshalling and message routing that can incorporate various binding options provided by the WSDL. Further, this approach allows certain efficiencies, for example, detecting that the client and the Web service exist on the same network host, and therefore avoiding the overhead of preparing for and executing the invocation using network protocols. One example of this approach to Web services is the Web Services Invocation Framework [7].

Within the client application runtime, a *proxy* provides a client-side representation of remote service instance's interface. Proxy behaviors specific to a particular encoding and network protocol (*binding* in Web services terminology) are encapsulated in a *protocol (binding)-specific stub*. Details related to the binding-specific access to the Grid service, such as correct formatting and authentication mechanics, happen here; thus, the application is not required to handle these details itself.

We note that it is possible, but not recommended, for developers to build customized code that directly couples client applications to fixed bindings of a particular Grid service. Although certain circumstances demand potential efficiencies gained this style of customization, this approach

introduces significant inflexibility into the system and therefore should be used under extraordinary circumstances.

3.3 Relationship to Hosting Environment

OGSA does not dictate a particular service provider-side implementation architecture. A variety of approaches are possible, ranging from implementing the Grid service directly as an operating system process to a sophisticated server-side component model such as J2EE. In the former case, most or even all support for standard Grid service behaviors (invocation, lifetime management, registration, etc.) is encapsulated within the user process, for example via linking with a standard library; in the latter case, many of these behaviors will be supported by the hosting environment.

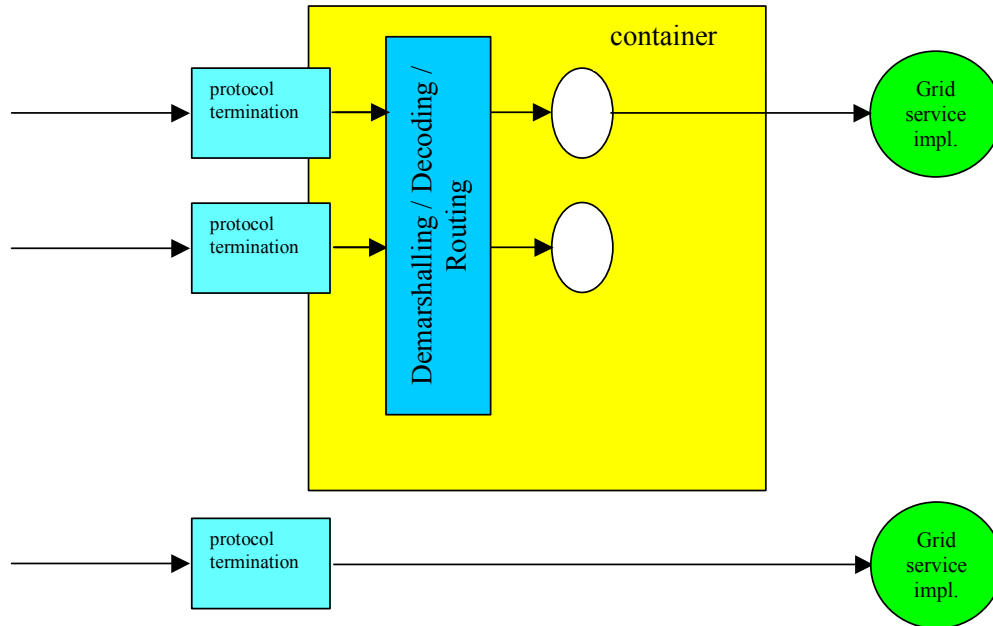


Figure 2: Two alternative approaches to the implementation of argument demarshalling functions in a Grid Service hosting environment

Figure 2 illustrates these differences by showing two different approaches to the implementation of argument demarshalling functions. We assume that, as is the case for many Grid services, the invocation message is received at a network protocol termination point (e.g., an HTTP servlet engine), which converts the data in the invocation message into a format consumable by the hosting environment. At the top of Figure 2, we illustrate two Grid services (the ovals) associated with container-managed components (for example EJBs within a J2EE container). Here, the message is dispatched to these components, with the container frequently providing facilities for demarshalling and decoding the incoming message from a format (such as an XML/SOAP message) into an invocation of the component in native programming language. In some circumstances (the upper oval), the entire behavior of a Grid service is completely encapsulated within the component. In other cases (the lower oval), a component will collaborate with other server-side executables, perhaps through an adapter layer, to complete the implementation of the Grid services behavior. At the bottom of Figure 2, we depict another scenario wherein the entire behavior of the Grid service, including the demarshalling/decoding of the network message, has been encapsulated within a single executable. Although this approach may have some efficiency advantages, it provides little opportunity for reuse of functionality between Grid service implementations.

A container implementation may provide a range of functionality beyond simple argument demarshalling. For example, the container implementation may provide lifetime management functions, intercepting lifetime management functions and terminating service instances when a service lifetime expires or an explicit destruction request is received. Thus, we avoid the need to re-implement these common behaviors in different Grid service implementations.

4 The Grid Service

The purpose of this document is to specify the interfaces and behaviors that define a *Grid service*. In brief, a *Grid service* is a WSDL-defined service that conforms to a set of conventions relating to its interface definitions and behaviors. In this section, we expand upon this brief statement by in turn:

- Introducing a set of WSDL conventions that we make use of in our Grid service specification;
- Defining *Grid service description* and *Grid service instance*, as organizing principles for the extensions and their use;
- Defining *service data*, which provides a standard way for representing and querying meta-data and state data from a service instance;
- Defining the **serviceType** and **serviceImplementation** WSDL extensibility elements that we use to define the collection of **portTypes** that define a Grid service interface;
- Defining the **instanceOf** WSDL extensibility element that we use to link a Grid service instance to the Grid service description that it implements; and
- Defining the Grid Service Handle and Grid Service Reference constructs that we use to refer to Grid service instances;
- Providing example WSDL documents illustrating the extensibility elements;
- Defining a common approach for conveying fault information from operations;
- Defining the lifecycle of a Grid service instance.

In subsequent sections, we introduce the various **portTypes** defined by OGSA, starting with the GridService **portType** that must be supported by any Grid service and then proceeding to HandleMap, Notification, and the remainder of the portTypes that describe fundamental behavior of Grid services.

4.1 WSDL Extensions and Conventions

Web services technologies are designed to support loosely coupled, coarse-grained dynamic systems. As such, they do not fully address all needs of the types of distributed systems that OGSA is defined to support. To close this gap, this specification defines a set of WSDL extensions (defined using extensibility elements allowed by the WSDL language) and conventions on the use of Web services, which we list in Table 1 and define in more detail in subsequent sections. We emphasize that the extensions are not specific to Grid computing per se, but have general applicability within Web services as a means of structuring complex and long-lived stateful applications. We advocate their adoption within the broader Web services standards bodies such as the W3C.

Table 1: Proposed WSDL conventions and extensions introduced by OGSA

Concept	WSDL Element	Brief Description	See Section
Interface Naming	convention on portType name	Naming conventions and immutability of portType , serviceType , and serviceImplementation names.	4.2
compatibility-Assertion	Extends definitions	Mechanism to associate equivalent interface elements and implementations.	4.5.3
serviceType	Extends definitions	Named aggregation of portType elements forming an interface definition.	4.4.1
serviceImplementation	Extends definitions	Mechanism to assert implementation semantics with a serviceType .	4.4.2
instanceOf	Extends service	Mechanism to assert that a service element refers to a Grid service instance that conforms to the semantics defined by a serviceImplementation .	
Grid Service Handle	N/A	Conventional use of URL to act as unique identifier of a Grid service instance.	4.6.1
Grid Service Reference	N/A	Mechanism to convey capabilities of a service to a client. <i>Can</i> be a WSDL document.	4.6.2
handleMap	portType definition	OGSA basic service to map GSH into GSR.	6

4.2 Service Description and Service Instance

We distinguish in OGSA between the *description* of a Grid service and an *instance* of a Grid service:

- A *Grid service description* is a **serviceImplementation** extensibility element (defined in Section 4.4.2) and associated definitions (e.g. **serviceType**, **portTypes**, etc.). Its purpose is to describe how a client interacts with service instances, independent of any particular instance. This is an example of an interface definition, commonly found in computing.
- A Grid service description may be simultaneously used by any number of *Grid service instances*, each of which:
 - embodies some state with which the service description describes how to interact;
 - has one or more unique Grid Service Handles;
 - and has one or more Grid Service References to it.

A common form of Grid Service Reference (defined in section 4.6.2) is a WSDL document comprising a **service** element that contains an **instanceOf** extensibility element referring to the service description.

A service description is primarily used for two purposes. First, it can be used by tooling to automatically generate client interface proxies, server skeletons, etc. Second, it can be used for discovery, for example, to find a particular service instance that implements a particular service description, or to find a factory that can create instances with a particular service description.

The service description is meant to capture both interface syntax, as well as (in a very rudimentary fashion) semantics. Interface syntax is, of course, described by the collection of one or more **portTypes** referred to by the **serviceType**. Note that the **portTypes** may come from different namespaces.

Semantics may be inferred through the names assigned to the **portType**, **serviceType**, and **serviceImplementation** elements. For example, when defining a Grid service, one defines zero or more uniquely named **portTypes**, collects a set of **portTypes** defined from a variety of sources into a uniquely named **serviceType**, and then upon implementing the service assigns a unique **serviceImplementation** name to that implementation version. Concise semantics can be associated with each of these names in specification documents – and perhaps in the future through Semantic Web or other formal descriptions. These names, along with **compatibilityAssertions** that define relationships between the names, can then be used by clients to discover services with the sought-after semantics, by searching for service instances and factories with the appropriate names. Of course, the use of namespaces to define these names provides a vehicle for assuring globally unique names.

4.3 Service Data

We use the term *service data* to refer to descriptive information about a Grid service instance. This term encompasses both *meta-data* (information about the structure of a service instance) and *state data* (properties of the service instance, such as its lifetime).

Service data can be associated with a service instance in two ways. First, a service description MAY contain zero or more service data elements (SDEs) as part of its service description (**serviceType** and/or **portType**). We call these *structural SDEs*. Such elements apply to all service instances that conform to the description containing that **serviceType**. Each instance, of course, may have its own values for each structural SDE defined for the **portType**. Some structural SDEs can be defined as “constant” meaning that a single value is fixed and is the same for all instances of the service.

Second, each instance maintains a collection of its own instance-specific SDEs, which we call *instance SDEs*. This collection of instance SDEs MUST include all of the structural SDEs defined in the service description associated with the instance, and MAY include additional dynamic or static SDEs. The GridService::FindServiceData (see Section 5.2.1) operation is used to query this collection of instance SDEs.

A service instance’s service data is represented as a logical collection of **serviceData** elements, each of which contains an XML element conforming to some XML schema. We sometimes refer to the **serviceData** element as the *service data container*, and XML element that it contains as the *service data value*.

A **serviceData** element has the following grammar:

```
<gsdl:serviceData
  qname="qname"
  type="qname"
```

```

    <!-- extensibility attribute -->* >
    <import namespace="uri" location="uri"> *
    <!-- extensibility element --> *
</gsdl:serviceData>

```

Attributes of **serviceData** are:

- **qname**: A global name (i.e. qualified name) for this Grid service data element. This same qname MAY be used by any number of SDEs in the same or different service instances.
- **type**: The XML schema type of the element contained in the extensibility element.

Like any XML element, a **serviceData** element may additionally include any number of namespace declarations, by including attributes of the form **xmlns:ns="uri"**. Unless overridden by an explicit “xmlns:” declaration, the default namespace of an instance SDE is implicitly defined to be the Grid Service Handle (see Section 4.6.1) of that instance.

The **import** elements within **serviceData** MAY be used to bind the location of the schema definition to a namespace used in the **serviceData** attributes and the service data value. The location associated with the **type** attribute’s namespace SHOULD be a URL to a document containing the named XML schema definition referred to by that **type** attribute.

A **serviceData** element MUST have exactly one extensibility element that conforms to the **type** of the service data value. A **serviceData** element MAY have additional extensibility elements of other types, which MAY be used for application-specific extensions to service data such as meta-data associated with the service data value.

The OGSA specification does not specify how service data elements are represented internally. As discussed in Section 5, the GridService **portType** provides a FindServiceData operation that allows clients to issue queries against this collection of *logical* XML elements. We use the term *logical* XML element since there is no requirement for the Grid service implementation to actually maintain the SDEs in a persistent form. Instead, a Grid service instance MAY choose to create the XML elements dynamically from other data sources at the time the FindServiceData operation is invoked.

4.3.1 XML Element Lifetime Declaration Attributes

Since service data elements may represent point-in-time observations of dynamic state of a service instance, it is critical that consumers of service data be able to understand the valid lifetimes of these observations. The client MAY use this time-related information to reason about the validity and availability of the **serviceData** element, though the client is free to ignore the information at its own discretion.

We define three XML attributes, which together describe the lifetimes associated with an XML element and its sub-elements. These attributes MAY be used in any XML element that allows for extensibility attributes, including the **serviceData** element.

In the following description, we use the term “*value*” to refer to the body of an XML element (that is, the information carried between the <element> and </element> tags, including any sub-elements), as well as to any attributes of the element that are used to carry information for the element.

The three lifetime declaration attributes are:

- **gsdl:goodFrom="xsd:dateTime"**: Declares the time from which the *value* of the element is said to be valid. This is typically the time at which the value was created.

- **gsdl:goodUntil="xsd:dateTime"**: Declares the time until which the *value* of the element is said to be valid. This value **MUST** be greater than the **goodFrom** time.
- **gsdl:availableUntil="xsd:dateTime"**: Declares the time until which this element itself is expected to be available, perhaps with updated values. Prior to this time, a client **SHOULD** be able to obtain an updated copy of this element. After this time, a client **MAY** no longer be able to get a copy of this element. This attribute **MUST** be greater than the **goodFrom** time.

We use the following **serviceData** element example to illustrate and further define these lifetime declaration attributes:

```
<gsdl:serviceData qname="tns:foo" type="n1:sometype"
  goodFrom="2002-04-27T10:20:00.000-06:00"
  goodUntil="2002-04-27T11:20:00.000-06:00"
  availableUntil="2002-04-28T10:20:00.000-06:00">
  <n1:e1>
    <n1:e2>
      abc
    </n1:e2>
    <n1:e3 gsdl:goodUntil="2002-04-27T10:30:00.000-06:00">
      def
    </n1:e3>
    <n1:e4 gsdl:availableUntil="2002-04-27T20:20:00.000-06:00">
      ghi
    </n1:e4>
  </n1:e1>
</gsdl:serviceData>
```

The **goodFrom** and **goodUntil** attributes of the **serviceData** element refer to the value contained in the **serviceData** element's extensibility element, which in this example is `<n1:e1>`. These attributes declare to the consumer of this SDE what the expected lifetime is for this element's value, which in this example is from 10:20am until 11:20am EST on 27 April 2002. In other words, the consumer of the SDE is being advised that after 1 hour the SDE value is likely to no longer be valid, and therefore the client should query the service again for the SDE with the same **qname** ("tns:foo") to obtain a newer value of `<n1:e1>`.

The **availableUntil** does not refer to the value of the SDE, but rather to the availability of this named **serviceData** element itself. Prior to the declared **availableUntil** time, a client **SHOULD** be able to query the same service instance for an updated value of this named SDE. In this example, a client should be able to query the same service until 28 April 2002 10:20am EDT for the **serviceData** element named "tns:foo", and receive a response with an updated copy of the `<n1:e1>` value. However, after that time, such a query **MAY** result in a response indicating that no such service data element exists. In other words, the consumer of the SDE is being advised that it can expect to be able to obtain an updated value of this SDE for 1 day, but after that time the service may no longer have an SDE with the name "tns:foo".

It is sometimes not sufficient for lifetime information of a SDE to refer only to the complete SDE value. Rather, the value of a SDE may contain sub-elements with varying lifetimes. Any XML element contained within a **serviceData** element **MAY** use any combination of the **goodFrom**, **goodUntil**, and **availableUntil** attributes, assuming that the schema for that element allows for these extensibility attributes. Such attributes override the lifetime declarations contained in parent elements.

In the above example, the lifetime attributes carried in the **<serviceData>** element provide default values for all children of that element. For example, the **<n1:e2>** element uses these default values, as described above. However, the **<n1:e3>** element overrides the **goodUntil** attribute, thus stating that its value (“def”) is only expected to be valid for 10 minutes, instead of 1 hour as is declared in the **<serviceData>** element. Such a situation might arise if a portion of a complex element changes more quickly than other portions of the element. Likewise, the **<n1:e4>** element overrides the **availableUntil**, thus stating that the **<n1:e4>** element may no longer exist within **<n1:e1>** after 10 hours. In other words, after 10 hours, a client that queries for the value of this **serviceData** element MAY be returned a **<n1:e1>** element that does not contain a **<n1:e4>** sub-element.

It is RECOMMENDED that the XML schema for elements that are intended to be SDE values should allow all elements within their schema to be extended with these lifetime declaration attributes, in order to allow for fine-grained lifetime declarations. However, since the **serviceData** element supports extensible attributes, service data values that lack attribute extensibility can be enclosed with a **serviceData** element with the appropriate the lifetime declarations for that entire value.

Since a SDE may be an observation or “by-value copy” of service instance state or some other definitive source of the data, these attributes do not necessarily reflect temporal aspects of that definitive source. So the fact that a **serviceData** element has a particular **goodUntil** value does not necessarily imply that the underlying definitive source of that data will not change prior to that time.

Note: Since these attributes are not required, we need to define the semantics when they are not specified.

4.3.2 serviceDataDescription

In order to assist in discovery and tooling, information about a service instance’s service data elements should be published as part of the service description. Just as a service description uses **portTypes** to describe the set of operations that a service instance supports, we define a **portType** extensibility element that describes any service data elements that a service instance supports. We define a **serviceDataDescription** element, which extends **portType**, to convey this service data description information

Note that WSDL v1.1 does *not* support extensibility in the **portType**. However, indications from the W3C Web Services Description (WSD) working group strongly suggest that WSDL v1.2 will have an open extensibility model, including an extensibility element in **portType**. Since this is the natural location for a **serviceDataDescription**, we have decided to exploit this expected feature of WSDL v1.2. If the W3C WSD working group decides not to support extensibility of the **portType** element, then we will be required to devise a different method of placing a **serviceDataDescription** element into a WSDL document.

The **serviceDataDescription** element has the following grammar:

```
<gsdl:serviceDataDescription name="NCName"
  type="qname"
  minOccurs="nonNegativeInteger"?
  maxOccurs="nonNegativeInteger"?
  requiredQName="qname"?
  constant="boolean"?> *
  <wsdl:documentation .... />?
</gsdl:serviceDataDescription>
```

Each **serviceDataDescription** element contains the following information:

- **type**: The qualified name of an XML schema type of the service data value contained in any **serviceData** elements that conform to this **serviceDataDescription**. The **type** attribute of any **serviceData** element conforming to this **serviceDataDescription** MUST be the same as this **type** attribute of the **serviceDataDescription**.
- **minOccurs**: The minimum number of service data elements that MUST be contained by a service instance that conforms to this service data description. If this attribute is omitted, then it defaults to 0.
- **maxOccurs**: The maximum number of service data elements that MUST be contained by a service instance that conforms to this service data description. If this attribute is omitted, then the default is that there is no maximum.
- **requiredQName**: If a service instance includes one or more **serviceData** elements conforming to service data description (subject to **minOccurs** and **maxOccurs**), then at least one of those **serviceData** elements MUST have a **qname** attribute set to the value defined by **requiredQName**. If this attribute is omitted, then serviceData elements of this type MAY have any **qname** attribute.
- **constant**: If true, then the value of the instance service data element MUST be the same as the value of the structural service data element contained in the service description implemented by that instance. Further, that value MUST be constant for the lifetime of the instance. If this attribute is omitted, then the default is false.

A **serviceDataDescription** element is optional. However, if it exists then it MUST be an extensibility element within the WSDL **portType** element. A **portType** element MAY contain any number of **serviceDataDescription** elements.

4.4 Defining Service Types: *ServiceType* and *ServiceImplementation*

In WSDL and thus in OGSA, the interface to a service is defined by a collection of **portTypes**. In OGSA, we introduce the WSDL extensibility element **serviceType** (Table 1) in order to support the explicit representation of such collections, with a view to supporting discovery and change management (see Section 4.5). Each Grid service has a single **serviceType** element, which defines the **portTypes** that the service supports. The related **serviceImplementation** element represents a particular implementation semantic of an abstract interface (**serviceType**), in a manner analogous to a version number.

4.4.1 ServiceType

The WSDL extensibility element **serviceType** (Table 1) is used to represent an aggregation of **portTypes**. A **serviceType** is declared using the extensibility element under the WSDL **definitions** element, and has the following grammar:

```
<gsdl:serviceType name="NCName">
  <wsdl:documentation .... />?
  <gsdl:portTypeReference="qname" />*
  <gsdl:compatibility statementName="qname" />*
  <-- extensibility element --> *
</gsdl:serviceType>
```

The **serviceType** element contains the following information:

- The **name** attribute allows for **serviceTypes** to be globally and uniquely named by forming a QName in combination with the namespace of the WSDL definitions element. The name of a **serviceType** element must follow the conventions of Section 4.5.
- One or more **portTypeReference** elements, each of which refers (by QName) to a **portType** that is part of a Grid service interface. This information SHOULD be used by client tools to recognize the relationship of the **portTypes** as belonging to a single service, so that it can generate client proxies and binding stubs appropriately. This information MAY also be used by clients to simplify discovery of what **portTypes** are supported by a Grid service.
- The **compatibility** element associates the **serviceType** being defined with a **compatibilityAssertion** of the given QName. See Section 4.2 for more details on compatibility assertions.

4.4.2 ServiceImplementation

A **serviceImplementation** element represents a particular implementation semantic of an abstract interface (**serviceType**) and collection of compatibility assertions (see Section 0). This element is roughly analogous to the CORBA *serviceImpl* concept or version number. A **serviceImplementation** is declared using the extensibility element under the WSDL **definitions** element, which has the following grammar:

```
<gsdl:serviceImplementation name="NCName" serviceType="qname">
  <wsdl:documentation .... />?
  <gsdl:compatibility statementName="qname" />*
  <-- extensibility element --> *
</gsdl:serviceType>
```

The **serviceImplementation** element contains the following information:

- The **statementName** attribute allows for **serviceImplementations** to be globally and uniquely named by forming a QName in combination with the namespace of the WSDL **definitions** element. The name of a **serviceImplementations** element must follow the conventions of Section 0.
- The **serviceType** attribute identifies (by QName) the **serviceType** element associated with this **serviceImplementation**.
- The **compatibilityStatement** element associates the **serviceImplementation** being defined with a **compatibilityAssertion** of the given QName. See Section 4.2 for more details on compatibility assertions.

4.5 Interface Naming and Change Management

A critical issue in distributed systems is enabling the upgrade of services over time. This implies in turn that clients need to be able to determine when services have changed their interface and/or implementation. Here, we discuss this issue and some of the OGSA mechanisms that are used to address it.

4.5.1 The Change Management Problem

The semantics of a particular Grid service instance are defined by the combination of two things:

1. Its *interface definition*. Syntactically, the Grid service interface is defined by a **serviceType** and by the **portTypes**, **operations**, **messages**, and **types** implied by that

- serviceType**. Semantically, the interface typically is defined in specification documents such as this one.
2. *The implementation of the interface.* While expected implementation semantics are defined in interface specifications, ultimately it is the implementation that truly defines the semantics of any given Grid service instance. Implementation decisions and errors may result in a service having behaviors that are ill-defined in and/or at odds with the interface definition. Nonetheless, such implementation semantics may come to be relied upon, whether by accident or by design.

In order for a client to be able reliably to discover and use a Grid service instance, the client must be able to determine whether it is compatible with both of these two semantic definitions of the service. In other words, does the Grid service support the **serviceType** that the client requires? And does the implementation have the semantics that the client requires, such as a particular patch level containing a critical bug fix?

Further, Grid service descriptions will necessarily evolve over time. If a Grid service description is extended in a backward compatible manner, then clients that require the previous definition of the Grid service should be able to use a Grid service that supports the new extended description. Such backward compatible extensions might occur to the interface definition, such as through the addition of a new operation to the interface, or the addition of options to existing operations. Or, backward compatible extensions might occur through implementation changes, such as a patch that fixes a bug. For example, a new implementation that corrects an error that previously caused an operation to fail would generally be viewed as being backwards compatible.

However, if a Grid service description is changed in a way that is *not* backward compatible, a client must be able to recognize this as well. Again, this could be the result of incompatible changes to the interface or implementation of a Grid service. A bug fix that “fixes” an “erroneous” behavior that users have learned to take advantage of would not be backward compatible.

This discussion points to the need to be able to provide concise descriptions of both the interface and implementation of a Grid service, as well as to make unambiguous compatibility statements about Grid services that support different interfaces or implementations.

4.5.2 Naming Conventions for Grid Service Descriptions

In WSDL, each **portType** is globally and uniquely named via what is known as its qname—that is, the combination of the namespace containing the **portType** definition, and the name of the **portType** element within that namespace.

In OGSA, our concern with change management leads us to require further that all elements of a Grid service definition **MUST** be immutable. That is that the qname of a Grid service **serviceType**, **portType**, **operation**, **message**, and underlying **type** definitions **MAY** be assumed to refer to one and only one WSDL specification. If a change is needed, a new **serviceType** or **portType** **MUST** be defined with a new qname—that is, defined with a new local name, and/or in a new namespace.

4.5.3 Compatibility Assertions

Immutable service definitions ensure that services do not change their interface or implementation silently. However, clients then require mechanisms for determining when two elements with different names are in fact compatible.

The purpose of a **compatibilityAssertion** is to declare two elements of the same underlying *type* compatible. The following types of elements can be subjects of **compatibilityAssertions**:

portTypes, **serviceTypes**, **serviceImplementations**. The exact meaning of “compatible” can vary, depending on the type of the items being compared and the designer’s intended use of the **compatibilityAssertion**.

A **compatibilityAssertion** element MAY be added to the WSDL definition of a Grid service. This element uses the extensibility mechanism on the WSDL **definitions** element. This element has the following grammar:

```
<gsdl:compatibilityAssertion name="NCName">
  <wsdl:documentation .... />?
  <gsdl:compatible name="qname" withName="qname" type="NCName" />*
  <!-- extensibility element -->*
</gsdl:compatibilityAssertion>
```

This **compatibility** element MAY contain a set of **compatible** elements, each of which declares that a named element of a given type is compatible with another named element of the same type. The **type** attribute MUST take one of the values “**portType**,” “**serviceType**,” or “**serviceImplementation**.”

A **compatibilityAssertion** is not commutative; the assertion is unidirectional. For example, a **compatibilityAssertion** that **portType** “ns1:pt1” is compatible with “ns2:pt2,” does not imply that “ns1:pt2” is compatible with “ns2:pt2.”

Note: it is **not clear** at the moment what statements we can make about transitivity. If a designer declares the following, can a reader safely assume that “ns1:pt1” is compatible with “ns1:pt3”?

```
<gsdl:compatibilityAssertion name="example">
  <gsdl:compatible name="ns1:pt1" withName="ns1:pt2" type="portType" />
  <gsdl:compatible name="ns1:pt2" withName="ns1:pt3" type="portType" />
</gsdl:compatibilityAssertion>
```

A **compatibilityAssertion** can be used to state many different forms of compatibility, including (but not limited to) compatibility between two different versions of an element, backwards compatibility between two different versions of an element, and equivalency of alternate vendor’s implementations of a standard interface.

The **compatibilityAssertion** mechanism has many uses. For example, we can augment Registry (Section 9) functionality to find services that implement a particular **serviceImplementation** or a “compatible” one. To support this style of registry find operation, the registry may aggregate all **compatibilityAssertion** elements contained in the service descriptions published to the Registry. It is worth noting that there is no requirement for a Registry to be aware of all **compatibilityAssertion** elements that reference a particular **portType**, **serviceType** or **serviceImplementation**. A registry can only find “compatible” services based upon the **compatibilityAssertion** elements that have been published to it.

4.5.3.1 PortType Compatibility

In declaring **portType** PT1 to be compatible with **portType** PT2, we assert that for every **operation** defined by **portType** PT2, there MUST be an **operation** in **portType** PT1 with the same signature.

4.5.3.2 ServiceType Compatibility

In declaring **serviceType** ST1 to be compatible with **serviceType** ST2, we assert that for every **portType** aggregated by **serviceType** ST2, there MUST be a compatible **portType** aggregated by ST1.

4.5.3.3 ServiceImplementation Compatibility

In declaring **serviceImplementation** SI1 to be compatible with **serviceImplementation** SI2, we assert that the **serviceType** associated with SI2 is compatible with the **serviceType** associated with SI1, and the **serviceTypes** are semantically compatible.

A **serviceType** ST1 is semantically indistinguishable from the **serviceType** ST2 if for each **portType** aggregated by ST2, there is a semantically indistinguishable **portType** in ST1.

A **portType** PT1 is semantically indistinguishable from **portType** PT2 if each operation defined in PT2 can be replaced by a compatible operation in PT1 without altering the results of the operation or changing the state of the service in a way that would make future operations behave differently than if the operation in PT2 had been used.

Thus, if a client is looking for a service that is an instance of **serviceImplementation** SI2, the **compatible** element declares that a service that is an instance of **serviceImplementation** SI1 can be safely used instead.

4.5.3.4 CompatibilityAssertion Mutability

The contents of a **compatibilityAssertion** can be changed. New **compatible** elements can be added, reflecting newly discovered similarity with other **portTypes**, **serviceTypes** or **serviceImplementations** that were not known at the time the **compatibilityAssertion** was created. A **compatible** element may be removed, for example, reflecting information derived from more extensive testing that reveals two **serviceImplementations** are not in fact compatible.

Note: given that a **compatibilityAssertion** can change, there needs to be some statement of duration of validity. Should this be similar to the **serviceData** elements (Section 5.1) or should this be a simple timestamp?

Note: is there the requirement to add the notion of a “compatibility authority” attribute formerly into the specification of a **compatibilityAssertion**? The purpose of this attribute would be to assert the party that created the assertion. This would be used in an analogous fashion to certificate authority in PKI-based systems. Can this be accommodated by the extensibility element, or must this be a formally defined part of the **compatibilityAssertion** element. Is it a required attribute?

4.6 Naming Grid Service Instances: Handles and References

Each Grid service instance is globally, uniquely, and for all time named by a *Grid Service Handle (GSH)*. However, the GSH is just a minimal name in the form of a URL, and does not carry enough information to allow a client to communicate directly with the service instance. Instead, a GSH must be *mapped* to a *Grid Service Reference (GSR)*. A GSR contains all information that a client requires to communicate with the service via one or more network protocol bindings.

The actual format of the GSR is specific to the binding mechanism used by the client to communicate with the Grid service instance. For example, if an RMI/IIOP binding were used, the GSR would take the format of an IOR. If a SOAP binding were used, the GSR would take the form of a properly annotated WSDL document.

While a GSH is valid for the entire lifetime of the Grid service instance, a GSR may become invalid, therefore requiring a client to use the mapping service to acquire a new GSR, appropriate to a particular binding, using the GSH.

4.6.1 Grid Service Handle (GSH)

Each Grid service instance **MUST** have an associated *Grid Service Handle (GSH)*. The GSH **MUST** be a globally unique, printable string, which uniquely names the Grid service for all time. Two Grid service instances **MUST NOT** ever have the same GSH.

A GSH for any Grid service **MUST** be a URL with the following format:

<GSHomeHandleMapID>/<GSInstanceID>

where:

1. <GSHomeHandleMapID> is a globally unique identifier for the GSH's home handleMap service (Section 6), which is responsible for mapping this GSH to one or more valid GSRs at all points during the lifetime of the service instance referred to by the GSH. The format of the <GSHomeHandleMapID> is:

<scheme>://<hostname>[:<port>]/<path>

where:

- A. <scheme> is either "http" or "https."
 - B. <hostname> is a fully qualified hostname.
 - C. <port> (optional) is an positive integer. If <port> is not specified, then it defaults to the standard port for the <scheme>, which is 80 for "http" and 443 for "https."
 - D. <path> consists of zero or more '/'-separated path components.
2. <GSInstanceID> is a single path component, and therefore **MUST NOT** include the '/' character. Each GSH with the same <GSHomeHandleMapID> **MUST** have a <GSInstanceID> that is unique for all time. The <GSInstanceID> value "handleMap" is reserved for use only by the service that is acting as the home handleMap for this <GSHomeHandleMapID>. All other values are opaque to the client, interpretable only to the handleMap referred to by <GSHomeHandleMapID>.

For example, the following is a valid GSH:

http://server.ggf.org:2345/gridservices/examples/instance42

where the GSHomeHandleMapID "http://server.ggf.org:2345/gridservices/examples" and the GSInstanceID is "instance42."

A GSH **MAY** be used for the following purposes:

1. As a globally unique name for the GS, it **MAY** be used any purpose for which such a name is useful.
2. An http GET performed on the GSH with a ".gsr" suffix **MUST** return the GSR for the Grid service instance associated with the GSH (modulo security, if the URL scheme is https). In this case, since an http protocol is used, the format of the GSR is a WSDL document. (Notice that if https is used, then the server may decline to respond with a GSR.) Note: We need to define how a client knows that it is talking to the right server, if this is an https URL.
3. The <GSHomeHandleMapID> component of the GSH identifies, and can be used to locate, the home handleMap service for this GSH. As an alternative to #2, this is a Grid service instance that implements the HandleMap **portType** (see Section 6), which can

be asked to map this GSH to a GSR. This home handleMap service is itself a Grid service, with a GSH of “<GSHomeHandleMapID>/handleMap.”

4.6.2 Grid Service Reference (GSR)

Grid service instances are made accessible to (potentially remote) client applications through the use of a Grid Service Reference (GSR). A GSR is fundamentally a network-wide pointer to a specific Grid service instance that is hosted in an environment responsible for its execution. A client application can use a GSR to send requests (represented by the operations defined in the WSDL **portType**(s) of the target service) directly to the specific instance at the specified (potentially network-attached) service endpoint identified by the GSR. In other words, the GSR supports the programming notion of passing Grid service instances "by reference". The GSR contains all of the information required to access the Grid service instance resident in its hosting environment over one or more communication protocol bindings.

A new WSDL type definition is introduced to represent a Grid Service Reference. The <**gsdl:reference**> XML schema definition is used to represent the GSR so that references may be introduced into typed message parts in the operation signatures of a WSDL service interface definition.

The encoding of a Grid Service Reference may take many forms in the system. Like any other operation message parts, the actual encoded format of the GSR "on the wire" is specific to the Web service binding mechanism used by the client to communicate with the Grid service instance. Below we define a WSDL encoding of a GSR that MAY be used by some bindings, but the use of any particular encoding is defined in binding specifications, and is therefore outside of the scope of this specification. However, it is useful to elaborate further on this point here. For example, if an RMI/IIOP binding were used, the GSR would take the format of a CORBA compliant IOR. If a SOAP binding were used, the GSR may take the form of the WSDL encoding defined below. This "on the wire" form of the Grid Service Reference is created both in the Grid service hosting environment, when references are returned as reply parameters of a WSDL defined operation, and by the client application or its designated execution environment when references are passed as input parameters of a WSDL defined operation. This "on the wire" form of the Grid Service Reference, passed as a parameter of a WSDL defined operation request message, must include all of the service endpoint binding address information required to communicate with the associated service instance over any of the communication protocols supported by the designated service instance, regardless of the Web service binding protocol used to carry the WSDL defined operation request message.

Any number of Grid Service References to a given Grid service instance may exist in the system. The lifecycle of a GSR is independent of the lifecycle of the associated Grid service instance. However, the GSR is valid only when the associated Grid service instance exists and can be accessed through use of the Grid Service Reference. A GSR may become invalid during the lifetime of the Grid service instance. Typically this occurs because of changes introduced at the Grid service hosting environment. These changes may include modifications to the Web service binding protocols supported at the hosting environment, or of course, the destruction of the Grid service instance itself. Use of an invalid Grid Service Reference by a client will result in an exception being presented to the client.

When a Grid Service Reference is found to be invalid and the designated Grid service instance exists, a client can obtain a new GSR using the Grid Service Handle of the associated Grid service instance, as defined in Section 4.6.1. It is RECOMMENDED that the Grid Service Handle be contained within each binding-specific implementation of the Grid Service Reference. Maintaining the GSH within the GSR facilitates a cached copy of the GSH at the client, allowing

it to avoid a (potentially time-consuming) round-trip to the service instance to retrieve the GSH from the instance's service data. In addition, reliability of the system may be improved since the acquisition of a GSH for a given Grid service instance by a client would not have to be dependent upon a valid GSR being held and used by the client. (Note: we may consider changing this to a requirement that a GSR **MUST** contain the GSH, in order to allow `Factory::createService` to return an GSR, yet still avoid any possibility of a race condition due to invalidation of the returned GSR prior to retrieving a GSH.)

A binding-specific implementation of a Grid Service Reference **MAY** include an expiration time, which is a declaration to clients holding that GSR that the GSR **SHOULD** be valid prior to that time, and it **MAY NOT** be valid after the expiration time. After the expiration time, a client **MAY** continue to attempt to use the GSR, but **SHOULD** retrieve a new GSR using the GSH of the Grid service instance. While an expiration time provides no guarantees, it nonetheless is a useful optimization in that it allows clients to refresh GSRs at convenient times (perhaps simultaneously with other operations), rather than simply waiting until the GSR becomes invalid, at which time it must perform the (potentially time-consuming) refresh before it can proceed.

Mere possession of a GSR does not entitle a client to invoke operations on the Grid service. In other words, a GSR is not a capability. Rather, authorization to invoke operations on a Grid service instance is an orthogonal issue, to be addressed elsewhere.

4.6.2.1 WSDL Encoding of a GSR

We define a standard encoding of a GSR in WSDL. This GSR encoding **SHOULD** be used by any network protocol bindings that use an XML encoding, and **MAY** be used by any other bindings.

The WSDL encoding of a GSR **MUST** be a conformant WSDL document, which **MUST** have exactly one **service** element, and **MAY** contain other WSDL elements. This **service** element **MUST** contain the **instanceOf** extensibility element.

The **instanceOf** element has the following grammar:

```
<gsdl:instanceOf serviceImplementation="qname"
  handle="uri" expiration="xsd:dateTime"?>
  <wsdl:documentation .... />?
  <-- extensibility element --> *
</gsdl:implements>
```

This **instanceOf** element contains the following attributes:

- The **serviceImplementation** attribute refers (by qname) to the **serviceImplementation** of the Grid service (see Section 4.4.2).
- The **handle** attribute is the Grid Service Handle of this service. See Section 4.6.1 for a definition of the Grid Service Handle.
- The **expiration** attribute is the date and time after which this WSDL service element is no longer valid. See Section 4.6.2 for more about expiration.

When a **service** element contains an **instanceOf** element it implies that the service represents a specific instance implementation of the corresponding **serviceImplementation**. This means that the service endpoint fulfills the interface implied by the **serviceType** of the **serviceImplementation** in a manner consistent with the compatibility assertions associated with the **serviceImplementation**.

It is RECOMMENDED that this WSDL document be the minimal information required to describe fully how to reach the particular Grid service instance. This information will commonly be just the WSDL **service** element, which in turn contains references (qnames) to elements in other namespaces of the other WSDL elements that are non-instance specific.

4.7 Grid Service WSDL Examples

The following is an example of a WSDL document for a Grid service description, including the extensibility elements defined in this section.

A **serviceType** declaration can appear in a WSDL document as shown below. Note: this can be in the same WSDL document declaring a Grid service instance, or a separate document.

```
<wsdl:definitions name="ServiceTypeExample"
  targetNamespace="http://gridExample/Aggregate3.wsdl"
  xmlns:tns="http://gridExample/Aggregate3.wsdl"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  xmlns:ns1="http://gridExample/someNamespace"
  xmlns:ns2="http://gridExample/someOtherNamespace"
  xmlns:comp="http://gridExample/myCompatibilities.wsdl"
  xmlns="http://schemas.xmlsoap.org/wSDL/">

  <gsdl:serviceType name="IAggregate3">
    <wsdl:documentation>
      Steve's first serviceType
    </wsdl:documentation>
    <gsdl:portTypeReference="ns1:pt1"/>
    <gsdl:portTypeReference="ns1:pt2"/>
    <gsdl:portTypeReference="ns2:pt3"/>

    <gsdl:compatibility statementName="comp:IA3Compatibilities"/>
  </gsdl:serviceType>
  ...
</wsdl:definitions>
```

A **serviceImplementation** element declares certain semantics related to a particular implementation of this **serviceType**:

```
<wsdl:definitions name="ServiceImplExample"
  targetNamespace="http://gridExample/Aggregate3Impl.wsdl"
  xmlns:tns="http://gridExample/Aggregate3Impl.wsdl"
  xmlns:IA3="http://gridExample/Aggregate3.wsdl"
  xmlns:comp="http://gridExample/myCompatibilities.wsdl"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  xmlns="http://schemas.xmlsoap.org/wSDL/">

  <gsdl:serviceImplementation name="IAggregate3GridCoImpl"
    serviceType="IA3:IAggregate3">
    <wsdl:documentation>
      This is GridCo's implementation of IAggregate3 ServiceType.
      General Available version 1.0.
    </wsdl:documentation>
```

```

    <gSDL:compatibility statementName="comp:IA3Compatibilities"/>
</gSDL:serviceType>
...
</wsdl:definitions>

```

And the use of this **serviceImplementation** declaration would appear in a **service** element of a WSDL encoded Grid Service Reference:

```

<wsdl:definitions name="ServiceTypeUseExample"
  targetNamespace="http://gridExample/UseOfAggregate3.wsdl"
  xmlns:tns="http://gridExample/UseOfAggregate3Impl.wsdl"
  xmlns:IA3Impl="http://gridExample/Aggregate3Impl.wsdl"
  xmlns:gSDL="http://schemas.gridforum.org/gridServices/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:bdgs="http://gridExample/bindings"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:service name="StevesFirstService">
    <gSDL:instanceOf
      serviceImplementation="IA3Impl: IAggregate3GridCoImpl">
    </gSDL:instanceOf>

    <wsdl:documentation>
      Steve's first service using serviceType IAggregate3
    </wsdl:documentation>

    <wsdl:port name="port1" binding="bdgs:pt1_SOAP_HTTP">
      <soap:address location="www.steve.com/services/numeroUno"/>
    </wsdl:port>
    <wsdl:port name="port2" binding="bdgs:pt2_SOAP_HTTP">
      <soap:address location="www.steve.com/services/numeroUno"/>
    </wsdl:port>
    <wsdl:port name="port3" binding="bdgs:pt3_SOAP_HTTP">
      <soap:address location="www.steve.com/services/numeroUno"/>
    </wsdl:port>

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

```

A single **compatibilityAssertion** is also published:

```

<wsdl:definitions name="aCompatibilityAssertion"
  targetNamespace="http://gridExample/myCompatibilities.wsdl"
  xmlns:tns="http://gridExample/myCompatibilities.wsdl"
  xmlns:IA3="http://gridExample/Aggregate3.wsdl"
  xmlns:IA3Impl="http://gridExample/Aggregate3Impl.wsdl"
  xmlns:ano="http://another.com/gridServices/Aggregate3Impl.wsdl"
  xmlns:gSDL="http://schemas.gridforum.org/gridServices/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:ns1="http://gridExample/someNamespace"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <gSDL:compatibilityAssertion name="IA3Compatibilities">
    <wsdl:documentation>
      These compatibility statements were made by Grid Co
    </wsdl:documentation>
  </gSDL:compatibilityAssertion>

```

```

</wsdl:documentation>

<gsdl:compatible name="IA3:IAggregate3"
                 withName="ns1:anotherAggregator"
                 type="serviceType"/>

<gsdl:compatible name="IA3Impl:IAggregate3GridCoImpl"
                 withName="ano:YetAnotherIA3Impl"
                 type="serviceImplementation"/>

<gsdl:compatible name="IA3Impl:IAggregate3GridCoImpl"
                 withName="IA3Impl:IAggregate3GridCoImplBeta"
                 type="serviceImplementation"/>
</gsdl:compatibilityAssertion>

</wsdl:definitions>

```

Note: Need to add serviceData to this example.

4.8 Grid Service Lifecycle

The lifecycle of any Grid service is demarked by the creation and destruction of that service. The actual mechanisms by which a Grid service is created or destroyed are fundamentally a property of the hosting environment, and as such are not defined in this document. There is nonetheless a **serviceType** and a collection of related **portTypes** defined here that specify how clients may interact with these lifecycle events in a common manner. As we describe in subsequent sections:

- A client may request the *creation* of a Grid service by invoking the create operation on a *Factory service*. (A **serviceType** that includes the Factory **portType**.)
- A client may request the *destruction* of a Grid service via either client invocation of an *explicit* destruction operation request to the Grid service (the Destroy operation, supported by the GridService portType: Section 5) or via a *soft-state* approach, in which (as motivated and described in [4]) a client registers interest in the Grid service for a specific period of time, and if that timeout expires without the service having received re-affirmation of interest from any client to extend the timeout, the service may be automatically destroyed. Periodic re-affirmation can serve to extend the lifetime of a Grid service as long as is necessary (the SetTerminationTime operation in the GridService **portType**: Section 5).

In addition, a Grid service MAY support notification of lifetime-related events, through the standard notification interfaces defined in Section 7, and standard XML types for notification data (NSDataType) and interest statements (NSInterestType) related to lifecycle events.

A Grid service MAY support soft state lifetime management, in which case a client negotiates an initial service instance lifetime when the Grid service is created through a factory (Section 8), and authorized clients MAY subsequently send SetTerminationTime (“keepalive”) messages to request extensions to the service’s lifetime. If the Grid service termination time is reached, the server hosting the service MAY destroy the service, reclaim any resources associated with the service, and remove any knowledge of the service maintained by the home handleMap (see Section 4.6).

Termination time MAY change non-monotonically. That is, a client MAY request a termination time that is earlier than the current termination time. If the requested termination time is before the current time, then this SHOULD be interpreted as a request for immediate termination.

A Grid service MAY decide at any time to extend its lifetime. A service MAY also terminate itself at any time, for example if resource constraints and priorities dictate that it relinquish its resources.

Note that the GMT global time standard is assumed for Grid services, allowing operations to refer unambiguously to absolute times. Grid service hosting environments and clients SHOULD utilize the Network Time Protocol or equivalent function to synchronize their clocks to the global standard GMT time. However, no specific accuracy of synchronization is specified or expected as this is a service-quality issue. Clients and services MUST accept and act appropriately on messages containing time values that might be out of range due to inadequate synchronization. Furthermore, clients and services requiring global ordering or synchronization MUST coordinate through the use of additional synchronization service interfaces, i.e. through transactions or synthesized global clocks.

4.9 Common Handling of Operation Faults

OGSA will define a small collection of base fault messages that may be used by the **portTypes** defined in this document. These faults will be described in this Section in a subsequent version of this document.

4.10 Summary of PortTypes Defined in this Document

Table 2 Summary of the portTypes defined in this document

PortType Name	See Section	Description
GridService	5	encapsulates the root behavior of the component model
HandleMap	6	mapping from a GSH to a GSR
NotificationSource	7.1	allows clients to subscribe to notification messages
NotificationSink	7.2	defines a single operation for delivering a notification message to the service instance that implements the operation
Factory	8	standard operation for creation of Grid service instances
Registry	9.2	allows clients to register and unregister registry contents

5 The GridService PortType

In this section and those that follow, we describe the various standard **portTypes** that are defined by OGSA. We start with the GridService **portType**, which must be implemented by any Grid services and that thus serves as the base interface definition in OGSA. This **portType** is analogous to the base Object class within object-oriented programming languages such as Smalltalk or Java, in that it encapsulates the root behavior of the component model.

At the core of the OGSA model is the ability of any Grid service instance to convey information about itself, including metadata and state. This core behavior allows clients to query the data of the service instance in a well-known, common manner.

In Web services interface design, there is a choice to be made between document-centric messaging patterns and remote procedure call (RPC). Using a document-centric approach, the interface designer defines a loosely coupled interaction pattern wherein the API to the service is defined in terms of document exchange; both input and output are XML documents. This

approach shifts the complexity of the interaction away from the API level and into the data format of the document exchange itself. This style tends to yield simpler, more flexible APIs. The RPC approach defines a specific, strongly-typed operation signature. This approach tends to produce less flexible API, but is often easier to map onto APIs of existing objects and can have better runtime performance.

Designers of Grid service interfaces also face the document-centric vs. RPC choice. The GridService **portType** provides a document-centric approach to querying service data. Grid service designers are free to mix and match the document-centric and RPC approaches. Both mechanisms are provided in the GridService **portType**.

5.1 GridService PortType: ServiceData Elements

Each Grid service MUST make the following types of **serviceData** elements available:

```
<gsdl:serviceDataDescription name="GridServiceType"
  type="gsdl:serviceType"
  minOccurs="1"
  maxOccurs="1"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
  <wsdl:documentation>
    The serviceType implemented by this Grid service.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription name="GridServiceHandle"
  type="gsdl:handleType"
  minOccurs="1"
  maxOccurs="1"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
  <wsdl:documentation>
    The Grid Service Handle for this Grid service.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription name="GridServiceDescription"
  type="wsdl:definitions"
  minOccurs="1"
  maxOccurs="1"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:documentation>
    The WSDL representation of the service description for this Grid
    service.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription name="GridServiceFactoryHandle"
  type="gsdl:handleType"
  minOccurs="0"
  maxOccurs="1"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
  <wsdl:documentation>
    The Handle to the factory that created this Grid service
    instance.
  </wsdl:documentation>
</gsdl:serviceDataDescription>
```

```

<gsdl:serviceDataDescription name="GridServiceQueryExpressionTypes"
  type="gsdl:QueryExpressionListType"
  minOccurs="0"
  maxOccurs="unbounded"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
  <wsdl:documentation>
    A list of qnames, which is the set of types that can be used to
    express queries to a Grid service via its query operation.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription name="GridServiceTerminationTime"
  type="xsd:dateTime"
  minOccurs="1"
  maxOccurs="1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:documentation>
    The dateTime value of the time when the lifetime for the Grid
    service has expired.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription name="GridServiceDataDescriptions"
  type="gsdl:serviceDataDescription"
  minOccurs="1"
  maxOccurs="1"
  xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
  <wsdl:documentation>
    The collection of serviceDataDescriptions of service data
    elements
    contained directly under the root service data element. A service
    instance is not required to include all such service data
    elements, but may choose to publish just a subset of the service
    data element names through GridServiceDataDescriptions.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

```

5.2 GridService PortType: Operations and Messages

5.2.1 GridService :: FindServiceData

Query the service data.

Input

- *QueryExpression*: The query to be performed, expressed as a valid XML element, which conforms to the XML schema of a query expression type understood by the target Grid service instance.

Output

- *Result*: The result of the query. The format of this result is dependent upon the QueryExpression.

Fault(s)

- TBD.

Every Grid service instance MUST support *QueryExpressions* that conform to the `queryByServiceDataName` type, as described in Section 5.2.2. A Grid service instance MAY support other types of *QueryExpressions*.

The list of query expression types supported by a Grid service are expressed in the service data of that Grid service, with a service data element name of “GridServiceQueryExpressionTypes” (see Section 15). A `queryByServiceDataName` on this element can be used to retrieve this list.

5.2.2 queryByServiceDataName

queryByServiceDataName is an XML schema type defined in the “<http://schemas.gridforum.org/gridService>” namespace.

The `GridService::FindServiceData` operation of every Grid service instance MUST support *QueryExpressions* that conform to this type.

The grammar of this type is:

```
<gsdl:queryByServiceDataName qname="qname" />
```

A `queryByServiceDataName` results in all service data elements which have a **qname** with the specified qname.

The *Result* output parameter a `queryByServiceDataName` query is list of zero or more **serviceData** elements that resulted from the query expression.

5.2.3 GridService :: SetTerminationTime

Request that the termination time of this service be changed. The request specifies a minimum and maximum requested new termination time and includes a timestamp with the request. Upon receipt of the request, the service MUST discard any requests that have arrived out of order based on the timestamp; MAY adjust its termination time, if necessary, based on its own policies and the requested minimum and maximum; and if acknowledgement is requested MUST return the new termination time, a timestamp of when this new termination time was set, and a maximum lifetime extension allowed for subsequent requests. Upon receipt of the response, the client SHOULD discard any responses that have arrived out of order, based on the timestamp in the response.

Input:

- *ClientTimestamp*: The time at which the client generated the request. Any request MUST be discarded by the service that has a *ClientTimestamp* that is earlier than the latest *ClientTimestamp* received in a previous `SetTerminationTimeWithAck` or `SetTerminationTimeWithoutAck` request.
- *EarliestTerminationTime*: The earliest termination time of the Grid service that is acceptable to the client.
- *LatestTerminationTime*: The latest termination time of the Grid service that is acceptable to the client. This value must be \geq *EarliestTerminationTime*. If *LatestTerminationTime* is prior to the time at which the message is received by the service, this SHOULD imply immediate termination of the service.

Output:

- *ServiceTimestamp*: The time at which the Grid service handled the request.

- *CurrentTerminationTime*: The service's currently planned termination time.
- *MaximumExtension*: The maximum extension that the service will currently allow a client to request of its termination time. This value SHOULD change infrequently over the lifetime of the service, but a service MAY change this value at any time.

Fault(s):**5.2.4 GridService :: Destroy**

Explicitly request destruction of this service. Upon receipt of an explicit destruction request, a Grid service MUST either initiate its own destruction and return a response indicating success; or ignore the request and return a fault message indicating failure. Once destruction of the Grid service is initiated, any subsequent operation invocations by clients to that service will be denied.

Input:

- None

Output:**Fault(s):****6 The HandleMap PortType**

As described in Section 4.6.1, one method of mapping from a GSH to a GSR is to perform an http GET on the GSH. Another method is through a service invocation to a *handleMap service*: that is, a Grid service that implements the HandleMap **portType**.

For each Grid service, there MUST be one *home handleMap service* that MUST return (modulo authorization) a GSR for the Grid service's GSH for the lifetime of that Grid service. There MAY be additional handleMap services that MAY return a GSR for a GSH, for example from a cache of mappings gathered from other handleMap services.

As described in Section 4.6.1, given a GSH to a service, the GSH for that service's home handleMap service can be easily derived, simply by replacing the <GSInstanceID> portion of the GSH with the string "handleMap."

6.1 HandleMap PortType: ServiceData Elements

None.

6.2 HandleMap PortType: Operations and Messages**6.2.1 HandleMap :: FindByHandle**

Returns a Grid Service Reference for a Grid Service Handle.

Input

- *Handle*: A Grid Service Handle.

Output

- *Reference*: A Grid Service Reference in the format specific to the binding used to invoke the FindByHandle operation.

Fault(s)

- *InvalidHandle*, indicating the operation failed to resolve the handle.

7 Notification

The purpose of notification is to deliver interesting messages from a notification source to a notification sink, where:

- A *notification source* is a Grid service instance that implements the NotificationSource **portType**, and is the sender of a notification message. A source MAY be able to send notification messages to any number of sinks.
- A *notification sink* is a Grid service instance that implements the NotificationSink **portType**, and is the receiver of a notification message. A sink MAY be able to receive notification messages from any number of sources.
- A *notification message* is contained in a **serviceData** element. That is, the message is an XML element, conforming to an XML schema type.
- A source contains, as part of its **serviceData**, a set of *topics* about which it is able to send notification messages. A topic includes:
 - The type of all notification messages that this source may generate for this topic.
 - 0 or more Grid service instances (*subjects*) to which messages of this topic refer.
 - Additional metadata about the topic.
 - An XML type that can be used during subscription to constrain which of the messages of this topic are of interest to the sink.
 - A name for this topic.
- In order to establish what and where notification messages are to be delivered, a *subscription* request is issued to a source, containing the topic to which to subscribe, interest constraints on that topic, the GSH of the notification sink to which notification messages are to be sent, and a expiration for the subscription.
- Subscription requests are soft-state. A subscription will expire after some period time, unless the subscription is re-affirmed and extended prior to that expiration time.
- Because source topics are maintained as **serviceData**, numerous approaches to discovery of such topics can be supported, including for example by registries (see Section 9).

This notification framework allows for either direct service-to-service notification message delivery, or for the ability to integrate various intermediary delivery services including messaging service products commonly used in the commercial world.

Note: it is worth considering a unification of the notification model and the serviceData query model proposed for GridService portType. The notion is to unify the two models around serviceData elements. A client can pull serviceData using the GridService::findServiceData operation, or subscribe to a service to have (changing) serviceData pushed to the client as notification messages. In both cases, a query expression can be used to express what serviceData (i.e topics) are of interest, and under what constraints it should be sent. Such a unification would mean leaving the basic notification model the same, and just changing some of the details of how topics and constraints are expressed.

7.1 The NotificationSource PortType

The NotificationSource **portType** allows clients to subscribe to notification messages from the Grid service instance that implements this **portType**.

7.1.1 Notification Topics

A Grid service instance that implements the NotificationSource **portType** may have any number of topics to which clients may subscribe. Each topic **MUST** be described in the instance's **serviceData**, with an entry with the following grammar:

```
<gsdl:serviceData name="NCName"?
  goodFrom="xsd:dateTime" goodUntil="xsd:dateTime"
  <gsdl:notificationSourceTopic
    messageType="qname"
    constraintType="qname"?>
    <gsdl:topicSubject handle="uri"> *
    <gsdl:topicMetadata>
      <!-- extensibility element --> *
    </gsdl:topicMetadata>
  </gsdl:notificationSourceTopic>
</gsdl:serviceData>
```

Where:

- **serviceData name**: the topic name.
- **serviceData goodFrom, goodUntil**: these follow the standard serviceData interpretation, regarding the validity time of the **notificationSourceTopic** element contained with this serviceData element.
- **notificationSourceTopic messageType**: the qname of the XML schema type that describes all messages that will be sent as a result of a subscription to this topic.
- **notificationSourceTopic constraintType**: the qname of the XML schema type that describes the how a subscriber can place additional constraints on which messages of this topic will be sent. If this attribute is not specified, then no additional constraints can be placed during a subscription to this topic.
- **topicSubject handle**: the GSH of a Grid service instance to which messages of this topic **MAY** pertain. There may be 0 or more **topicSubjects** declared.
- **topicMetadata**: Any additional meta-data describing the topic or messages **MAY** be added as an extensibility elements under this element.

The **constraintType** is specific to the nature of the topic and the messages that are sent for the topic. Each subscription to a notification source topic may specify different constraints, in order to select the subset of all topic messages that the sink is interested in receiving. Examples of constraints that might be expressed via the **constraintType** include:

- The maximum frequency at which messages regarding this topic should be sent.
- A message-specific filter that specifies that only certain messages of this topic be sent.
- If various messages of a topic may pertain to various subjects, a constraint may be used to specify that only messages pertaining to a subset of the subjects be sent.

7.1.2 NotificationSource PortType: ServiceData Elements

If a Grid service implements the NotificationSource interfaces, the following table contains the **serviceData** associated with the NotificationSource interface:

```
<gsdl:serviceDataDescription name="NotificationSourceTopicList"
  type="gsdl:nsTopicList"
```

```

        minOccurs="1"
        maxOccurs="1"
xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
<wsdl:documentation>
    A list of names (NCNames) of serviceData elements contained
    directly under the root service data element, when contain
    notificationSourceTopic elements.
</wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription name="NotificationSourceTopic"
    type="gsdl:NSSourceTopic"
    minOccurs="0"
    maxOccurs="1"
xmlns:gsdl="http://schemas.gridforum.org/gridServices/">
<wsdl:documentation>
    A document describing the notification source topic, to
    which notificationSinks may be subscribed.
</wsdl:documentation>
</gsdl:serviceDataDescription>

```

7.1.3 NotificationSource PortType: Operations and Messages

7.1.3.1 NotificationSource :: SubscribeToNotificationTopic

Subscribe to a topic, such that notification message will subsequently be sent from this source to a specified sink.

Input:

- *TopicName*: The identifier of the notification source topic to subscribe to. (Note: This should probably be changed to be extensible, in a similar manner as the GridService::findServiceData QueryExpression parameter.)
- *InterestConstraint* (optional): A description of the subset of notification messages the sink is interested in receiving from this source regarding this topic. The type of this argument is specified by the constraintType attribute of the notificationSourceTopic element. If the interest constraint is not specified, then the source will choose a default constraint of its own choosing.
- *Sink*: The GSH of the notification sink to which messages will be delivered.
- *SubscriptionID* (optional): A subscriptionID that was returned by a previous subscription request. Including this argument causes re-affirmation of an existing or lapsed subscription.
- *ExpirationTime*: The time at which this subscription should timeout and notification delivery to this sink be halted, unless this notification source receives a subsequent SubscribeNotificationSink with the same SubscriptionID. Note: we need to unify the soft state mechanism for Grid service instances, notification subscriptions, and registry publish operations.

Output:

- *SubscriptionID*: A unique identifier for this subscription request to notification source. It MAY be used from the same client in subsequent re-affirmation of the subscription to extend the timeout, or to unsubscribe.

Fault(s):**7.1.3.2 NotificationSource :: UnsubscribeFromNotificationTopic**

Unsubscribe a previously subscribed notification sink.

Input:

- *SubscriptionID*: String identifier that was previously returned by *SubscribeToNotificationTopic*. Only a client that authenticates with the same subject as was used to subscribe under this subscription id may unsubscribe using this subscription id.

Output:**Fault(s):****7.2 The NotificationSink PortType**

A notification sink **portType** defines a single operation for delivering a notification message to the service instance that implements the operation.

7.2.1 NotificationSink PortType: ServiceData Elements

None.

7.2.2 NotificationSink PortType: Operations and Messages**7.2.2.1 NotificationSink :: DeliverNotification**

Deliver message to this service.

Input:

- *Message*: A **serviceData** element containing the notification message. The content of the message is dependent upon the notification source topic.

Output:

The service does not reply to this request.

Fault(s):**7.3 Integration With Notification Intermediaries**

While the *NotificationSource* and *NotificationSink* define how notification messaging is performed between two parties, these same **portTypes** can be used in various combinations to allow for third-party services to intermediate the notification process.

For example, an intermediary notification service may implement the *NotificationSink portType* in order to receive notification messages from some other sources, as well as the *NotificationSource portType* to send notifications to other subscribing sinks. The intermediary may simply forward the notification messages on to subscribers, or it may transform them in various ways making topics available to subscribers that are different than those of the original notification source. Intermediary notification sources are generally characterized by the fact that their notification topics have subjects referring to other service instances, rather than to themselves.

Intermediary notification services may be used for a variety of purposes, including:

- To provide for a notification source service that has a lifetime that is independent from the notification source service that originally generated the message.
- To filter, modify, or aggregate notification messages from other sources.
- To represent third party messaging services, which may transport notification messages with different delivery protocols, semantics, and/or qualities of service.

The third purpose, that of integrating messaging service products, deserves further explanation. Such messaging service products can be exploited in this framework by:

1. Defining an intermediary messaging service (what we call the “messaging service instance”) that implements both the NotificationSource and NotificationSink portTypes, as well as possibly other portTypes for managing the behavior of the messaging service product.
2. This intermediary messaging service instance can then subscribe to various notification source topics in various services. Note that the client issuing the subscription request need not be the same Grid service instance as the notification sink designated in the subscription request. This property allows for clients to stitch together notification message paths, without being directly in those paths.
3. The third-party messaging service instance can advertise various notification topics for which it produces notification messages, relating to any incoming notification messages it receives via its sink interface. For example, for any notification message that it receives through its sink interface, it may resend it to subscribers with a particular quality of service. The intermediary messaging service instance may have its own efficient, scalable broadcast network, thus allowing the incoming message to be efficiently broadcast to a large number of subscribing sinks. Or it may guarantee delivery of the notification message for up to 24 hours, even in the face of various failures. Or it may distribute incoming notification messages to subscribers in a round-robin fashion, rather than sending all notification messages to all subscribers. The possible behaviors that the intermediary messaging service instance can introduce to the notification message delivery are limitless.
4. The intermediary messaging service instance, the originating notification source service, and the final notification sink service may all implement a particular network protocol binding to optimize the transmission of the notification messages. For example, if the intermediary messaging service instance represents a particular message service product with its own custom protocol, implementing that protocol as a Grid service network protocol binding allows the integration of this product and its protocols, without requiring different interfaces or models to be imposed on the sources and sinks.

8 The Factory PortType

From a programming model perspective, a *factory* is an abstract concept or pattern. A factory is used by a client to create an instance of a Grid service. A client invokes a create operation on a factory and receives as response a GSR for the newly created service. This specification defines one approach to realizing the factory pattern as a Grid service. OGSA uses a document-centric approach to define the operations of the basic factory. Service providers can, if they wish, define their own factories with specifically typed operation signatures.

In OGSA terms, a factory is a Grid service that **MUST** implement the Factory **portType**, which provides a standard WSDL operation for creation of Grid service instances. A factory **MAY** of

course also implement other **portTypes** (in addition to the required GridService **portType**), such as:

- Registry (Section 9), which allows clients to inquire of the factory as to what Grid service instances created by the factory are in existence.

Upon creation by a factory, the Grid service instance **MUST** be registered with, and receive a GSH from, a home handleMap service (see Section 6). 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.

8.1 Factory PortType: ServiceData Elements

The following contains the **serviceDataDescription** elements associated with the Factory **portType**:

```
<gsdl:serviceDataDescription
name="GridServiceFactoryServiceTypeCreated"
  type="qname"
  minOccurs="1"
  maxOccurs="unbounded"
  <wsdl:documentation>
    QName to a serviceType created by this Factory.
  </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
name="GridServiceFactoryCreationInputTypes"
  type="qname"
  minOccurs="0"
  maxOccurs="unbounded"
  <wsdl:documentation>
    A qname of an XML type supported by this Factory for the
    ServiceParameters argument of the CreateService operation.
    Note: there is a consideration to have this as a property of the
    Factory ServiceType, using the extensibility element..
  </wsdl:documentation>
</gsdl:serviceDataDescription>
```

8.2 Factory PortType: Operations and Messages

8.2.1 Factory :: CreateService

Create a new Grid service instance. Note that to support soft state lifetime management (Section 4.7), a client may specify an initial termination time, within a window of earliest and latest acceptable initial termination times. The factory selects an initial termination time within this window, and returns this to the client as part of its response to the creation request. Additionally, the factory returns the maximum lifetime extension that clients can subsequently request of this new Grid service instance. Alternatively, the Grid service creation request may fail if the requested termination time is not acceptable to the factory.

Input

- *EarliestTerminationTime* (optional): The earliest termination time of the Grid service instance that is acceptable to the client.

- *LatestTerminationTime* (optional): The latest termination time of the Grid service instance that is acceptable to the client. This value must be \geq *EarliestTerminationTime*.
- *ServiceParameters* (optional): An XML document that is specific to the factory and the services that it creates.

Output

- *Reference*: A Grid service Reference to the newly created Grid service instance. (Note: The Grid service Handle is carried as part of the reference.)
- *ServiceTimestamp*: The time at which the Grid service was created.
- *CurrentTerminationTime*: The Grid service's currently planned termination time.
- *MaximumExtension*: The maximum extension that the Grid service will currently allow a client to request of its termination time.
- *ExtensibilityOutput* (optional): An XML extensibility element that is specific to the factory and the services that it creates.

Fault(s):

9 Registry

A *registry* is a Grid service that maintains a collection of Grid Service Handles, with policies associated with that collection. Authorized clients may query the registry to discover what services are available and the properties of the services and policies it maintains, and (if supported by the registry) publish and unpublish registry entries.

A Registry implements the *Registry portType*, in order to allow clients to register and unregister registry contents. Because a registry is a Grid service, it must also implement the *GridService portType*. The *findServiceData* (see Section 5.2.1) operation provides rich query interface against the contents of the registry. In addition, a registry's *findServiceData* operation SHOULD support the XPath query language, and MAY support other query languages. A registry SHOULD implement the *NotificationSource portType* (Section 7), in order to support notification of registry existence and changes in registry contents.

A Grid service MAY be a member of any number of registries, and for any portion of the service's lifetime.

9.1 WS-Inspection Document

The registry makes available a WS-Inspection document [1] to aid in discovery of the services in that registry. This document contains information about any Grid service that has been registered with the registry.

This WS-Inspection document can be retrieved by a client in two ways:

1. An http (or https) GET using the URL “<GSH>/inspection.wsil” (that is, the registry service's GSH with an additional “/inspection.wsil” suffix) MUST respond with the registry's WS-Inspection document.
2. The registry service has operations (via the *findServiceData* interface) for retrieving and querying the WS-Inspection document.

The registry's WS-Inspection document MAY have the following properties:

1. WS-Inspection WSDL service description elements that refer other Grid services (including other registries), where the location values are the GSHs of the services with a .gsr suffix.
2. WS-Inspection link elements that refer to registry services, where the location values are the GSHs of services with a “/inspection.wsil“ suffix.
3. any other valid WS-Inspection element.

9.2 The Registry portType

The *Registry portType* allows clients to register and unregister registry contents.

9.2.1 Registry PortType: ServiceData Elements

The following contains the **serviceDataDescription** elements associated with the *Registry portType*:

```
<gsdl:serviceDataDescription name="GridServiceRegistryWSInspection"
  type="wsil:inspection"
  minOccurs="1"
  maxOccurs="1"
  xmlns:wsil="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  <wsdl:documentation>
    A WS-Inspection document containing all of the Grid services in
    this registry.
  </wsdl:documentation>
</gsdl:serviceDataDescription>
```

9.2.2 Registry PortType: Operations and Messages

9.2.2.1 Registry :: RegisterService

Add or atomically update a Grid Service Handle to the registry.

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.

Input

- *Handle*: The Grid Service Handle of the service to register.
- *Timeout*: The time at which this registration should timeout and be removed from the registry, unless the registry receives a subsequent RegisterService.
- *Abstract* (optional): A text description to include with the service element.
- *Extensibility* (optional): An XML fragment to be inserted as an extensibility element for this service. This field may be used, for example to register a UDDI service description for the service as well.

Output:

Fault(s):

It is worth noting that the registry has the freedom to interpret registrations of other Registry Grid services in one of two ways. Consider the case where Registry A receives a register operation for Registry B. Registry A can choose to treat B simply as another Grid service, generating a WS-

Inspection **service** element based on the GSH for B, or it can be treat B as a WS-Inspection *link* element.

9.2.2.2 Registry :: UnregisterService

Remove a Grid Service Handle from the registry.

Removal of a GSH MUST only be allowed by a client that authenticates with the same subject as the client that registered this handle previously.

Input:

- *Handle*: The Grid Service Handle of the service to remove.

Output:

Fault(s):

9.3 Existence and Change Notification

In order to assist in discovery of services, a registry service MAY be requested to notify other services of:

1. *Existence*: A registry can periodically notify other services of its existence.
2. *Changes*: A registry can notify other services whenever it changes.

These notifications are registered and delivered using the normal Grid service notification operations (see Section 7). Note: this needs to be specified further.

10 Other Properties of OGSA To Be Addressed

There are a variety of issues which have not yet adequately addressed by the OGSA work, but which may motivate changes to this specification, and/or the creation of new, complimentary specifications. Such issues include:

- *Reliable invocation*: Various operations defined by this specification may require reliable invocation semantics. For example, many uses of `Factory :: CreateService` may require once-and-only-once message invocation semantics. Such requirements will be considered by sub-groups focused on network protocol bindings, but may impact this specification as well if general, non-binding-specific requirements are uncovered.
- *Endpoint properties*: There are many important properties of a Grid service in addition to its interface definition (e.g. **serviceType**, **portTypes**, etc.). So-called endpoint properties, dealing with how the endpoint is secured, quality of service assertions, idempotency properties, etc., must also be discussed. These properties may be specific to particular bindings of the Grid service. Client runtimes and/or developers likely need to be able to discover these properties of a Grid service by inspecting the Grid service's description. This implies that endpoint property descriptions might be decorations on the Grid service's description, appearing at the binding, operation (under binding) and service WSDL elements.
- *Security*: The OGSA must define security mechanisms that are pluggable and discoverable by the client from the service description. This allows a service provider to choose from multiple distributed security architectures supported by multiple different vendors and plug it into the infrastructure supporting their Grid services. Architectural requirements for authentication, authorization etc. are the purview of sub-working groups of the overall OGSA standardization effort. This effort is sufficiently important to the

architecture, that one or more security-discipline focused sub-working groups will be constituted to define detailed security requirements for the architecture. If the work of this sub-team implies changes to the overall OGSA, those changes will be reflected in subsequent versions of this document.

- *Manageability*: The way resources are exposed in OGSA, including how they are instrumented and managed, is a major aspect of a Grid. How manageability serviceType and portTypes are defined, the role of various manageability standards, etc. is the responsibility of a manageability sub-working group.
- *Concurrency*: In any system where shared resources are made available to clients, the issue of concurrency control must be addressed. Grid services may be stateful instances made available to and accessed by multiple clients on multiple concurrent threads of execution. It is the responsibility of the service implementation or its hosting execution environment to provide the appropriate concurrency control to govern update access to the service. It is not the responsibility of the client or user of the service to provide or ensure serialized access to the state of the target Grid service.

The Grid service implementation must ensure, at a minimum, that updates to the state of a given Grid service instance cannot occur on concurrent threads of execution. Therefore, the Grid service implementation must ensure that updates to the service state made within the scope of a given operation request are isolated from those made within the scope of a different operation request on the same service instance. This requirement can be characterized as basic thread safety through the service implementation. In other words, the state of a given service instance must be stable with respect to a given operation request executing in the service.

Above and beyond the base requirement for service side concurrency control at the operation request level, the Grid service specification does not either require nor prevent the use of a traditional transactional, two-phase committed, unit of work around one or more serially executed operations of a Grid service. If the Grid service implementation supports the use of a transactional unit of recovery, and the client supports the ability to provide demarcation of the transaction, and the underlying web services binding used to support the interaction with the service is capable of transactional context propagation from the client to the service, then it is expected that the well known ACID properties of a transactional unit of work will be applied to govern the updates to the state of the grid service. Further, it is expected that the service implementation or its hosting environment will apply the appropriate level of transaction-scoped concurrency control and that the client will not be responsible for providing this concurrency control.

11 Change Log

11.1 Draft 1 (2/15/2002) → Draft 2 (6/13/2002)

- Improved introduction to Section 4, and reordered the subsections to make it flow better.
- Added Section 4.2, containing an explanation of service description vs service instance.
- Added/rewrote service data section (4.3) including: cleaned up **serviceData** container; moved lifetime declarations out to an extensibility element that can be included on any XML element; introduced schema to be able include service data declarations into the WSDL service description.

- Changed tables containing service data declarations to use correct XML elements that conform to the new **serviceDataDescription** element defined in Section 4.3.2.
- Moved description of **instanceOf** to be part of the WSDL GSR description (4.6.2.1), since it is a sub-element of the WSDL **service** element, which is part of the WSDL GSR.
- Removed old Section 5, “How portType Definitions are Presented”. This was subsumed by the rewrite of Section 4, including the new service data specification.
- Removed all primary key material, including old Section 10, and references to it from the Factory discussion.
- Simplified the schema for serviceType.
- Added Section 11, Change Log.

12 Acknowledgements

We are grateful to numerous colleagues for discussions on the topics covered in this document, in particular (in alphabetical order, with apologies to anybody we've missed) Malcolm Atkinson, Brian Carpenter, Francisco Curbera, Andrew Grimshaw, Marty Humphrey, Keith Jackson, Bill Johnston, Kate Keahey, Gregor von Laszewski, Lee Liming, Miron Livny, Norman Paton, Jean-Pierre Prost, Thomas Sandholm, Sanjiva Weerawarana, and Von Welch.

This work was supported in part by IBM and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 and DE-AC03-76SF0098; by the National Science Foundation; and by the NASA Information Power Grid project.

13 References

1. Brittenham, P. An Overview of the Web Services Inspection Language. 2001, <http://www.ibm.com/developerworks/webservices/library/ws-wslover>.
2. Foster, I. and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 259-278.
3. Foster, I. and Kesselman, C. (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
4. Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002, www.globus.org/research/papers/ogsa.pdf.
5. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15 (3). 200-222. 2001. www.globus.org/research/papers/anatomy.pdf.
6. Graham, S., Simeonov, S., Boubez, T., Daniels, G., Davis, D., Nakamura, Y. and Neyama, R. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Sams, 2001.
7. Mukhi, N. Web Service Invocation Sans SOAP. 2001, <http://www.ibm.com/developerworks/library/ws-wsif.html>.

14 Contact Information

Steven Tuecke
Distributed Systems Laboratory

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
Phone: 630-252-8711
Email: tuecke@mcs.anl.gov

Karl Czajkowski
University of Southern California, Information Sciences Institute
Email: karlcz@isi.edu

Ian Foster
Argonne National Laboratory & University of Chicago
Email: foster@mcs.anl.gov

Jeffrey Frey
IBM
Poughkeepsie, NY 12601
Email: jafrey@us.ibm.com

Steve Graham
IBM
4400 Silicon Drive
Research Triangle Park, NC, 27713
Email: sggraham@us.ibm.com

Carl Kesselman
University of Southern California, Information Sciences Institute
Email: carl@isi.edu

15 XML and WSDL Specifications

This Section will contain the full WSDL types, message, and portType for each of the operations described in this document. Watch this space.