Grid Service Specification

Draft 3 (7/17/2002)

Steven Tuecke[1] Karl Czajkowski[3] Ian Foster[1,2]
Jeffrey Frey[4] Steve Graham[5] Carl Kesselman[3]

[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

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 registration 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 (of course, subject to policy and compatible protocol bindings).

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 Notational Conventions

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].

This specification uses namespace prefixes throughout; they are listed in Table 1. Note that the choice of any namespace prefix is arbitrary and not semantically significant.

Table 1: Prefixes and Namespaces used in this specification.

Prefix	Namespace
gsdl	"http://www.gridforum.org/namespaces/2002/07/gridServices
mdt	"http://www.gridforum.org/namespaces/2002/07/measuredDateTime"
wsdl	"http://www.w3.org/2002/07/wsdl"
wsil	"http://schemas.xmlsoap.org/ws/2001/10/inspection/"
http	"http://www.w3.org/2002/06/wsdl/http"
xsd	"http://www.w3.org/2001/XMLSchema"
xsi	"http://www.w3.org/2001/XMLSchema-instance"

Namespace names of the general form "http://example.org/..." and "http://example.com/..." represent application or context-dependent URIs [cite: RFC2396].

In this document we use **bold** font face to emphasize WSDL defined elements and properties, as well as elements and properties of WSDL extensions defined in this document.

The following abbreviations and terms are used in this document:

- *GSH*: Grid Service Handle, as defined in Section 4.7.
- *GSR*: Grid Service Reference, as defined in Section 4.7.
- *SDE*: Service Data Element, as defined in Section 4.3.
- *SDD*: Service Data Description, as defined in Section 4.3.
- The terms *Web services*, *XML*, *SOAP*, *WSDL*, and *WS-Inspection* are as defined in [4].

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.

Unresolved issues with the specification are interspersed in appropriate locations through this specification, are highlighted in yellow, and begin with “Issue N:”, where N is the GGF OGSI working group bugzilla database bug number for this issue. This database is located at http://www.gridforum.org/ogsi-wg/bugzilla.

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 **portType**s within the context of an aggregating **serviceType**. Grid service factories (Section 9) 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 the name of its **serviceType** and the collection of WSDL **portType**s that it implements.

Grid service instances are made accessible to (potentially remote) client applications through the use of a Grid Service Handle (Section 4.7.2) and a Grid Service Reference (Section 4.7.1). 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 OGSA. 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 implementation inheritance, service mobility, 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.

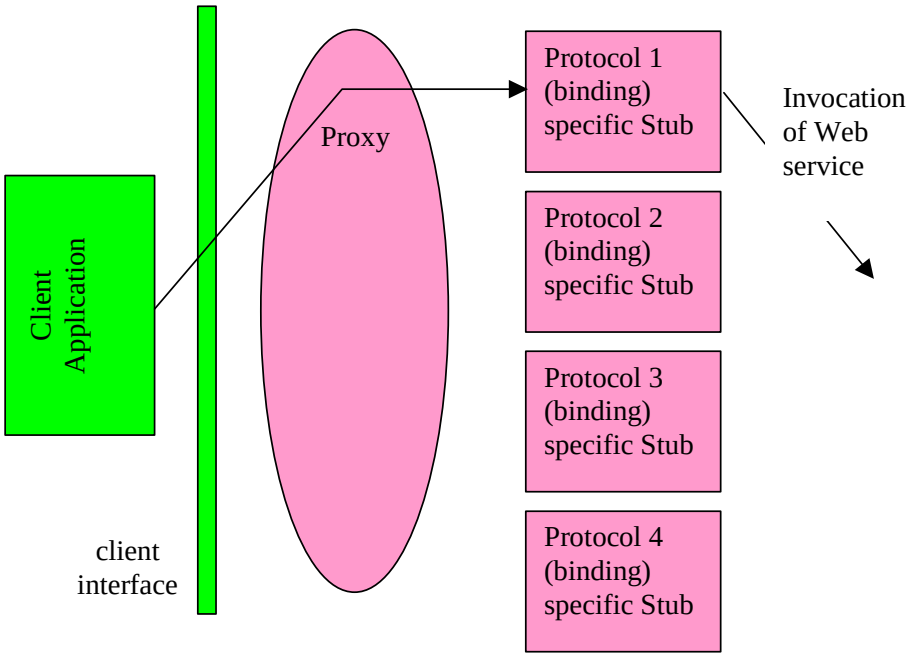

Figure 1: A possible client-side runtime architecture

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.

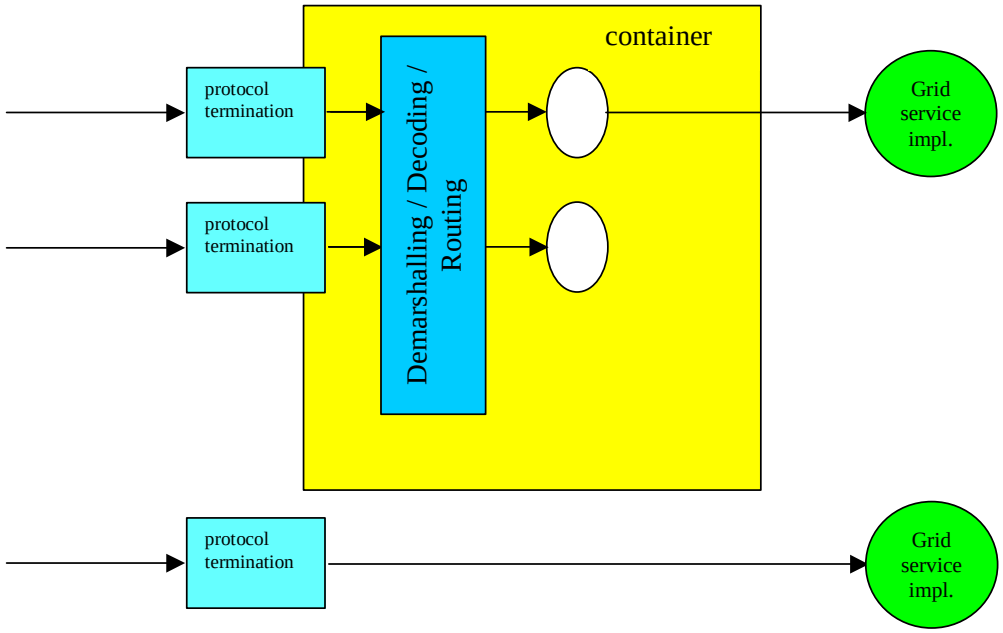

Figure 2: Two alternative approaches to the implementation of argument demarshalling functions in a Grid Service hosting environment

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 lower oval), the entire behavior of a Grid service is completely encapsulated within the component. In other cases (the upper 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:

- 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 how time is modeled in OGSA;
- Defining *service data*, which provides a standard way for representing and querying meta-data and state data from a service instance;
- Extending the **serviceType** element introduced in WSDL v1.2 [8] to include **serviceType** reuse through aggregation;
- 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 various **portType**s, starting with the GridService **portType** that must be supported by any Grid service, and then proceeding to HandleResolver, 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 2and 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 2: Proposed WSDL conventions and extensions introduced by OGSA

Concept	WSDL Element	Brief Description	See Section

Service state data and meta-data	**serviceData**	**serviceData** elements represent properties of the service's state that may be externally queried. Some **serviceData** elements may appear as extensions of the service's description, as part of the service's type.	4.3
Service data description	**serviceData Description**	Formal descriptions of serviceData elements. These descriptions appear as extensions of the Grid service's WSDL.	4.4.2
Interface Naming	convention on **portType** name	Naming conventions and immutability of **portType**, and **serviceType** names.	4.2
serviceType reuse	Extends **serviceType**	Derivation of a new **serviceType** from existing **serviceType**(s)	4.5
Grid Service Reference	N/A	Mechanism to convey capabilities of a service to a client. *Can* be a WSDL document.	4.7.1
Grid Service Handle	N/A	Conventional use of URI to act as unique identifier of a Grid service instance.	4.7.2

This draft is based on extensions to the WSDL language proposed by the W3C Web Services Description Working Group [8]. In particular, we are relying upon the following new constructs proposed for WSDL 1.2 (draft):

- open content model (extensibility elements appearing in each WSDL element)
- **serviceType**.

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* describes how a client interacts with service instances. This description is independent of any particular instance. Within a WSDL document, the Grid service description is embodied in the **serviceType** of the instance, along with its associated **serviceType**s, **portType**s, **serviceDataDescription**s, **messages**, and **types** definitions.
- 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 4.7.1) is a WSDL document comprising a **service** element, which carries an **implements** property that refers to a **serviceType** defined by the service description of that instance.

A service description is primarily used for two purposes. First, as a description of a service interface, 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 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 non-normative fashion) semantics. Interface syntax is, of course, described by the collection of one or more **portType**s referred to by the **serviceType**. Note that the **portType**s may come from different namespaces.

Semantics may be inferred through the names assigned to the **portType** and **serviceType** elements. For example, when defining a Grid service, one defines zero or more uniquely named **portType**s, and then collects a set of **portType**s defined from a variety of sources into a uniquely named **serviceType**. 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 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 Modeling Time in OGSA

Throughout this specification there is the need to represent time that is meaningful to multiple parties in the distributed Grid. For example: information may be tagged by a producer with timestamps in order to convey that information's useful lifetime to consumers; clients need to negotiate service instance and registration lifetimes with services; and multiple services may need a common understanding of time in order for clients to be able to manage their simultaneous use and interaction.

The GMT global time standard is assumed for Grid services, allowing operations to refer unambiguously to absolute times. However, assuming the GMT time standard to represent time does *not* imply any particular level of clock synchronization between clients and services in the Grid. In fact, no specific accuracy of synchronization is specified or expected by this specification, as this is a service-quality issue.

Given this lack of any required accuracy of synchronization, it is instead important that accuracy and resolution information about the time source be included whenever timestamps are used in OGSA. Accuracy information about a timestamp describes the maximum expected clock skew between GMT and the time source from which timestamp was taken. Resolution information about a timestamp describes the smallest unit by which the time source is updated. This combination of a GMT timestamp, along with accuracy and resolution of that timestamp, allows consumers of that timestamp to make informed decisions about the quality of that timestamp.

Issue 13: There is an emerging specification [9] that defines the XML schema for *measured timestamps*, which are timestamps derived from xsd:dateTime that also include accuracy and resolution information. Assuming that this measured timestamp specification evolves to meet the needs of timestamps for Grid services, we should update this Grid Service Specification to use those measured timestamps instead of xsd:dateTime. We should also recommend that designers of higher-level Grid services also use these same measured timestamps where appropriate.

Grid service hosting environments and clients SHOULD utilize the Network Time Protocol (NTP) or equivalent function to synchronize their clocks to the global standard GMT time. However, clients and services MUST accept and act appropriately on messages containing time values that might be out of range due to inadequate synchronization, where "appropriately" MAY include refusing the use the information associated with those time values. Furthermore, clients and services requiring global ordering or synchronization at a finer granularity than their clock accuracies or resolutions allow for MUST coordinate through the use of additional synchronization service interfaces, such as through transactions or synthesized global clocks.

4.4 Service Data Concept

In order to support discovery, introspection, and monitoring of Grid service instances, we introduce the concept of *service data*, which refers to descriptive information about a Grid service instance, including both *meta-data* (information about the service instance) and *state data* (runtime properties of the service instance).

Each Grid service instance is associated with a set of *service data elements (SDEs)*. Each SDE is represented in XML by a **serviceData** element (see Section 4.4.1). A **serviceData** element is a container that MUST contain zero or more XML elements of some XML type. We refer to each XML element as a *service data value element*, and the complete set of elements within an SDE as the *service data value*.

SDEs MAY appear in a Grid service's service description. Specifically, **serviceData** elements MAY appear as extensibility elements in the **portType** and **serviceType** elements of the service description. We refer to such an SDE as a *structural SDE*. A structural SDE declares that any Grid service instance that implements the given **portType** or **serviceType** MUST include an SDE of the same name amongst its set of SDEs. The **serviceData** element that appears in a service description also indicates initial service data value elements for the SDE. A Grid service instance MAY additionally include *non-structural* SDEs within its **serviceDataSet**. Non-structural SDEs are *not* declared within the instance's service description.

SDEs MUST additionally be made accessible to clients at runtime from the instance itself within a **serviceDataSet** element (see Section 4.4.3), and MAY change over the lifetime of the instance. The **serviceDataSet** contains the complete set of SDEs (structural and non-structural) associated with the Grid service instance. A Grid service's **serviceDataSet** MAY be accessible to clients in two ways. A Grid service instance MUST implement the **FindServiceData** operation of the GridService **portType** (see Section 6.2.1), which provides a simple, extensible, client-initiated query against the instance's **serviceDataSet**. A Grid service instance MAY additionally implement the **NotificationSource portType** that provides the **Subscribe** operation (see Section 8.1.2.1). This operation enables a client to ask the instance to notify it of subsequent changes to the instance's **serviceDataSet**. Note that the service data that is available for query by a client MAY be subject to policy restrictions. For example, some service data elements MAY not be available to some clients, and some service data value elements within a SDE MAY not be available to some clients.

The characteristics of each service data element MUST be declared using a **serviceDataDescription** element (see Section 4.4.2). Such a declaration, called a *service data description (SDD)*, specifies properties such as the name of the SDE, the XML type of the service data value elements, how many times service data value elements may occur, whether the value elements may change during the lifetime of the instance, etc. **serviceDataDescription** elements MAY appear as part of a Grid service's service description, as an extension of the **definitions** element.

Each SDD has a name that MUST be unique amongst all **serviceDataDescription** elements within its namespace. This name, when prepended with the URI of the enclosing namespace forms a qname that is globally unique. The name property of a **serviceData** element MUST correspond to the name of the SDD to which that SDE conforms.

4.4.1 serviceData

A **serviceData** element MAY appear as part of a **portType**, a **serviceType** or as part of a service instance's **serviceDataSet** (see Section 4.4.3). A **serviceData** element is a container for a collection of service data value elements.

A **serviceData** element has the following non-normative grammar:

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

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

- **name**: This attribute's value is the qname of the **serviceDataDescription** (see Section 4.4.2) element that describes this **serviceData** element. The **serviceDataDescription** may come from any namespace.
- **type**: This attribute's value is the qualified name of the XML type to which service data value elements contained in this **serviceData** element must conform. This type MUST be either the same as the **type** property of the **serviceDataDescription** element referred to by the **name** property, or a derivation of that type using XSD extension or restriction. If this property is not specified, this value MUST default to be the type specified by the **type** property of the **serviceDataDescription** element referred to by the **name** property. Note: the designer would use xsi:schemaLocation attribute to suggest a possible location for further information about the namespaces associated with the value of the **type** property.
- *Extensibility attributes*: A **serviceData** element MAY have other extensibility attributes including but not limited to:
  - *Lifetime declarations*: As defined in Section 4.4.4, **gsdl:goodFrom**, **gsdl:goodUntil**, and **gsdl:availableUntil** attributes MAY be placed on a **serviceData** element to declare the lifetime characteristics of that service data element and its value.
  - *Application-specific*: A **serviceData** element MAY have additional, application-specific extensibility attributes from any namespace.
- *Extensibility elements*: A **serviceData** element MAY have other extensibility elements including but not limited to:
  - *Service data value*: The **serviceData** extensibility element MUST include zero or more elements that conform to the XML type referred to by the **type** property – we refer to these as *service data value elements*. The number of such service data value elements MUST be greater than or equal to the value specified by the **minOccurs** property of the **serviceDataDescription** element referred to by the **name** property, and MUST be less than or equal to the value specified by the **maxOccurs** property of that same **serviceDataDescription** element. These **serviceDataDescription**

properties govern the number of service data value elements that appear in a **serviceData** element within a service instance's **serviceDataSet**, and do not apply to a **serviceData** element that appears in **portType** or **serviceType** elements.

- *Application-specific*: A **serviceData** element MAY have additional extensibility elements from any namespace.

The following is an example declaration of **serviceData** in a **portType**:

```
…
<portType name="CPU">
…
...<gsdl:serviceData name=”tns:CPUSpeed”/>

   <gsdl:serviceData name="tns:CPULoad">
     <xsd:float>0.00</xsd:float>
   </gsdl:serviceData>
…
</portType>
```

The SDDs that correspond to these **serviceData** elements appears in Section 4.4.2. Note the inclusion of initial service data value elements in the definition for the CPULoad **serviceData** element.

4.4.2 serviceDataDescription

The **serviceDataDescription** element MAY extend the **definitions** element, in order to describe service data elements that MAY appear in a Grid service instance's **serviceDataSet**.

The **serviceDataDescription** element has the following non-normative grammar:

```
<gsdl:serviceDataDescription
        name=”NCName”
        type=”qname”
        minOccurs=”nonNegativeInteger”?
        maxOccurs=(”nonNegativeInteger” | “unbounded”)?
        mutability=”constant”|”append”|”mutable”? >
   <wsdl:documentation .... />?
   <-- extensibility element --> *
</gsdl:serviceDataDescription>
```

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

- **name**: A name for this service data description, which MUST be unique amongst all **serviceDataDescription** names within the namespace in which the it is defined.
- **type**: The qualified name of the XML type that is the type of service data value elements contained in any **serviceData** element that conforms to this **serviceDataDescription**.
- **minOccurs**: The minimum number of service data value elements, each conforming to the XML type defined by the **type** property, which MUST be contained in an SDE that conforms to this SDD. If this attribute is omitted, then it defaults to 0.
- **maxOccurs**: The maximum number of service data value elements, each conforming to the XML type defined by the **type** property, which MUST be contained in an SDE that conforms to this SDD. If this attribute is omitted, then it defaults to “unbounded”.

- **mutability**: An SDE MAY appear within a **portType**, within a **serviceType**, and/or within a service instance's **serviceDataSet**. When SDEs conformant to the same **serviceDataDescription** appear in multiple of these locations, the **mutability** property declares the semantic associated with combining these SDEs. If this property is omitted, then it defaults to "mutable".

  Issue 21: In the description of each option for the SDD mutability attribute, we need to describe how SDEs that appear in the service description should be interpreted by a client that is inspecting the service description.

  - **mutability="constant"**
    If a **serviceData** element appears within a **portType**, and it defines a service data element value that is non-empty, then any **serviceData** element conformant to the same SDD appearing in a **serviceType** that includes the **portType** must define a service data element value that is empty. Further, any other **portType** aggregated by that **serviceType** MUST NOT include a **serviceData** element conformant to that SDD, unless that **serviceData** element also defines an empty service data element value.

    If a **serviceData** element appears within a **portType**, but it defines an empty service data element value, then a **serviceData** element conformant to the same SDD MAY appear in a **serviceType** that includes the **portType** (or in any other **portType** aggregated by the **serviceType**), but only in at most one of these places MAY the **serviceData** element contain a non-empty service data element value.

    If a **serviceData** element conformant to this SDD appears in a service instance's **serviceType** or any **portType** aggregated by the **serviceType**, and in one of these places a non-empty service data element value is declared, then the SDE conformant to this SDD appearing in the instance's **serviceDataSet** MUST include the same service data element value, unchanged, for its entire lifetime.

    If a **serviceData** element conformant to this SDD appears in a service instance's **serviceType** or any **portType** aggregated by the **serviceType**, but in none of these places is a non-empty service data element value declared for the SDE, then the SDE conformant to this SDD appearing in the instance's **serviceDataSet** MAY be assigned a service data element value, but once assigned, this value must remain unchanged, for the remaining lifetime of the instance.

    If a **serviceData** element conformant to the SDD does not appear within a service instance's **serviceType** nor any **portType** aggregated by the **serviceType**, then the instance MAY include a **serviceData** element conformant to the SDD within its **serviceDataSet**, but once its service data element value is set it MUST NOT be changed for the remaining lifetime of the instance.

  - **mutability="append"**
    If a **serviceData** element conformant to this SDD appears in a service instance's **serviceType** or any **portType** aggregated by the **serviceType**, then the service data element value for the SDE appearing within the instance's **serviceDataSet** MUST include a concatenation of all the service data value elements declared for each SDE declared in the **serviceType** and any **portType** aggregated by the **serviceType**. The instance MAY also include additional service data value elements to the SDE. The order of appearance of the service data value elements MUST NOT be significant.

Once a service data value element appears in an instance SDE that is contained in the instance's **serviceDataSet**, that value element MUST remain in that SDE, unchanged, for the remaining lifetime of the instance.

- **mutability="mutable"**
  If a **serviceData** element conformant to this SDD appears in a service instance's **serviceType** then the *initial* service data value for the corresponding SDE in the instance's **serviceDataSet** is exactly the one defined by the serviceData element in the serviceType (portType SDEs are ignored).

  If a **serviceData** element conformant to this SDD does not appear in a service instance's **serviceType,** but does appear in one of the **portType**s aggregated by the **serviceType**, then the *initial* service data value for the corresponding SDE in the instance's **serviceDataSet** is exactly the one defined by the serviceData element in the **portType**.

  If a **serviceData** element conformant to this SDD does not appear in a service instance's **serviceType,** but does appear in more than one of the **portType**s aggregated by the **serviceType**, then the *initial* service data value for the corresponding SDE in the instance's **serviceDataSet** is derived from the serviceData element in one of the **portType**s. The algorithm to choose which **portType** is not specified.

  At any time during the lifetime of the instance, the service data element value of the SDE corresponding to this SDD MAY change.

Some example SDDs:

```
<gsdl:serviceDataDescription
        name="CPUSpeed"
        type="xsd:float"
        minOccurs="1"
        maxOccurs="1"
        mutability="mutable">
   <wsdl:documentation>
      Example definition of a measurement of CPU speed
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="CPULoad"
        type="xsd:float"
        minOccurs="1"
        maxOccurs="1"
        mutability="mutable">
   <wsdl:documentation>
      Example definition of a measurement of CPU load
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

4.4.3 serviceDataSet and Instance Service Data

A set of **serviceData** elements MAY be aggregated into a **serviceDataSet** element, which has the following non-normative grammar:

```
<gsdl:serviceDataSet>
   <gsdl:serviceData …> *
      …
   </gsdl:serviceData>
</gsdl:serviceDataSet>
```

Each Grid service instance MUST make available exactly one **serviceDataSet** element, against which it evaluates all FindServiceData (Section 6.2.1) and Subscribe (Section 8.1.2.1) operation requests from its clients.

An instance's **serviceDataSet** element MUST include the **serviceData** elements declared in the instance's **serviceType** and all the **portTypes** that the **serviceType** aggregates (structural SDEs). The **serviceDataSet** MAY also include additional **serviceData** elements (non-structural SDEs).

However, this specification does *not* dictate how the service data set and its elements are represented internally within the runtime of a Grid service instance. The GridService **portType** (Section 6) provides a **FindServiceData** operation that allows clients to issue queries against this collection of *logical* XML elements, and the **NotificationSource portType** (Section 8.1) provides a **Subscribe** operation that allows clients to subscribe to changes to this same *logical* collection. We use the term *logical* since there is no requirement for the Grid service implementation to actually maintain the service data set and its elements 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, and as necessary to generate notification messages as a result of the **Subscribe** operation.

4.4.4 XML Element Lifetime Declaration Properties

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 and its value, 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.

The three lifetime declaration properties are:

- **gsdl:goodFrom="mdt:measuredDateTime"**: Declares the time from which the content of the element is said to be valid. This is typically the time at which the value was created.
- **gsdl:goodUntil="mdt:measuredDateTime"**: Declares the time until which the content of the element is said to be valid. This property MUST be greater than or equal to the **goodFrom** time.
- **gsdl:availableUntil="mdt:measuredDateTime"**: 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 property MUST be greater than or equal to the **goodFrom** time.

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

```
<wsdl:definitions
   targetNamespace=”http://example.com/ns”
   xmlns:n1=”http://example.com/ns”
   … >

   <wsdl:types>
      <xsd:schema …
         targetNamespace=http://example.com/ns
…
>
         <xsd:complexType name="MyType">
            <xsd:sequence>
               <xsd:element name="e2" type="xsd:string" minOccurs="1"/>
               <xsd:element name="e3" type="xsd:string" minOccurs="1"/>
               <xsd:element name="e4" type="xsd:string" minOccurs="0"/>
            </xsd:sequence>
         <anyAttribute namespace="##any"/>
         </xsd:complexType>
      </xsd:schema>
   </wsdl:types>
   …
   <gsdl:serviceDataDescription
            name=”MySDE”
            type=”n1:MyType”
            minOccurs=”1”
            maxOccurs=”1”
            mutability=”mutable”/>
   …
   <wsdl:portType name=”MyPortType”>
      …
      <gsdl:serviceData name=”n1:MySDE” />
      …
   <wsdl:portType>
   …
</wsdl:definitions>
```

And within the service instance’s **serviceDataSet**:

```
<gsdl:serviceData
      name=”n1:MySDE”
      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 xsi:type=”n1:MyType”>
      <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 service data value contained in the **serviceData** element's extensibility element, which in this example is the element n1:e1, which conforms to the type "n1:MyType". 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 service data value is likely to no longer be valid, and therefore the client should query the service again for the SDE with the same **name** (n1:MySDE) to obtain a newer value of n1:e1.

The **availableUntil** does *not* refer to the service data 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 n1:MySDE, 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 named SDE for 1 day, but after that time the service may no longer have an SDE with the name n1:MySDE.

It is sometimes not sufficient for lifetime information of a SDE to refer only to the complete service data value. Rather, the value of a SDE may contain sub-elements with different lifetimes than those declared in the **serviceData** element. 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 on sub-elements override the default values specified on parent elements. There are no constraints on the values of these attributes in the sub-elements, relative to those specified in the parent elements, except that the ordering constraints between the effective goodFrom, goodUntil and availableUntil values for any element must be maintained.

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. This example, of course, assumes that the MyType schema allows for n1:e4 to be an optional element, and thus be omitted from n1:e1.

It is RECOMMENDED that the XML schema for elements that are intended to be service data values allow all elements within their schema to be extended with these lifetime declaration properties, in order to allow for fine-grained lifetime declarations. However, since the **serviceData** element supports extensible properties, service data values that lack property 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, a processor of these properties MUST NOT assume that they necessarily reflect temporal aspects of that definitive source, unless otherwise specified (for example, in the semantic specification of a particular service data element). 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.

Issue 22: Since the element lifetime attribute are not required, we need to define the semantics when they are not specified. The consensus seems to be that absence of these properties means that "don't know".

Issue 23: Should there be a special value of "forever" that is allowed on the goodUntil and availableUntil, to specify the expectation by the creator that these elements will not change. One could instead use values far in the future to represent this expectation. The problem is, how far in the future to go? A specially encoded value can be efficiently used to express the designer's intent. Another alternative is to designate the maximum representable dateTime as meaning "forever"..

4.5 ServiceType Inheritance

The purpose of the **serviceType** element introduced into WSDL 1.2 [8] is to aggregate **portType** elements into a named set. The following is the syntax proposed for **serviceType** by the W3C Web Services Description Working Group WSDL 1.2:

```
<wsdl:serviceType name="ncname"> *
   <wsdl:portType name="qname"/> +
</wsdl:serviceType>
```

A **service** declares that it implements a **serviceType** by the following syntax:

```
<wsdl:service name="ncname" serviceType="qname">
```

Note that a **service** MUST implement exactly one **serviceType**.

Although this is an improvement over WSDL 1.1, it remains inadequate for our needs in OGSA. For purposes of reuse of definition, we propose a limited form of **serviceType** inheritance, extending the serviceType element in WSDL 1.2.

4.5.1 Requirements for ServiceType Inheritance

Why do we need to extend **serviceType**? Fundamentally it boils down to two concepts:

1. ease of expression to define specializations of the interfaces that are standardized by more primitive **serviceTypes**, and
2. ease of discovery of Grid services that implement a target **serviceType** or a specialization of that target **serviceType**.

Consider a simple situation. A designer has declared a **serviceType** to define the interface to a specialized form of registry (See section 10). This **serviceType** aggregates the GridService portType (See section 6), the Registration **portType** and a domain-specific **portType** as shown in the following XML:

```
<wsdl:portType name="specializedRegistry">
   <operation name="specialFind">
…
```

```
</wsdl:portType>

<wsdl:serviceType name="specializedRegistry">
   <wsdl:portType name="gsdl:gridService"/>
   <wsdl:portType name="gsdl:Registration"/>
   <wsdl:portType name="gsdl:specializedRegistry"/>
</wsdl:serviceType>
```

So far, all of this can be accomplished with WSDL 1.2, however, another designer may want to further specialize this interface by adding notification behavior (See section 8), defining a new **serviceType** named “notifyingSpecializedRegistry”. The designer of this new **serviceType** has two choices:

1. Duplicate the description of specializedRegistry by (for example) copy/paste of the existing WSDL into the notifyingSpecializedRegistry **serviceType**:

```
<wsdl:serviceType name="notifyingSpecializedRegistry">
   <wsdl:documentation>
      Definition of the notifyingSpecializedRegistry by copy/paste
      from the specializedRegistry serviceType and then adding
      the portType reference to gsdl:notificationSource.
   </wsdl:documentation>

   <wsdl:portType name="gsdl:gridService"/>
   <wsdl:portType name="gsdl:Registration"/>
   <wsdl:portType name="gsdl:specializedRegistry"/>

   <wsdl:portType name=”gsdl:notificationSource”/>
</wsdl:serviceType>
```

or

2. use **serviceType** inheritance proposal here to clearly identify that notifyingSpecializedRegistry is a specialization of specializedRegistry.

```
<wsdl:serviceType name="notifyingSpecializedRegistry">
   <wsdl:documentation>
      Definition of the notifyingSpecializedRegistry by inheriting
      from the specializedRegistry serviceType and then adding
      the portType reference to gsdl:notificationSource.
   </wsdl:documentation>

   <gsdl:serviceType name="tns:specializedRegistry"/>
   <wsdl:portType name=”gsdl:notificationSource”/>
</wsdl:serviceType>
```

The first approach has several downsides:

1. the list of **portType** elements for the base behavior is now duplicated in two places (specializedRegistry and notifyingSpecializedRegistry).
2. there is no formal mechanism that associates Grid service instances that implement notifyingSpecializedRegistry as also being an implementation of specializedRegistry, this situation hinders discovery.

The second approach clearly indicates that notifyingSpecializedRegistry is a **serviceType** derived from specializedRegistry and that therefore a Grid service that implements this **serviceType** also MAY be considered an implementation of specializedRegistry.

4.5.2 Syntax and Interpretation

The following is the non-normative grammar to represent **serviceType** interface inheritance:

```
<wsdl:serviceType name="ncname"> *
   <gsdl:serviceType name=”qname”/>*
   <wsdl:portType name="qname"/> +
</wsdl:serviceType>
```

A **wsdl:serviceType** element MAY be extended with zero or more **gsdl:serviceType** sub-elements, each of which refers to another **wsdl:serviceType** by its qualified name. This approach leverages the open content model proposed for WSDL 1.2.

When a gsdl:**serviceType** element appears as a child of a wsdl:**serviceType** element, we say that the latter *references* and is *derived* from the wsdl:**serviceType** named in the gsdl:**serviceType**.

A **serviceType** MAY be derived from zero or more other **serviceTypes**. The *derivation* of a **serviceType** is the union of the set of **serviceTypes** referenced by a **serviceType** and the derivations of each referenced **serviceType**.

Any **serviceType** MUST appear at most once in the derivation of a **serviceType**. Consider the following case:

- **serviceType** X references **serviceType** Y and **serviceType** Z
- **serviceType** A references **serviceType** X and **serviceType** Y.

In this case, **serviceType** Y MUST appear in the derivation of **serviceType** A only once.

A **serviceType** MUST NOT derive from itself.

From the perspective of aggregating **portTypes**, the set of **portTypes** *represented* by a **serviceType** is equivalent to the union of the set of **portTypes** declared directly in that **serviceType** and the set of **portTypes** represented by all of that **serviceType**'s referenced **serviceType**s. The set of **portTypes** represented by a **serviceType** defines the interface (set of operations) that must be supported by any service that implements that **serviceType**.

Any **portType** MUST NOT appear more than once in the set of **portType**s represented by a **serviceType**. Consider the following case:

- **serviceType** B contains **portType** “pt1” in its set of **portType**s
- **serviceType** A references **portType** pt1 directly
- **serviceType A** derives from **serviceType** B.

The set of **portType**s represented by **serviceType** A does not contain two references to **portType** pt1 (the one declared directly and the one inherited from **serviceType** B), but rather contains only one reference to **portType** pt1.

If a **service** element declares that it implements a **serviceType,** then for each **portType** represented in the **serviceType**, there must be one **port** element child of the **service** element that references a **binding** for that **portType**.

For the purposes of discovery, a **service** that implements a **serviceType** MUST be considered an implementation of that **serviceType** and all of its referenced **serviceType**s. If that is not the intended interpretation, then the designer MUST NOT use the service inheritance mechanism to declare his/her **serviceType**, and instead MUST revert to a more primitive mechanism such as copy/paste to define his/her **serviceType**.

With respect to service data (See Section 4.3), the set of **serviceData** elements declared by a **serviceType** is the union of the **serviceData** elements declared directly in the **serviceType** and the **serviceData** elements declared in each of the **serviceType**'s referenced **portType**s and the **serviceData** elements declared in each of the **serviceType**'s referenced **serviceType**s. The means by which service data element values are combined is dictated by the mutability property of the service data description associated with each **serviceData** element (See Section 4.4.2).

4.6 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, requirements, and recommendations that are used to address it.

4.6.1 The Change Management Problem

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

1. Its *interface specification*. Syntactically, a Grid service's interface is defined by its service description, comprising a **serviceType** and its associated **serviceType**s, **portType**s, **operation**s, **serviceDataDescription**s, **message**s, and **types**. Semantically, the interface typically is defined in specification documents such as this one, though it may also be defined through other formal approaches.
2. The *implementation of the interface*. While expected implementation semantics may be implied from 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 specification. Nonetheless, such an implementation semantics may come to be relied upon by clients of that service interface, whether by accident or by design.

In order for a client to be able to reliably 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 or service data description to the interface, or the addition of optional extensions 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 might not be considered 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.

4.6.2 Naming Conventions for Grid Service Descriptions

In WSDL, each **portType** is globally and uniquely named via its qualified name—that is, the combination of the namespace containing the **portType** definition, and the locally unique name of the **portType** element within that namespace. Similarly, each **serviceType** is globally and uniquely named via its qualified name. In OGSA, our concern with change management leads us to require that all elements of a Grid service description 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.

Issue 24: Several people have commented on the need to loosen the immutability statement to say that a service description must be immutable once the service designer no longer has control of all possible clients of instances that use that description. The wording in this specification supports this interpretation because the MUST, MAY, etc. indicate what assumptions a compliant client or service implementation may make. A designer with full control of the client and hosting environment implementations can propagate changes without relying on features in the specification. However, this needs to be expressed more clearly in the text.

Issue 25: Some people have expressed the concern that the statement of immutability of service descriptions is too strict for small, backward compatible additions to a portType or serviceType. One suggestion is to add some notion of a version number as a constant service data element.

4.7 Naming Grid Service Instances: Handles and References

Each Grid service instance is globally, uniquely, and for all time named by one or more *Grid Service Handles (GSH)*. However, a GSH is just a minimal name in the form of a URI, and does not carry enough information to allow a client to communicate directly with the service instance. Instead, a GSH must be *resolved* 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.

Like any URI, a GSH consists of a *scheme*, followed by a string containing information that is specific to the scheme. The scheme indicates how one interprets the scheme-specific data to resolve the GSH into a GSR, within the bounds of the requirements defined below. A client MAY choose to implement a set of GSH resolution protocols itself, or it MAY choose to outsource all resolution, for example, to a pre-configured service that implements the **HandleResolver portType** (see Section 7).

The 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 resolve the GSH into a new, valid GSR.

4.7.1 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 typically 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 programmatic 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. However, a GSR may be localized to a given client context or hosting environment and the scope of portability for a GSR is determined by the binding mechanism(s) it supports.

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 be encoded as 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, SHOULD 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 MAY be independent of the lifecycle of the associated Grid service instance. A GSR is valid when the associated Grid service instance exists and can be accessed through use of the Grid Service Reference, but validity MAY only be detected by the client attempting to utilize the GSR. 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 SHOULD 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 MAY obtain a new GSR using the Grid Service Handle of the associated Grid service instance, as defined in Section 4.7.2. It is RECOMMENDED that the Grid Service Handle be contained within each binding-specific implementation of the Grid Service Reference. A client encountering an invalid GSR would otherwise be unable to acquire a new, valid GSR unless he

cached the GSH himself or repeated a discovery operation to reacquire the GSH without being able to interact with the service instance.

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 hint 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.

4.7.1.1 WSDL Encoding of a GSR

It is RECOMMENDED that a WSDL document that encodes a GSR 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.

4.7.2 Grid Service Handle (GSH)

A GSH MUST be a valid URI (cite: RFC 2396). The URI scheme defines the protocol for resolving the GSH to a GSR.

Issue 26: Who is in control of handle namespaces? Who decides what handles are given to a particular instance? Does a service instance necessarily know about all handles that refer to that instance? This has implications on: how one can determine if a handle refers to a particular instance; whether the GridServiceHandles SDE is authoritative or just a hint; whether there can be restrictions on the number of handles of a URI scheme that refer to a particular instance.

The following are properties of a GSH, resolver, Grid service instance, and GSR:

1. A GSH MUST globally, uniquely and for all time refer the same Grid service instance. The same GSH MUST NOT ever refer to more than one Grid service instance, whether or not they exist simultaneously.
2. A Grid service instance MUST have at least one GSH.
3. A Grid service instance MAY have multiple GSHs that use the same URI scheme, and MAY have multiple GSHs that use different URI schemes. Issue 27: Should we restrict handles to only allow a single GSH within a given URI scheme to a particular instance? An advantage to this is that given two GSH's with the same URI scheme, you would be able to test for inequality – that is, if they are syntactically different, you would know that they refer to different instances.
4. The **GridServiceHandles** service data element (Section 6.1) of each Grid service instance MUST contain only GSHs that refer to that instance. If two GSHs are contained in a Grid service instance's **GridServiceHandles** SDE, then they MUST both refer to that Grid service instance.
5. Since a Grid service instance MAY have multiple GSRs that refer to that instance, multiple resolutions of the same GSH MAY result in different GSRs. For example, a

resolver MAY return different GSRs for the same GSH at different times, and it MAY return different GSRs to different clients that are resolving the same GSH.

An *untrusted resolver protocol* is one for which the client cannot trust that the GSR returned as a result of speaking that protocol is valid for the requested GSH. An example of an untrusted resolver protocol is the one associated with the http GSH scheme, as described in Section 4.7.2.1. In this situation, the party performing the resolution SHOULD assume that the resolution that results from the resolver protocol MAY be accidentally or maliciously wrong, and thus service invocation using that GSR SHOULD only be trusted by a client based on independently established trust policies.

A *trusted resolver protocol* is one for which the client can, based on its pre-configured trust anchors, trust that the GSR returned by the resolver protocol for a GSH is correct. An example of a trust resolver protocol is the one associated with the https GSH scheme, as described in Section 4.7.2.2. The following additional properties are assumed from a trusted resolver protocol:

1. Multiple resolutions of the same GSH MUST result in GSRs that refer to the same Grid service instance.
2. All GSRs that are resolved from the same GSH MUST obey the same service interface and semantics, as defined by the **serviceType** of the instance referred to by those GSRs.

It is important to note that a trusted resolver protocol does not imply any particular trust relationship between the client and the service referred by the GSR. The fact that a client is able to resolve a GSH to a GSR from a trusted resolver protocol say nothing about whether that client can trust the service to which the GSR refers. Such trust MUST be established through some other means. For example, a client MAY trust a service because it received the GSH to that service from a registry that it trusts. In this situation, a trusted resolver protocol simply allows the client to trust that the mapping from that trusted GSH into a GSR is correct, and that it is not instead receiving a GSR to a Trojan service from the resolver.

4.7.2.1 http GSH scheme

A GSH MAY be a URI with an “http” scheme and that conforms to the http URL syntax [cite: RFC 1738]. Support for the http scheme in implementations of this specification is OPTIONAL.

An http GSH MAY be resolvable by performing an http 1.1 GET on the http URL, as specified in [cite: RFC 2616]. A successful resolution MUST result in a “200 OK” response from the http server, where the Content-Type is “text/xml; utf-8” [cite: RFC 3023], and the response is a WSDL encoded GSR, as defined in Section 4.7.1.1.

A common convention in the Web community is to use the http scheme for URIs (i.e. abstract names) instead of URLs (i.e. resolvable locations). We do not want to disallow this use of http URIs, and therefore cannot require that an http GSH be resolvable by performing an http GET on that GSH. This implies that upon receipt of an http GSH, a client MAY use some mechanism to resolve this GSH that is not defined in this specification. If that does not result in a GSR, the client SHOULD attempt to resolve the GSH by performing an http GET as defined above. However, the client MUST NOT assume that it will get a response to the http GET, or that the http GET response will necessarily be a WSDL encoded GSR.

Issue 28: Do we need to place restrictions on the http protocol that is used in an http GSH resolution? We probably do not need many of the more complicated features of http, and by disallowing them for GSH resolution we make it easier to write a compliant implementation of an http GET resolver client. For example, we might disallow a chunked response to the GET.

The http GSH resolver protocol is untrusted.

4.7.2.2 https GSH scheme

A GSH MAY be a URI with an "https" scheme and that conforms to the https URL syntax. (The https syntax is the same as that of http, except for the different scheme portion of the URL.) Support for the https scheme in implementations of this specification is OPTIONAL.

An https GSH MUST be resolvable by performing an http 1.1 GET on the http URL, as specified in [cite: RFC 2616]. The resolution protocol for an https GSH is the same as for an http GSH (Section 4.7.2.1), except with regard to security. Whereas an http GSH resolution is carried over an unprotected TCP channel, an https resolution (i.e. http GET) MUST be carried over a TLS [cite: RFC 2246] protected channel.

The https GSH resolver protocol MAY be a trusted, depending upon the configuration of the client's trust anchor. (Note that the "client" referred to here MAY not be the client of the Grid service instance referred to by the GSH, but MAY instead be a resolver service to whom that Grid service client has outsourced the resolution request.) With an https GSH, the GET operation MUST be performed over a server-authenticated TLS channel, as described in [cite: RFC 2818]. The client MUST compare the https URL hostname with the subject and subjectAltName fields of the X.509 certificate with which the https server authenticates. If the https URL hostname is included the server's X.509 certificate as described in [cite: RFC 2818], and if the client trusts the Certificate Authority that signed the server's certificate, then the client MAY trust that the GSR returned by that https server is a correct GSR for the input GSH. Otherwise, the client SHOULD treat the GSR as if it came from an untrusted resolver protocol.

A client MAY authenticate with the https server when setting up the TLS channel. If the client does not authenticate, or if the https server does not have a trust policy that recognizes the client authentication credentials, then the https server SHOULD respond to the http GET as if it knows nothing about the client. For example, the https server MAY reject the resolution request, by returning an http "401 Unauthorized" client error response. But if the https server does have a trust policy that recognizes the client, then the https server MAY apply that policy and return an appropriate GSR.

4.7.3 serviceLocator

The gsdl:**serviceLocator** type contains either a GSH or a GSR, but not both. It is used by various operations in this specification that may accept either a GSH or a GSR.

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 collection of related **portType**s defined in this specification 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 **createService** operation on a *Factory service*. (A service instance that implements 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 (see the **Destroy** operation, supported by the **GridService** portType: Section 6) 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 (see the **SetTerminationTime** operation in the GridService **portType**: Section 6).

In addition, a Grid service MAY support notification of lifetime-related events, through the standard notification interfaces defined in Section 8.

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 9), 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 in handle resolvers under its control.

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.

4.9 Common Handling of Operation Faults

OGSA will define a small collection of base fault messages that may be used by the **portType**s defined in this document. These faults will be described in this Section in a subsequent version of this document. An XML schema of various XML types and wsdl:parts that define common fault messages for reuse amongst the operations defined in this specification and operations defined in domain-specific portTypes will appear with the final XML definition of Grid Services (See Section: 15).

5 Grid Service Interfaces

A Grid service is defined as any Web service that MUST implement a **serviceType** that aggregates the Grid Service **portType** (See Section 6).

This specification defines a collection of common distributed computing patterns that are considered to be fundamental to OGSA. The embodiment of these patterns appears as WSDL **portTypes**. The collection of portTypes specified in this document is listed in Table 3.

The task for the designer of components within OGSA is to design **serviceTypes** that aggregate the **GridService portType**, zero or more other **portType**s defined in this specification and one or more **portType**s that define domain specific behavior.

Table 3 Summary of the portTypes defined in this document

PortType Name	See Section	Description
GridService	6	encapsulates the root behavior of the component model
HandleResolver	7	mapping from a GSH to a GSR
NotificationSource	8.1	allows clients to subscribe to notification messages
NotificationSubscription	8.2	defines the relationship between a single NotificationSource and NotificationSink pair

		and NotificationSink pair
NotificationSink	8.3	defines a single operation for delivering a notification message to the service instance that implements the operation
Factory	9	standard operation for creation of Grid service instances
Registration	10.2	allows clients to register and unregister registry contents

6 The GridService PortType

In this section and those that follow, we describe the various standard **portType**s that are defined by OGSA.

We start with the GridService **portType**, which MUST be implemented by all Grid services and 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. The behavior encapsulated by the GridService **portType** is that of querying against the **serviceDataSet** of the Grid service instance, and managing the termination of the instance.

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, and have attempted to take a middle road. The GridService **portType** provides several operations with typed parameters, but leaves considerable extensibility options within several of those parameters. Service data is then used to express what specific extensibility elements a particular service instance understands. Grid service designers are free to mix and match the document-centric and RPC approaches in the portTypes that they design to compose with those described here.

6.1 GridService PortType: Service Data Descriptions and Elements

The GridService **portType** includes **serviceData** elements conformant to the following **serviceDataDescription** elements:

```
<gsdl:serviceDataDescription
        name=”ServiceType”
        type=”qname”
        minOccurs=”1”
        maxOccurs=”1”
        mutability=”constant”>
   <wsdl:documentation>
      The qname of the serviceType implemented by this Grid service.
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

```
<gsdl:serviceDataDescription
        name="ServiceDataNames"
        type="qname"
        minOccurs="6"
        maxOccurs="unbounded"
        mutability="mutable">
   <wsdl:documentation>
      A set of qnames, one for each service data element contained
      in this service instance. Note:the minOccurs corresponds to
      the number of serviceDataDescription elements in this
      GridService portType that are required (that have minOccurs > 0).
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="FactoryHandle"
        type="uri"
        minOccurs="0"
        maxOccurs="1"
        mutability="constant">
   <wsdl:documentation>
      The Grid Service Handle to the factory that created
      this Grid service instance, if appropriate.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="GridServiceHandles"
        type="uri"
        minOccurs="1"
        maxOccurs="unbounded"
        mutability="append">
   <wsdl:documentation>
      The Grid Service Handles of this Grid service instance.
      It is possible to have multiple handles, for example one
      for each handle scheme that might be deployed.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="GridServiceReferences"
        type="xml:any"
        minOccurs="1"
        maxOccurs="unbounded"
        mutability="mutable">
   <wsdl:documentation>
      A set of Grid Service References to this Grid service instance.
      One service data value element MUST be the WSDL representation
      of the GSR. Other service data value elements may represent
      other forms of the WSDL.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="QueryExpressionTypes"
        type="xsd:anyURI"
```

```
        minOccurs="1"
        maxOccurs="unbounded"
        mutability="append">
   <wsdl:documentation>
      A set of URIs any one of which MAY be used by a client in the
      FindServiceData operation's QueryExpression parameter.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="TerminationTime"
        type="xsd:dateTime"
        minOccurs="1"
        maxOccurs="1"
        mutability="mutable">
   <wsdl:documentation>
      The dateTime value of the time when the lifetime for the Grid
      service will expire.
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

In addition, the **GridService portType** defines the following initial set of service data value elements:

```
<gsdl:serviceData
      name="gsdl:QueryExpressionTypes">
   <xsd:anyURI>
     http://www.gridforum.org/namespaces/2002/07/queryByServiceDataName
   </xsd:anyURI>
</gsdl:serviceData>
```

6.2 GridService PortType: Operations and Messages

Issue 29: Do we need an additional operation on GridService that allows a client to reliably determine if a GSH (or perhaps two GSHs) refers to particular instance?

6.2.1 GridService :: FindServiceData

Query the service data.

Input

- QueryExpressionType: The URI that identifies the query mechanism used. This MUST be one of the URIs declared in the QueryExpressionTypes service data element of the target instance.
- *QueryExpression*: The query to be performed. This is an extensible parameter, which MUST be expressed using the query mechanism identified by the QueryExpressionType parameter.

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 a *QueryExpressionType* identified by the http://www.gridforum.org/namespaces/2002/07/queryByServiceDataName URI that corresponds to **queryByServiceDataName**. A Grid service instance MAY support other *QueryExpressionTypes*

The list of query expression types supported by a Grid service instance is expressed in the instance's **QueryExpressionTypes** service data element. Therefore, a client can discover the query expression types supported by a service instance by performing a **FindServiceData** request on the instance, using the **queryByServiceDataName** expression type URI searching for the name gsdl:QueryExpressionTypes.

The service data that is available for query by a client MAY be subject to policy restrictions. For example, some service data elements MAY not be available to some clients, and some service data value elements within a SDE MAY not be available to some clients.

6.2.2 queryByServiceDataName

A **queryByServiceDataName** results in all service data elements that have a qname as specified by the **name** property.

The non-normative grammar of this type is:

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

The FindServiceData operation's *Result* output parameter for a **queryByServiceDataName** query MUST be the **serviceData** element that has the requested **serviceDataName**.

6.2.3 queryByXPath

Issue 30: Define the XPath QueryExpressionType, QueryExpression schema, and the result format. Support for this QueryExpressionType is optional.

6.2.4 queryByXQuery

Issue 31: Define the XQuery QueryExpressiontype, QueryExpression schema, and the result format. Support for this QueryExpressionType is optional.

6.2.5 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 as determined by the ClientTimestamp parameter ; MAY adjust its termination time, if necessary, based on its own polices 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 ClientTampstamp received in a previous SetTerminationTime request.

- *TerminationTime*: The earliest termination time of the Grid service that is acceptable to the client. Issue 14: Do we need to be able to express “forever”?

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):

Issue 16: Should the termination time be expressed as absolute time, or relative to receipt of the message?

Issue 15:Should we collapse SetTerminationTime and Destroy into a single, extensible SetTerminationPolicy operation? Then immediate destruction and soft-state destruction would simply be standard policies that could be expressed in the request.

6.2.6 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 acknowledging the receipt of the destroy message; 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:

- Acknowledgement that the destroy has been initiated

Fault(s):

-

7 The HandleResolver PortType

A *handle resolver* is a Grid service instance that implements the HandleResolver **portType**.

Issue 17: The authoring team is divided on the recommendations regarding the role of the resolver protocol vs the use of HandleResolver. Should there be language such as: “Clients SHOULD use one or more HandleResolver services to resolve GSHs to GSRs. The handle resolver protocols discussed in this section SHOULD NOT normally be used by clients, but SHOULD be used by HandleResolver services only.”

Each GSH scheme defines a particular *resolver* protocol for resolving a GSH of that scheme to a GSR. Some schemes, such as the http and https schemes defined in Section 4.7.2, MAY not require the use of a HandleResolver service, as they are based on some other resolver protocol. However, there are two situations where a Grid service based resolver protocol MAY be used, and which therefore motivates the definition of a standard **HandleResolver portType**. First, a GSH scheme MAY be defined that uses the **HandleResolver** as a fundamental part of its resolver protocol, where the GSH carries information about to which **HandleResolver** service instance a

client should send resolution requests. Second, in order to avoid placing undo burden on a client by requiring it to directly speak various resolver protocols, a client instead MAY be configured to outsource any GSH resolutions to a third party **HandleResolver** service. This outsourced handle resolver MAY in turn speak the scheme-specific resolver protocols directly. Both of these situations are addressed through the definition of the HandleResolver **portType**.

Various handle resolvers may have different approaches as to how they are populated with GSH to GSR mappings. Some handle resolvers may be tied directly into a hosting environment's lifetime management services, such that creation and destruction of instances will automatically add and remove mappings, through some out-of-band, hosting-environment-specific means. Other handle resolver services may implement the Registration **portType**, such that whenever a service instance registers its existence with the resolver, that resolver queries the GridServiceHandles and GridServiceReferences service data elements of that instance to construct its mapping database. Other handle resolver services may implement a custom registration protocol via a custom **portType**. But in all of these cases, the HandleResolver **portType** MAY be used to query the resolver service for GSH to GSR mappings.

7.1 HandleResolver PortType: Service Data Descriptions

The HandleResolver **portType** includes **serviceData** elements conformant to the following **serviceDataDescription** elements:

```
<gsdl:serviceDataDescription
        name="HandleResolverSchemes"
        type="xsd:anyURI"
        minOccurs="0"
        maxOccurs="unbounded"
        mutability="mutable">
   <wsdl:documentation>
      A set of URIs that correspond to the handleResolver schemes that
      the HandleResolver implements.
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

7.2 HandleResolver PortType: Operations and Messages

7.2.1 HandleResolver :: FindByHandle

Returns a Grid Service Reference for a Grid Service Handle.

Input

- *Handle*: A Grid Service Handle.
- *GSRSet*: (optional) a set of one or more GSRs that the client already possesses that are not satisfactory for some reason. This is a hint from the client that these existing references should not be returned in response to this message.

Output

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

Fault(s)

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

- *NoValidReferences available,* indicating that the service cannot return a GSR that is not already contained in the GSRSet input parameter.
- *Redirection*, indicating that the clients SHOULD reissue to request to a different handle resolver, as specified by a GSR that is returned with this fault case.

8 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 notification messages. A source MAY be able to send notification messages to any number of sinks.
- A *notification sink* is a Grid service instance that receives notification messages from any number of sources. A sink MAY implement the DeliverNotification operation of the NotificationSink **portType,** which allows it to receive notification messages of any type. Alternatively, a sink MAY implement a *specialized notification delivery operation* from a different **portType**, where that operation is a specialization of the DeliverNotification operation. A specialized delivery operation MAY only accept a subset of the types of messages that the general DeliverNotification operation can accept, and like DeliverNotification is an input-only operation (i.e. it does not return a response).
- A *notification message* is an XML element sent from a notification source to a notification sink. The XML type of that element is determined by the subscription expression.
- A *subscription expression* is an XML element that describes what messages should be sent from the notification source to the notification sink. The subscription express also describes when messages should be sent, based on changes to values within a service instance's **serviceDataSet**.
- In order to establish what and where notification messages are to be delivered, a *subscription* request is issued to a source, containing a subscription expression, the **serviceLocator** of the notification sink to which notification messages are to be sent, the portType and operation name of the specialized notification delivery operation to which notification messages should be sent, and an initial lifetime for the subscription.
- A subscription request causes the creation of a Grid service instance, called a *subscription*, which implements the NotificationSubscription **portType**. This **portType** MAY be used by clients to manage the (soft-state) lifetime of the subscription, and to discover properties of the subscription.

This notification framework allows for either direct service-to-service notification message delivery, or for the ability to integrate various intermediary delivery services. Intermediary delivery services might include: messaging service products commonly used in the commercial world, message filtering services, message archival and replay services, etc.

Issue 12: The GGF Grid Monitoring Architecture (GMA) working group has written a draft document called "A Grid Monitoring Architecture". The GS Spec basically includes an implementation of this architecture, via its service data, notification, and FindServiceData. However, the GS Spec uses "Source" and "Sink" instead of "Producer" and "Consumer". Should we change the GS Spec to use "Producer" and "Consumer", to bring it into alignment with the GMA terminology?

8.1 The NotificationSource PortType

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

8.1.1 NotificationSource PortType: Service Data Descriptions and Elements

The NotificationSource **portType** includes **serviceData** elements conformant to the following **serviceDataDescription** elements:

```
<gsdl:serviceDataDescription
        name="NotifiableServiceDataNames"
        type="qname"
        minOccurs="0"
        maxOccurs="unbounded"
        mutability="mutable">
   <wsdl:documentation>
      A set of qualified names of service data elements to
      which a client MAY subscribe for notification of changes.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="SubscriptionExpressionTypes"
        type="xsd:anyURI"
        minOccurs="1"
        maxOccurs="unbounded"
        mutability="append">
   <wsdl:documentation>
      A set of URIs identifying expression mechanisms that are
      understood by the Subscribe operation. One of these would be used
      as the operation's SubscriptionExpression parameter.
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

The NotificationSource portType would also include the following initial service data value elements:

```
<gsdl:serviceData
      name="gsdl:SubscriptionExpressionTypes">
   <xsd:anyURI>
 http://www.gridforum.org/namespaces/2002/07/subscribeByServiceDataName
   </xsd:anyURI>
</gsdl:serviceData>
```

8.1.2 NotificationSource PortType: Operations and Messages

8.1.2.1 NotificationSource :: Subscribe

Subscribe to be notified of subsequent changes to the target instance's service data. This operation creates a Grid service subscription instance, which MAY subsequently be used to manage the lifetime and discovery properties of the subscription.

Input:

- SubscriptionExpressionType: The URI that identifies the subscription mechanism used. This MUST be one of the URIs declared in the SubscriptionExpressionTypes service data element of the target instance.
- *SubscriptionExpression*: The subscription to be performed. This is an extensible parameter, which MUST be expressed using the subscription mechanism identified by the SubscriptionExpressionType parameter.
- *Sink*: The **serviceLocator** of the notification sink to which messages will be delivered. This locator MAY be to some other service than the one that is issuing this subscription request, thus allowing for third-party subscriptions.
- *SpecializedNotificationDeliveryOperation* (optional): The name of the **operation**, and the qname of the **portType** in which that operation is defined, to be used by the notification source when delivering messages to the notification sink. The operation signature MUST be the same as, or a specialization of, the NotificationSink::DeliverNotification operation. If this parameter is not specified, then it defaults to the "DeliverNotification" **operation** name that is defined in the gsdl:NotificationSink **portType**.
- *ExpirationTime*: The initial time at which this subscription instance should terminate, and thus notification delivery to this sink be halted. Normal GridService lifetime management operations MAY be used on the subscription instance to change its lifetime.

Output:

- *SubscriptionInstanceLocator*: A **serviceLocator** to the subscription instance that was created to manage this subscription. This subscription instance MUST implement the NotificationSubscription **portType**.

Fault(s):

Every Grid service instance that implements the NotificationSource portType MUST support a *SubsciptionExpressionType* identified by the http://www.gridforum.org/namespaces/2002/07/subscribeByServiceDataName URI that corresponds to **subscribeByServiceDataName**. A Grid service instance MAY support other *SubscriptionExpressionTypes*.

The list of subscription expression types supported by a Grid service instance is expressed in the instance's **SubscriptionExpressionTypes** service data element. Therefore, a client can discover the subscription expression types supported by a service instance by performing a **FindServiceData** request on the instance, using a **queryByServiceDataName** element, which contains the name "gsdl:SubscriptionExpressionTypes".

The service data that is available for subscription by a client MAY be subject to policy restrictions. For example, some service data elements MAY not be available to some clients, and some service data value elements within a SDE MAY not be available to some clients.

8.1.2.2 subscribeByServiceDataName

A **subscribeByServiceDataName** results in notification messages being sent whenever the named service data element changes.

The non-normative grammar of this type is:

```
<gsdl:subscribeByServiceDataName
        name="qname"
        minInterval="xsd:duration"?
        maxInterval=("nonNegativeInteger"|"unbounded")? >
```

```
</gsdl:subscribeByServiceDataName>
```

The **minInterval** property specifies the minimum interval between notification messages, expressed in xsd:duration. If this property is not specified, then the notification source MAY choose this value. A notification source MAY also reject a subscription request if it cannot satisfy the minimum interval requested.

The **maxInterval** property specifies the maximum interval between notification messages, expressed in xsd:duration. If this interval elapses without a change to the named service data element's value, then the source MUST resend the same value. When the valueis "unbounded" the source need never resend a service data value if it does not change. If this property is not specified, then the notification source MAY choose this value.

For a **subscribeByServiceDataName** subscription, the type of the notification message sent from the notification source to the notification sink MUST be the **serviceData** element that has the requested **serviceDataName**.

8.2 The NotificationSubscription PortType

A subscription for notification causes the creation of a Grid service *subscription* instance, which MUST implement the NotificationSubscription **portType**. This instance MAY be used by clients to manage the lifetime of the subscription, and discover properties of the subscription.

8.2.1 NotificationSubscription PortType: Service Data Descriptions

The NotificationSubscription **portType** includes **serviceData** elements conformant to the following **serviceDataDescription** elements:

```
<gsdl:serviceDataDescription
        name="SubscriptionExpression"
        type="xsd:any"
        minOccurs="1"
        maxOccurs="1"
        mutability="mutable">
   <wsdl:documentation>
      The current subscription expression managed by this
      subscription instance.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="SinkHandle"
        type="uri"
        minOccurs="1"
        maxOccurs="1"
        mutability="mutable">
   <wsdl:documentation>
      The Grid Service Handle of the Notificationsink to
      who this subscription is delivering messages.
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

Issue 18: In order to represent a subscription expression in service data, should we create a gsdl:**subscriptionExpression** type which contains an attribute with the URI of the subscription expression type, and an extensibility element to carry the actual subscription expression? The

SubscriptionExpress SDD should be changed to have this type. This same type can also then be used in the NotificationSource::Subscribe operation.

8.2.2 NotificationSubscription PortType: Operations and Messages

None.

8.3 The NotificationSink PortType

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

8.3.1 NotificationSink PortType: Service Data Descriptions

None.

8.3.2 NotificationSink PortType: Operations and Messages

8.3.2.1 NotificationSink :: DeliverNotification

Deliver message to this service.

Input:

- *Message*: An XML element containing the notification message. The content of the message is dependent upon the notification subscription.

Output:

The service does not reply to this request.

Fault(s):

8.4 Integration With Notification Intermediaries

While the NotificationSource and NotificationSink define how notification messaging is performed between two parties, these same **portType**s 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 by making service data elements available to subscribers that are different than SDEs of the original notification source. Intermediary notification sources are generally characterized by the fact that their **serviceData** elements have **originator** properties that refers 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 that of the notification source service that originally generated the message.
- To filter, modify, aggregate, and/or archive 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, integrating messaging service products, deserves further explanation. Such messaging service products can be exploited in this framework by:

1. Defining an intermediary 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. 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 to receive notification messages. This property allows for clients to stitch together notification message paths, without being directly in those paths.
3. The intermediary messaging service instance can advertise various notification topics service data elements 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.
4. The intermediary messaging service instance may have its own efficient, scalable, message distribution network, thus allowing the incoming message to be efficiently delivered to a large number of subscribing sinks. Or it may guarantee delivery of the notification message for some period of time, even in the face of various failures. Or it may distribute incoming notification messages to sinks in a round-robin fashion, rather then sending all notification messages to all sinks. The possible behaviors that the intermediary messaging service instance can introduce to the notification message delivery are limitless.
5. The intermediary messaging service instance, the originating notification source service, and the final notification sink service may all implement a specialized 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.

9 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 **portType**s (in addition to the required GridService **portType**), such as:

- Registration (Section10), 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 handle resolution service (see Section 7). The method by which this registration is

accomplished is specific to the hosting environment, and is therefore outside the scope of this specification.

Issue 5: Consider adding another factory related portType that allows for management of the set of serviceTypes that a factory may create. Perhaps allow for downloading of code into the factory when adding a service.

9.1 Factory PortType: Service Data Descriptions

The Factory **portType** includes **serviceData** elements conformant to the following **serviceDataDescription** elements:

```
<gsdl:serviceDataDescription
        name="CreatesServiceTypes"
        type="qname"
        minOccurs="1"
        maxOccurs="unbounded"
        mutability="append"
   <wsdl:documentation>
      QNames to serviceTypes that are created by this Factory.
   </wsdl:documentation>
</gsdl:serviceDataDescription>

<gsdl:serviceDataDescription
        name="CreationInputTypes"
        type="qname"
        minOccurs="0"
        maxOccurs="unbounded"
        mutability="append"
   <wsdl:documentation>
      Qnames of 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>
```

9.2 Factory PortType: Operations and Messages

9.2.1 Factory :: CreateService

Create a new Grid service instance. Note that to support soft state lifetime management (Section 4.8), 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

- *TerminationTime* (optional): The earliest initial termination time of the Grid service instance that is acceptable to the client.
- *ServiceParameters* (optional): An XML document that is specific to the factory and the services that it creates.

Output

- *ServiceLocator*: A **serviceLocator** to the newly created Grid service instance.
- *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):

Issue 19: Can we reduce the number of output parameters in Factory::CreateService by moving many of them into service data of the created instance?

10 Registration

A *registry* is a Grid service that maintains a collection of Grid Service Handles, with policies associated with that collection. Clients may query the registry to discover what services are available.

A registry implements the Registration **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**. A registry MAY implement custom portType that define service data elements that are structured to support particular types of queries against the registry. The **FindServiceData** (see Section 6.2.1) operation provides rich query interface against the contents of the registry that is maintained in service data. 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 8), in order to support notification of registry existence and changes in registry contents.

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

10.1 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 via the FindServiceData operation.

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.
2. WS-Inspection link elements that refer to registry services, where the location values are the GSHs of services.
3. any other valid WS-Inspection element.

Issue 20: Should a WS-Inspection document be required of all registries? Or might a registry just use the RegisterService operation, define its own service data to represent the contents of the registry, and ignore the WS-Inspection document entirely? In other words, should the WS-

Inspection document be given a minOccurs=”0”, or separated out into a different portType entirely?

10.2 The Registration portType

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

10.2.1 Registration PortType: Service Data Descriptions

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

```
<gsdl:serviceDataDescription name=”GridServiceRegistryWSInspection”
        type=”wsil:inspection”
        minOccurs=”1”
        maxOccurs=”1”
        mutability=”mutable”
   <wsdl:documentation>
      A WS-Inspection document containing all of the Grid services in
      this registry. We expect that specializations of registries that
take this registration port type and combine it with other mechanisms
to do discovery (for example a specialization that finds discovery such
as “find by portType”). This portType forms the base assumption that an
inspection document is available.
   </wsdl:documentation>
</gsdl:serviceDataDescription>
```

10.2.2 Registration PortType: Operations and Messages

10.2.2.1 Registration :: RegisterService

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

Input

- *Handle*: The Grid Service Handle of the service to register. If the registry already contains a registration that matches this parameter, this operation is to be treated as an “update” to the registration information.
- *Timeout*: The time at which this registration should timeout and be removed from the registry, unless the registry receives a subsequent RegisterService.
- *Description* (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 Registration 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.

10.2.2.2 Registration :: UnregisterService

Remove a Grid Service Handle from the registry.

Input:

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

Output:

- Acknowledgement of receipt of the UnregisterService

Fault(s):

- Unregistration operation failed.

11 Change Log

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

- Improved introduction to Section 4, "The Grid Service, and reordered the subsections to make it flow better.
- Added Section 4.2, "Service Description and Service Instance", 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
- Moved description of **instanceOf** to be part of the WSDL GSR description, 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.

11.2 Draft 2 (6/13/2002) → Draft 3 (07/17/2002)

- Changed draft to assume new features in WSDL v1.2 draft 1, including **serviceType** and an open extensibility model.
- Added serviceType reuse/extension.
- Modified notion of Handle to be a URI, reflected changes in GSR and HandleResolver (previously called HandleMap) discussion. Introduced resolver protocols for the http and https GSH schemes.
- Substantially changed "Service Data" section (4.3), primarily to cleanup and plug holes in service data descriptions and elements, particularly around naming and typing. Changed various portType descriptions to reflect this change to service data.

- Added section “Modeling Time in OGSA”
- Overhauled the notification section, to completely integrate with service data, and to provide a “push model” that parallels the FindServiceData “pull model”.
- Renamed Registry portType to Registration, and did some cleanup on the section.
- Introduced gsdl:**serviceLocator**, which is an XML schema type that can be either a GSH or GSR. Changed various GSH and GSR argument to use this type.
- Renamed “Terminology and Abbreviations” section to “Notational Conventions”. In this section, added a table of namespace prefixes used in throughout the document, and cleaned up the rest of the section.
- Added inline “Issue” that need to be resolved, with numbers that refer to the GGF OGSI working group bugzilla database.

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, Ed Boden, Brian Carpenter, Francisco Curbera, Andrew Grimshaw, Marty Humphrey, Keith Jackson, Bill Johnston, Kate Keahey, Gregor von Laszewski, Lee Liming, Miron Livny, Tom Maguire, Norman Paton, Jean-Pierre Prost, John Rofrano, Thomas Sandholm, Ellen Stokes, Scott Sylvester, Sanjiva Weerawarana, Von Welch and Mike Williams.

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-wsilover.
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.
8. W3C Web Services Description Language 1.2 (Working Draft), http://www.w3.org/TR/2002/WD-wsdl12-20020709

9. Gunter, D, Tierney, B. A Timestamp for Distributed Computing (Draft), Global Grid Forum OGSI Working Group.

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.

Pending agreement from the OGSI-WG community on the directions and changes in this draft of the specification, the authors will produce formal WSDL and related XML definitions shortly after GGF5.