Open Grid Service Infrastructure Primer

Abstract

The OGSI Primer is an introduction to the Open Grid Services Infrastructure (OGSI) Specification that is aimed at a wide audience of architects and developers, implementers and users. No prior knowledge of the Grid or Web Services is assumed.

This is not a definitive specification of OGSI, but is intended to provide an easily readable description for quick understanding and summary of the basic fundamentals of creating and using OGSI-based services. It introduces the requirements for constructing large scale, distributed systems known as Grids and the way these requirements can be satisfied using Web Services and the OGSI specification.

Contents

Table of Figures

1. About this Primer

This chapter introduces the structure of this document, its intended audience, and other material that might be relevant.

1.1. Who should read this Document?

This is an introductory document to the Open Grid Services Infrastructure (OGSI) Specification [1] that is aimed at a wide audience of architects and developers, implementers and users.

No prior knowledge of the Grid or Web Services is assumed, though an awareness of distributed computing will help. If you plan to develop your own Grid Services, some knowledge or experience of XML, XML Schema and WSDL syntax would help you to understand the WSDL examples that appear in some chapters. If you need extensive background material on Grids, *The Anatomy of the Grid* [2] should help.

The remaining chapters of this document are organized as follows:

Chapter 2 summarizes the background of Grid computing, the requirements and context which inspired OGSI, and its relationship to the architectures and systems on which it builds. This chapter is suitable for a non-technical audience.

Chapter 3 gives an overview of Web Services whose specifications and technology form the basis for OGSI. It describes some major mechanisms underlying Web Services.

Chapter 4 is more technical. It describes the features introduced to Web Services by OGSI and introduces a simple example of a Web Service and a corresponding Grid Service as a comparison. This example is also used in succeeding chapters.

Chapter 5 covers all the main concepts and terminology of OGSI. It describes the scope and the framework in which the example of Chapter 4 can be created, and the techniques which might be used to extend it. Readers who are very familiar with Web Services, know the motivation for OGSI and simply want a summary of OGSI may start at this point.

Chapters 6 and 7 explain how Grid Services can be constructed and discusses scalability related features that are enabled by OGSI. These are the features necessary to create large-scale applications and systems.

From Chapter 8 onwards, the details of Grid Services interfaces are explained in a way that parallels the OGSI Specification. This makes it easy to correlate the information in the two documents.

Examples are used throughout the Primer to illustrate the features that the OGSI Specification requires. The examples are deliberately simple in order to avoid any need for knowledge of more realistic, but usually more complex, Grid applications. Also, although they contain correct interface definitions for the functions described, implementations are not provided. This is because the OGSI Specification aims to define a standard for interoperability while leaving freedom to implement Grid Services in a wide variety of ways. For implementation details, the reader should investigate one of the Grid Service toolkits such as Globus GT3 [3] and identify sample services that correspond to the ones described here.

1.2. Related Documents

This document is a companion document to the Open Grid Service Infrastructure (OGSI) Specification [1]. The Specification is the complete and authoritative description of Grid Services and should always be used to resolve questions of detail or ambiguity.

This document is, as far as possible, a self-contained introduction to the concepts of Grid Services and their main features that should quickly provide insight into the way Grid Services can be used and operated. Some of its sections are aligned to corresponding topics in the OGSI Specification, but it has more introductory material and uses illustrative examples that are not constrained by the need to be complete in detail, though this is also a goal wherever possible.

2. Grids and their Requirements

2.1. What are Grids?

Grid computing is a way of organizing computing resources so that they can be flexibly and dynamically allocated and accessed, often to solve problems requiring many organizations' resources. The resources can include central processors, storage, network bandwidth, databases, applications, sensors and so on. The objective of Grid computing is to make resources available so they can be more efficiently utilized.

The advantage of sharing is clearest when the need for resources is unpredictable, short-term, or changes quickly, or where it is simply larger than any single organization's capability to provide for it. For example, some problems that might take days to solve on a single installation's processing resources can be reduced to a few minutes with the right kind of parallelization and distribution on additional allocated computational resources.

This reduction in turn-around time has opened up areas and styles for computing applications that have previously been impractical. Similarly, information sharing can lead to discoveries and analytical predictions which are simply not possible with only part of the information. Such sharing may be enabled by sharing of access to storage resources, if the information is voluminous, or by agreeing pervasive standards for data description and access.

2.2. Features and Requirements of Computational Grid Systems

This section describes the main features of one kind of Grid system as background to the motivation for OGSI, namely a Grid which enables the sharing of processing capacity, sometimes called a Computational Grid. Figure 2-1 illustrates the concept.

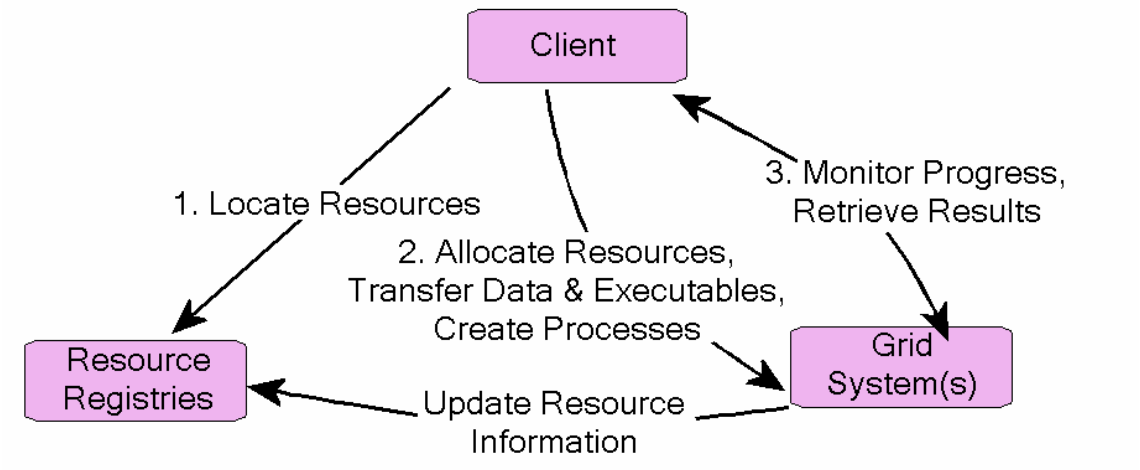

Figure 2-1: Common features of computational Grid systems.

The 'client' shown in the diagram need not be an end-user system, but could be an agent acting on its behalf, and there may be many such clients or agents, acting independently with no central control over the components shown. To make use of the resources, a client first uses information sources, here called 'Resource Registries', to discover those resources needed for execution of a task: multiple sources may need to be consulted to locate all the resources needed for a computation. Having located them, the client contacts the owning resource manager to request allocations for CPUs, secondary storage and/or other needed

resources. Assuming the discovery and allocation steps were successful, the client then sends the input data and executables, receiving a reference to the execution in return. These actions may be accomplished in several stages or as one consolidated action, depending on the nature and complexity of the task. As resources are allocated, the resource manager may need to update the information in the registry to enable reasonable bids for resource allocation from other clients. Lastly, the client monitors execution of the task using the reference it previously received. It can retrieve the results, or be sent status of the task as it progresses.

The construction of a computational Grid, and other kinds of Grid systems, requires facilities common to general distributed computing architectures – for example, a mechanism for identifying information by globally understood references which can be passed from one system to another, and for subscribing to updates to information sources as the information changes. However, Grids make extreme demands on distributed programming because they are typically large-scale, and they exploit wide-ranging networks consisting of a variety of protocols and systems which may span organizational boundaries. Some of these issues are addressed by standards which are not part of the scope of OGSI: interoperability amongst heterogeneous systems is addressed by Web Services standards (described in Chapter 3), while security issues relevant to widely-interconnected systems are being addressed by groups within the Global Grid Forum (GGF). OGSI exploits these standards, adding to them where necessary, and focuses on the mechanisms that enable resources to be located, managed and exploited at large scale. The concepts are introduced fully in Chapter 5, but, to summarize briefly, the main requirements addressed by OGSI are:

- Defining a framework which establishes a necessary common basic behaviour, while allowing wide scope for extension and refinement (such as optimization) and specialisation to particular uses.
- As part of the basic behaviour, enabling systems to describe their own interfaces (a facility known as introspection) as a means of flexibly enabling extensions, and to describe their underlying capabilities, such as the speed of a CPU or the currency or progeny of a database. The particular vocabularies for comparing capabilities are not part of OGSI, but the framework for identifying, describing, querying and modifying the descriptive information or other state of the system is central.
- The identification and description of the state which is the target of messages also enables a standard approach, called ‘statefulness’, to managing interactions between systems and their users.
- As part of the basic behaviour, establishing a system of communication of information about systems which requires the least synchronization amongst them while still providing a useful aggregation of information and behaviour. This attribute of OGSI is known as ‘soft-state’ ([4], [6]) and exploits caching of information for a limited period which can be extended by explicit requests from its users.
- As one aspect of statefulness and soft-state, describing a basic style of behaviour which gives artifacts (identities, information, resource allocations) an explicit point of creation and an explicit, limited but extendable lifetime. This provides a natural recovery mechanism for resources in the case of failed processes.

The next section explores another fundamental need: the form of standards which allow wide interoperability based on the concept of **services**.

2.3. Why a 'Service-Oriented' Architecture?

In computing terms, a 'service' is simply a function that can be invoked via a well-defined remote interface and in the terminology of OGSI a 'service' is used to represent the behaviour of any application, database, resource or any other artefact. A system defined in terms of service interfaces allows largely independent development and management of the requester (client) and the service provider. 'Service-Oriented Architecture' (SOA) [5] is an architectural style whose goal is looser coupling among interacting software systems compared to earlier forms of distributing computing architecture such as RPC [7], CORBA [8], Java RMI [9] and DCOM [10]. It allows great flexibility in the interconnecting protocol and the underlying platform for clients and servers, all of which are advantageous in constructing Grid systems. In this section we review the essential components of an SOA and in the next, their relationship to OGSI.

A service is defined in terms of the messages one uses to interact with it and the behaviour expected in response. This behaviour may depend on the **state** of resources such as files, databases or sensors that are encompassed by the service and the state may change in response to messages from one or more clients, or on internally generated events such as timers or external physical events. The messages, state and other behaviour must be described by a **service definition.**

A good service definition permits a variety of **implementations.** For example, an FTP server speaks the FTP protocol and supports remote read and write access to a collection of files. One FTP server implementation may simply write to and read from the server's local disk, while another may write to and read from a mass storage system, automatically compressing and uncompressing files in the process. If variants are possible, then discovery mechanisms that allow a client to determine the properties of a particular implementation of a service are important.

From the client's point of view, an important aspect of the implementation of a service is the **protocol** that is used to communicate requests. A well-constructed server side implementation may permit several protocols. For example, the http and https [11] protocols can both be used to retrieve Web pages from Web servers, but the latter is preferable if security is important.

2.4. Exploiting Web Services Architecture

Web Services is a particular type of Service-Oriented Architecture whose interfaces are defined using the Web Services Description Language (WSDL) and which focuses on simple, Internet-based standards (such as eXtensible Markup Language (XML) and the http protocol [11]) to enable heterogeneous distributed computing.

OGSI describes Grid Services using Web Services as a basis. This means exploiting:

- The mechanisms for encoding message transmission protocols that are described as **bindings** in Web Services. There are many ways of encoding messages, and Web Services separates these concerns from the XML definitions of application interfaces: OGSI standardizes interfaces at the application level, independently of the bindings.
- The conventions used in Web Services to separate the primary application interfaces (function calls and their parameters) from functions which can be managed by standardized, generic middleware. This includes issues such as authentication and access control.

- The techniques used in Web Services to separate service- and network-management functions from the application interface. The management issues include workload balancing, performance monitoring, and problem diagnosis.

Development of Version 1.0 of the OGSI Specification to the requirements already described has required extensions to WSDL [12]. The extensions are described in detail in Chapters 4 and 5, but they can be summarized as:

- A mechanism for defining an interface in terms of an 'extension' of one or more simpler interfaces. This is a reliable way of establishing the basic behaviour of a Grid Service.
- Enabling an XML-based language for describing the information (or 'state') associated with a service.

The extended language used to describe Grid Services is known as GWSDL. In the future, OGSI may be able to exploit developments in Web Service standards which can reduce, or eliminate, the need for Grid-specific extensions. Indeed, this is anticipated in developments by the Web Services Description Working Group [13], and the OGSI working group of the GGF, which developed the specification, is committed to updating it when this becomes possible.

2.5. What is needed to provide and use Grid Services?

In this section we introduce the requirements for the implementations of clients and servers participating in a Grid system. A client may, of course, also be a Grid Service in its own right, but the two aspects can be separately addressed. Further details of possible implementations are described in Chapter 6.

2.5.1. The Client Side

There are two meanings to the term 'client'. The end-users of a Grid system are its ultimate clients, but they may be shielded from the internal workings and requirements of the underlying systems by intermediaries such as Web portals, whose main objective is ease of use for the end-user. Secondly, there are the direct clients of Grid Services, which may be part of such a Web portal and are part of the concern of this Primer. An example of this type of client is illustrated in Figure 2-1 (on page 8). OGSI specifies many things that are required of services in order for them to be used by clients in a standard way.

Since OGSI is based on Web Services standards, a client which is capable of interacting with a Web Service can also interact with a Grid Service provided the interface description has been translated from GWSDL into WSDL. The normal process for developing services and applications in Service-Oriented Architectures can then be followed: the client may use the interface description during development of a Grid application to construct calls to the service, and can locate a suitable server and communicate with it using Web Service protocols during execution of the application.

2.5.2. The Server side

The provider of a Grid Service can take many forms, and the OGSI Specification is designed to impose the least possible restriction on its implementation, which can range from dedicated hardware components through operating system services and custom-built applications. However, since the OGSI standard is built on Web Services, a platform for hosting Grid Services may be built on existing systems for hosting Web Services, such as those based on the J2EE [14] component model. Such a system can provide many of the basics of a Grid Service via standard layers and services known as the 'hosting environment', and exploitation

of such an environment and associated development tools is an efficient way of developing Grid applications.

The details of the interfaces between a Grid Service implementation and a hosting environment will depend on the particular system, but some aspects of such systems can be described in generic terms. Firstly, the application-specific operations and state of the service can be described using an interface description language or tools specific to that system, such as Java for a J2EE system. This interface description can be translated into a form described by Web Services and OGSI for consumption by clients. Secondly, the runtime system creates the interface between the service implementation and the hosting environment. This may include management of the state of the service, converting messages to and from the implementation language, and provision of standard operations for Grid Services.

2.6. Associated and dependent standards

The OGSI Specification itself provides only basic levels of function. It enables Grid Services to be created, managed, discovered and destroyed, but does not say anything about the resources, databases and applications that make up a Grid. Standard descriptions of these facilities are required and OGSI provides a language and building blocks from which descriptions can be constructed. The work of defining high level constructs is in progress among working groups of GGF. These include standard techniques for accessing data, for configuring resources, for scheduling and allocating them, for monitoring system performance, and for accounting and paying for use. The Architecture for these standards is collectively called the Open Grid Services Architecture (OGSA).

3. Background Technology

This chapter provides a high-level overview of Web Services, which form the foundation for OGSI. It necessarily leaves out much detail, but describes the structure and important features and explains why these are important, especially in the context of Grids. The next chapter goes into more technical detail and illustrates Web Services and Grid Services via a simple example.

A more thorough and precise view of Web Services can be found in the Web Services Architecture document [19].

3.1. The Essentials of Web Services

3.1.1. Interactions defined by messages

For the purposes of this Primer we will define a Web Service as something that can be described using the Web Services Description Language (WSDL) [12]. A WSDL document defines the set of messages, the encodings, and protocols used to communicate with a service. Note that Web Service standards say nothing about the application interfaces (APIs) or the technologies used to manipulate the messages. This approach ensures that interoperability is the main focus of attention.

3.1.2. Exploiting Web Services standards and facilities

The use of WSDL defines message exchanges between users of the service and its implementers, but it also allows exploitation of a growing body of related standards for describing services, publishing them in registries, defining and implementing message protocols. These related standards provide facilities which simplify and support the Service itself. The simple concept of some of these facilities is shown in Figure 3-1.

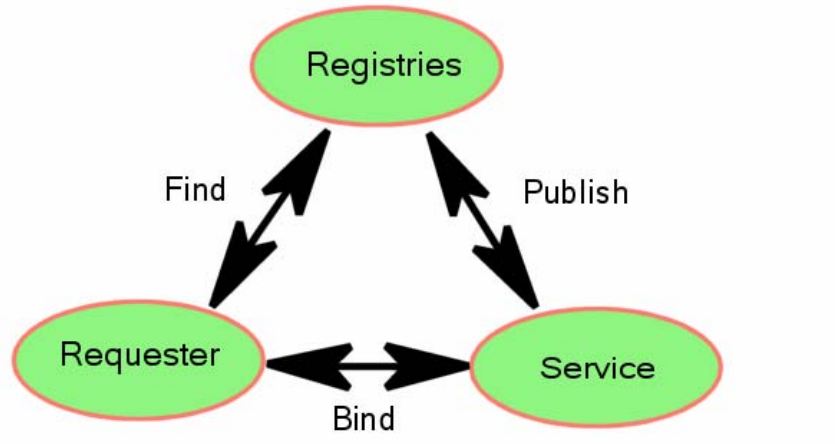

Figure 3-1: The basis of Web Service computing

The Services are created and deployed on a server which is able to receive messages in a particular encoding, over a transfer protocol. The server may enable several styles of encoding and/or protocol pairs, each of which is called a **binding** because it binds a high-level service definition to a low-level means of invoking the service and these can be described using the service's WSDL document. The server publishes the presence of a service in a Registry

(sometimes called a directory) which is also known to potential consumers. From the Requester's point of view, a Registry represents a collection of services which may provide suitable implementations of the interfaces that it needs to use and it may find them using criteria associated with bindings such as the security and transport protocols.

The design and implementation of Registries is a large and important topic, and many different solutions exist. It is the objective of the OGSI Specification to be able to exploit standards such as UDDI [20], which have been created for use by Web Services. There may be aspects of Registries and service description which need specific extensions or enhancements for Grid computing, and OGSI aims to enable these, and to adopt new Web Services standards as these develop. Figure 3-2 illustrates one sense in which the Computational Grid described in Chapter 2 extends the concepts of Web Services.

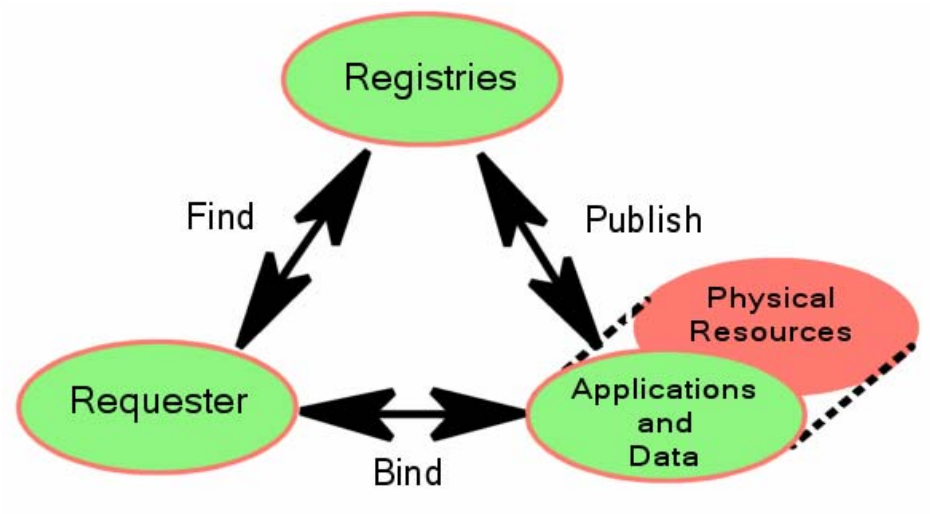

Figure 3-2: Web Services facilities used in a Computational Grid

In many Web Services, and indeed for many used in Grid systems, the Service presents an interface to the combination of data, application and physical resources needed to perform the service. However, in a Computational Grid a Service may represent the physical resources more explicitly, separately from the data and applications which exploit them. In this case, information visible to clients can change more dynamically. As Section 2.2 described, OGSI provides a framework of Service descriptions, including descriptions of state, which improve the manageability of distributed, dynamic information at large scale.

The definition of bindings is an aspect of Web Services which OGSI exploits directly. Although any kind of message encoding and security protocol can be used between a Server and Requester within the framework of Web Services, standard encodings are needed to ensure easy, pervasive interoperability. The Web Services Interoperability Organization (WS-I) [21] exists to make appropriate recommendations. However, leading-edge implementations of Web Services, including Grid applications, may, from time to time, adopt protocols which are not yet standardised.

4. Describing Grid Services

4.1. Web Services Description Language (WSDL)

WSDL is an XML document for describing a contract between a Web Service and its consumers about the format of the messages that can be exchanged and the message exchange patterns that the service can support. Figure 4-1 outlines the structure and major parts of a WSDL document, together with their relationships.

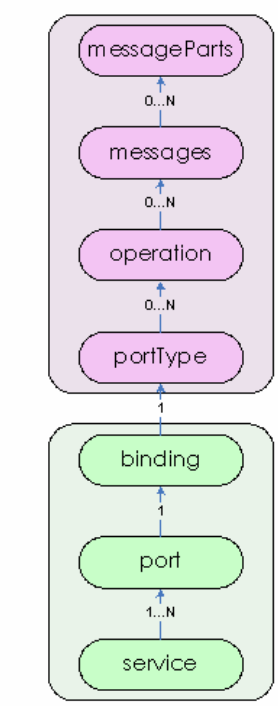

Figure 4-1: WSDL document structure

The elements of the document are divided into two vertical sections:

- The top section ('message', 'portType') is the abstract definition of the service interface, its operations, and their parameters. These definitions determine what a consumer of the service can do.
- The bottom section defines the binding(s) of the abstract definition to concrete message formats, protocols and endpoint addresses through which the service can be invoked.

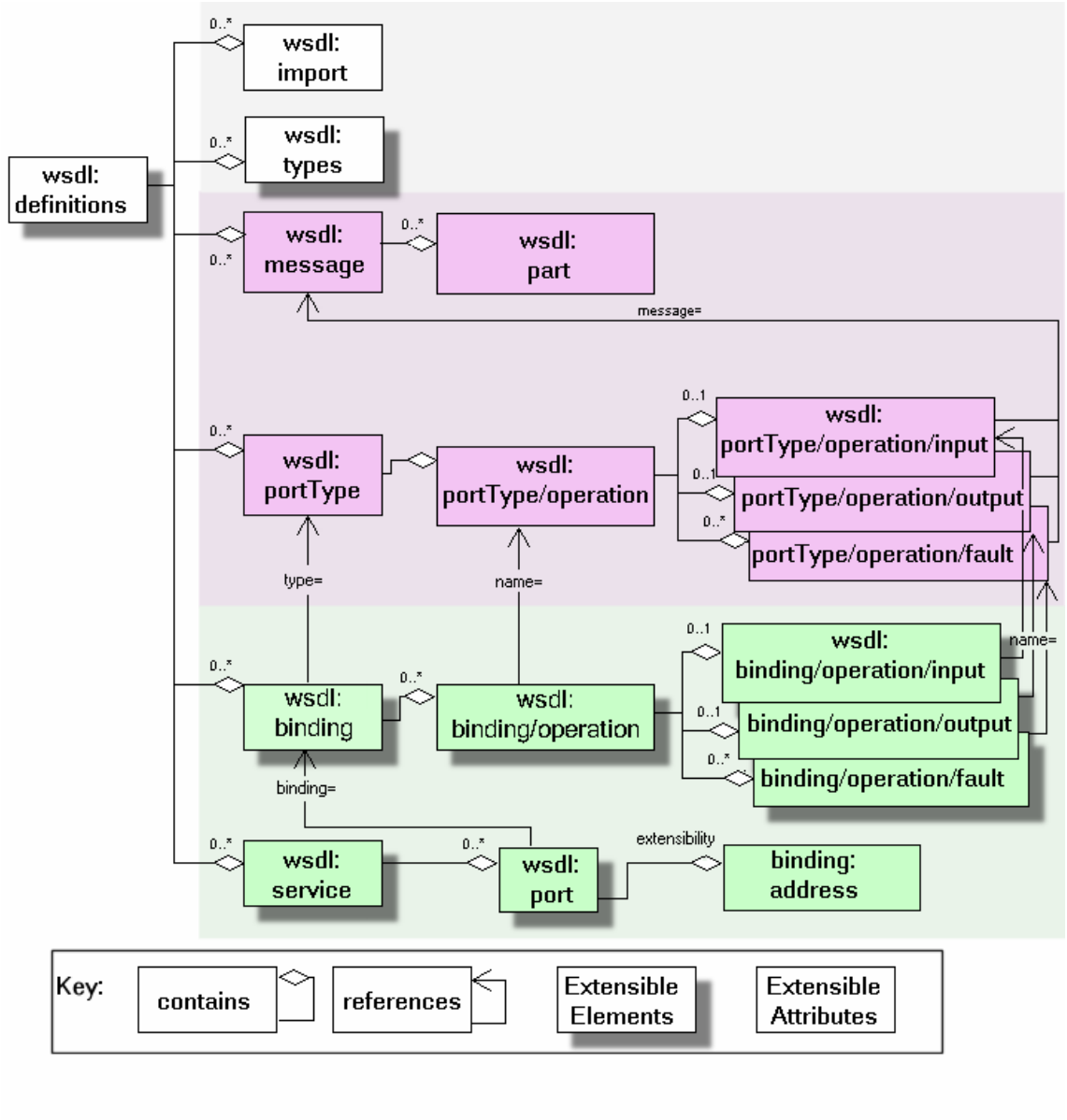

Figure 4-2: WSDL markup elements

Figure 4-2 describes the structure of a WSDL document in more detail. The root XML element called the wsdl:definitions is on the left of the diagram. It may contain the subsidiary elements, shown via the 'contains' linkage, such as wsdl:import and wsdl:types. Subsidiary elements can be repeated as shown by the range indicator; for example, all subsidiary elements of wsdl:definitions can appear any number of times (0..*).

At the heart of a WSDL description is the `portType` definition which defines a group of input, output, and fault messages that a service is prepared to accept or produce and the message exchange patterns in which it is prepared to participate. Sometimes, it is easy to think of a `portType` as a Java interface or a C++ class. A `message` in a `portType` may be composed of many parts, where each part can be of a different type. The `message` parts can be thought of as input and output parameters; a `message` simply combines them. Section 4.3 contains a simple example.

The types of message parts are defined within the `types` component of `wsdl:definition`. This element is *extensible,* meaning that it can contain arbitrary subsidiary elements to allow general data types to be constructed. The default type system in WSDL is XML Schema, which we won't discuss here in detail. In order to understand our example it is sufficient to know that at the end of the definition we link the type names of message parts to their XML definitions. WSDL allows the other elements to contain subsidiary extension elements as a means of allowing flexible description of interface types; these extension points are shown in the diagram. The purpose of other extensions is described in Appendix A.3 of WSDL[4].

Another important part of the WSDL definition is the `binding`. It describes the concrete implementation of messages: that is a data encoding, messaging protocol, and underlying communication protocol. The XML elements of a binding are operations which reference the corresponding abstract messages by name. An important aspect of WSDL bindings is their capacity for extension by the addition of new subsidiary element definitions. WSDL allows any and all data encoding, message and communications protocols to be described. The usefulness of a particular description depends on having client and server systems that are able to interpret the WSDL descriptions to encode/decode the messages.

4.2. Grid Services Description Language (GWSDL)

OGSI adds the features of Grid Services to basic Web Services by redefining the WSDL portType element. Grid Services are described using an extended form of WSDL known as GWSDL. Figure 4-3 illustrates the main structure of a GWSDL document, which is very similar to the WSDL structure shown in Figure 4-1 (page 15).

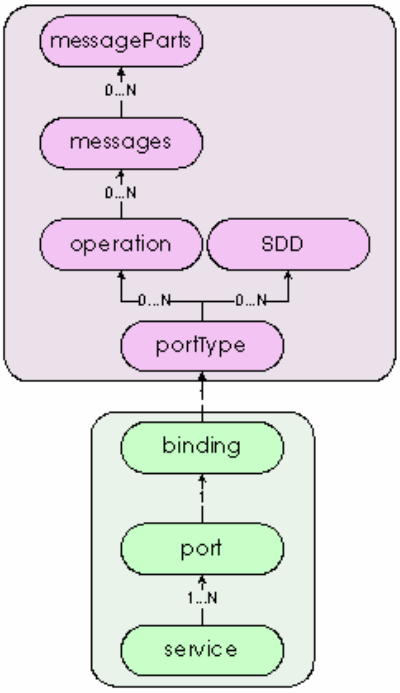

Figure 4-3: GWSDL document structure

The differences between WSDL and GWSDL are illustrated in more detail in Figure 4-4.

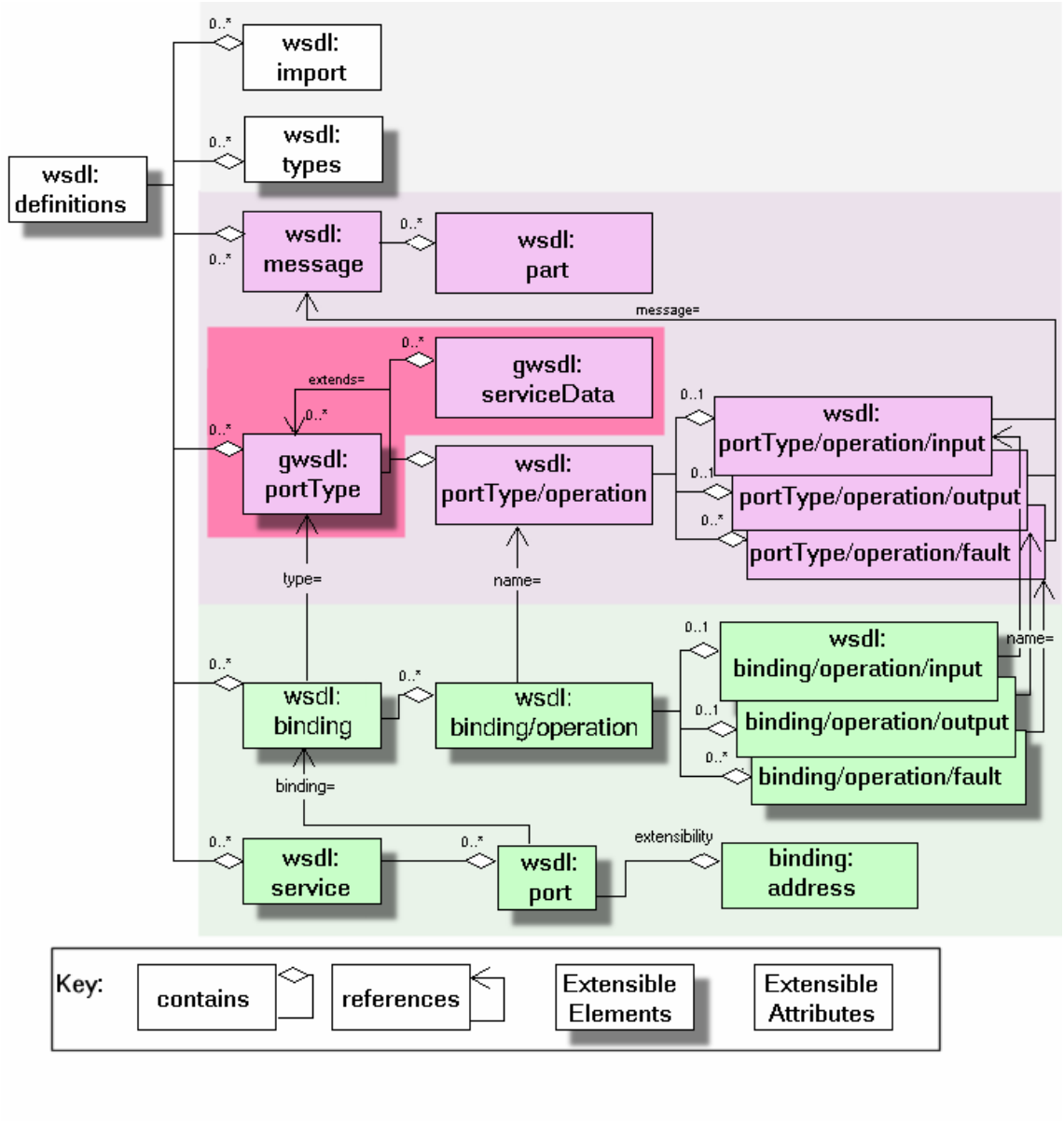

Figure 4-4: Markup language for Grid Services

- In OGSI, a `portType` can be constructed by referencing an existing `portType` or `portTypes`, with the `extends` attribute, adding new definitions as required. This enables standard behaviour to be defined authoritatively, and to be referenced by service definitions that need to incorporate it. Standard behaviour is important to clients that may need to use independently implemented services.
- While a WSDL portType contains only operations, an OGSI portType can also contain Service Data Declarations (SDDs), which help describe the identity and interfaces of the service, and how its state can be viewed by a client.

As an example, Section 4.3 (page 19) contains a GWSDL version of the Counter service.

These additions to the WSDL document schema are prefixed with *gwsdl* to identify their definition by OGSI. They may be incorporated into subsequent versions of WSDL, eliminating the need for new markup elements defined by OGSI. They have been addressed

by the current "work in progress" draft of WSDL 1.2 in the W3C Web Services Description Working Group [WSDL 1.2 DRAFT], but because WSDL 1.2 is currently "work in progress", OGSI cannot directly incorporate the entire WSDL 1.2 body of work.

Instead, OGSI defines an extension to WSDL 1.1, isolated to the wsdl:portType element, that provides the minimal required extensions to WSDL 1.1. These extensions to WSDL 1.1 match equivalent functionality agreed to by the W3C Web Services Description Working Group.

4.3. The 'Counter' Example

In this section, some of the Grid Services concepts and terminology are introduced through the Counter Service example, which is used as a vehicle for the discussion throughout the rest of this Primer. The relationship between Web and Grid Services is also demonstrated.

4.3.1. Counter Web Service

The WSDL for a simple Counter Service with two operations (`increase` and `getValue`) is shown in Listing 4-1 below (the bindings and service sections are not presented):

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:counter="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/webservice"
                  targetNamespace="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/webservice">
   <wsdl:types>
      <xs:schema/>
   </wsdl:types>
   <wsdl:message name="increaseMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <wsdl:message name="getValueMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <wsdl:portType name="counterPortType">
      <wsdl:operation name="increase">
         <wsdl:input message="increaseMsg"/>
      </wsdl:operation>
      <wsdl:operation name="getValue">
         <wsdl:output message="getValueMsg"/>
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

Listing 4-1: WSDL document for the Counter Web Service

In the Web Services world, the Counter Web Service is an agent whose interface is described in WSDL, and it is the logical receiver of operation requests.

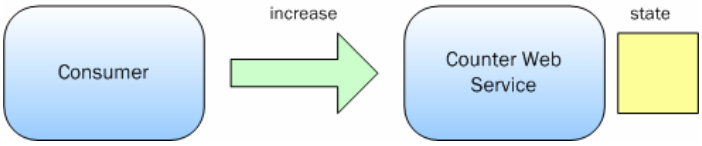

Figure 4-5: An increase operation to the Counter Web Service

The Counter Web Service of Figure 4-5 implements the WSDL interface presented in Listing 4-1. The interface does not say anything about the state that must be maintained by the

implementation of the Web Service. However, for the Counter Web Service to be useful, state has to be maintained in an implementation-specific way (e.g., in memory, in a database, in a file, etc.). Multiple consumers that have discovered the Counter Web Service through a registry can use its operations to change the counter's state (any security-related issues are outside the scope of this discussion and are not considered). In Figure 4-5, a consumer of the service submits a request for an increase operation.

If a Counter Web Service was to be extended so that multiple counters were supported, an ID would have to be introduced. The ID would identify the particular counter on which an operation was to be executed. Although the interface of the Counter Web Service would have to change in order to accept the ID, the same Counter Web Service would still be the logical receiver of the operations (Figure 4-6).

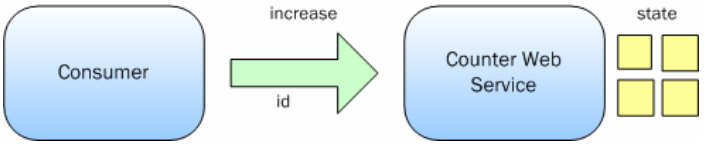

Figure 4-6: An increase operation to a Counter Web Service that supports multiple counters

Listing 4-2 has the modified WSDL of the Counter Web Service that can support multiple counters.

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:counter="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/basic "
                  targetNamespace="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/basic ">
   <wsdl:types>
      <xs:schema/>
   </wsdl:types>
   <wsdl:message name="increaseMsg">
      <wsdl:part name="counterId" type="xs:positiveInteger"/>
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <wsdl:message name="getValueRequestMsg">
      <wsdl:part name="counterId" type="xs:positiveInteger"/>
   </wsdl:message>
   <wsdl:message name="getValueResponseMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <wsdl:portType name="counterPortType">
      <wsdl:operation name="increase">
         <wsdl:input message="increaseMsg"/>
      </wsdl:operation>
      <wsdl:operation name="getValue">
         <wsdl:input  message="getValueRequestMsg"/>
         <wsdl:output message="getValueResponseMsg"/>
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

Listing 4-2: Modified version of the Counter Web Services WSDL that supports multiple counters

4.3.2. Counter Grid Service

In OGSI-terms, a *Grid Service* is a general term referring to a template, a contract, an interface described in GWSDL and the Grid Service Instances that must adhere to it. The term Grid Service does not, strictly speaking, characterize a component that can be part of a Grid application (i.e., it does not refer to an agent that can receive operation requests). Instead, it is *Grid Service Instances* that are the logical recipients of operation requests.

In the OGSI world, a Grid Service Instance (GSI) has to be explicitly created (by a factory or by the hosting environment), assigned a *Grid Service Handle (GSH)* which uniquely identifies it, and then registered with a handle resolver so it can be found by other Grid Service Instances. While a GSH is a unique, abstract name for a Grid Service Instance, a *Grid Service Reference (GSR)* provides the endpoint-related information that is necessary to access the identified Grid Service Instance. Details of the steps that may be required for a Grid Service Instance to be discovered, created, and consumed are given in Section 5.4 (page 43).

Once a Grid Service Reference (GSR) to a Grid Service Instance is available to a consumer, the GSI can be used to provide functionality that is equivalent to its Web Service counterpart. However, the OGSI-specific characteristics of a GSI can be used to provide richer and more consistent solutions.

The Counter GSI of Figure 4-7 is equivalent to the Counter Web Service presented in Section 4.3.1. Different service consumers can invoke operations on the same Counter GSI in order to change or access the state of the counter. This example differs from the Web Service version in that the state is logically attached to the instance. The semantics of a Grid Service Instance, as defined by OGSI, specify that state has to be maintained between consumer-service interactions. This is done in an implementation-specific way (i.e., the OGSI standard does not talk about *how* state is to be maintained).

Figure 4-7: An increase operation to a Counter Grid Service Instance

In contrast to the Counter Web Service, each Counter Grid Service Instance is associated with lifetime properties that can be used to manage the lifecycle of a counter. Also note that the OGSI Specification does not deal with the semantics of concurrent access to the same Counter Grid Service Instance by multiple clients. Hence, the consistency semantics of state related information are application-specific.

The GWSDL interface of the Counter Grid Service looks very similar to its Web Service equivalent. Apart from the obvious namespace change, the only other difference is the introduction of portType extension using the "extends" attribute (Listing 4-3). This indicates to GWSDL processors that the `counterPortType` extends the functionality defined by the `ogsi:GridService` portType.

```
<wsdl:definitions
xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:counter="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/basic"
                  targetNamespace="http://www.gridforum.org/namespaces/
```

```
2003/05/ogsiprimer/counter/basic">
   <wsdl:types>
      <xs:schema/>
   </wsdl:types>
   <wsdl:message name="increaseMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <wsdl:message name="getValueMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <ogsi:portType name="counterPortType" extends="ogsi:GridService">
      <wsdl:operation name="increase">
         <wsdl:input message="increaseMsg"/>
      </wsdl:operation>
      <wsdl:operation name="getValue">
         <wsdl:output message="getValueMsg"/>
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

Listing 4-3: GWSDL document for a basic Counter Grid Service

Once a Grid Service Instance that adheres to the above GWSDL is created, it behaves in exactly the same way as the Counter Web Service but it offers the additional functionality defined by OGSI. The GWSDL of a Grid Service Instance, of course, contains the necessary bindings and service elements, as required by WSDL.

The implicit association between a GSI and its state allows us to replicate the functionality of the Multiple-Counter Web Service without making significant changes to the interface. All that is required is a new Grid Service Instance. One could argue that a new Web Service identical to the one of Figure 4-6 could be deployed if multiple counters were required, while still keeping the same interface. However, OGSI encourages the instantiation of transient Grid Service Instances to represent state, so it is more natural to represent each separate counter as a separate Grid Service Instance. The OGSI model encourages, but does not mandate, a one-to-one association between a resource, in this case the counter, and a Grid Service Instance. By adopting this approach we get the additional benefit of lifetime and identity management for resources, as specified by OGSI.[1]

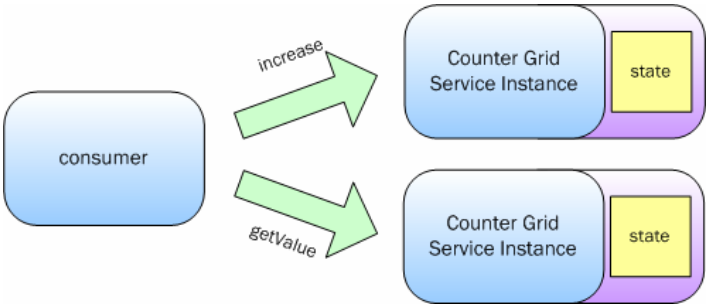

Figure 4-8: Operations to two different Counter GSIs representing a counter each

In addition to state management, Grid Service Instances can provide access to information through attribute-like constructs, called Service Data Elements (SDEs). An SDE is declared in

[1] The process by which Grid Service Instances can be created, registered, and discovered is discussed in detail in later chapters.

the Grid Service's interface and accessed through operations defined by OGSI's `GridService` portType.

The Counter Grid Service may expose the value of the counter through an SDE, removing the need for the explicit `getValue` operation. The GWSDL of the Counter Grid Service with an SDE is shown in Listing 4-4 below:

```
<wsdl:definitions
xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
                  xmlns:sd="http://www.gridforum.org/namespaces
/2003/03/serviceData"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
                  xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:counter="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/basic"
                  targetNamespace="http://www.gridforum.org/namespaces/
2003/05/ogsiprimer/counter/basic">
   <wsdl:types>
      <xs:schema/>
   </wsdl:types>
   <wsdl:message name="increaseMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
      <wsdl:message name="getValueMsg">
      <wsdl:part name="value" type="xs:positiveInteger"/>
   </wsdl:message>
   <ogsi:portType name="counterPortType" extends="ogsi:GridService">
      <wsdl:operation name="increase">
         <wsdl:input message="increaseMsg"/>
      </wsdl:operation>
      <sd:serviceData name="counterValue"
                      type="xs:positiveInteger"
                      minOccurs="1"
                      maxOccurs="1"
                      mutability="mutable">
         <sd:documentation>
            The value of the counter.
         </sd:documentation>
      </sd:serviceData>
   </ogsi:portType>
</wsdl:definitions>
```

Listing 4-4: GWSDL for a simple Counter Grid Service with an SDE

It is now possible to access the counterValue SDE through operations specified by the `GridService` portType (Figure 4-9).

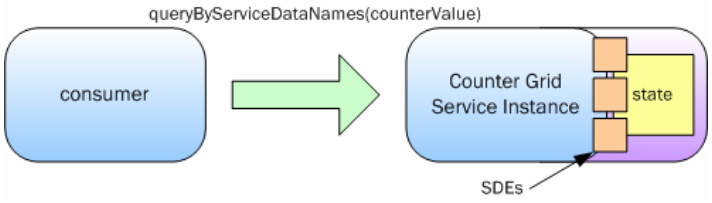

Figure 4-9: Consumer requesting the value of the counter through the counterValue SDE

Note that OGSI says nothing about *how* the value of an SDE is maintained, if indeed it is maintained at all – for example, a business process associated with an SDE could calculate the value of the SDE through some application-specific logic, or a Grid Service that reports

temperature might read the value directly from the temperature sensor each time a client accesses the SDE.

5. The Concepts of Grid Services

This chapter introduces more thoroughly the ideas that are essential features of Grid Services. These are:

- The use of 'extension' to create complex portTypes from simpler, ones, how this is enabled by Grid Service WSDL and the behaviour every Grid Service must have as a result of extending the basic OGSI GridService portType.
- The Factory and Instance pattern.
- The lifetime of Instances.
- Grid Service References.
- The Identity of a Grid Service Instance: Handles, References and Locators.
- Service Data.
- Extensibility of operations.
- Notification.
- Faults.
- The basic services defined by OGSI: ServiceGroup, HandleResolver and Factory.

Chapters 6 and 7 describe how Grid Services can be implemented and what large scale features are needed to create and/or use services based in a Grid. More detailed explanations of the standard portTypes can be found in subsequent chapters, starting with the core GridService portType in Chapter 8.

5.1. Using portType Extension

Grid Service Descriptions can use portType extension to define standard operations through the use of the GWSDL 'extends' attribute. For example, the Counter Grid Service (Section 4.3.2) begins:

```
<ogsi:portType name="counterPortType" extends="ogsi:GridService">
```

This causes the inclusion of operations and ServiceData Declarations (SDD) from the `ogsi:GridService` portType. All Grid Service descriptions include these SDD definitions which are, in summary:

- The identity (called the Handle) of any Grid Service Instance which implements the description. This identity distinguishes an Instance, and the state associated with it, from all others.
- The termination time of the Instance. This can be set when the Instance is created and may sometimes be modified by users or managers of the Instance.
- The names of all the portTypes that the Service implements.
- The identity of the Factory that created the Instance (or xsi:nil if the instance was not created by a Factory).

The operations provided by the `ogsi:GridService` portType allow a client to query and modify the state reported by ServiceData, to specify the earliest and latest desired termination time and, finally, to destroy the Instance.

5.2. The Factory and Instance Pattern

The Factory design pattern is commonly used in Objected-Oriented software systems to enable the creation of multiple, similar artefacts. An OGSI Factory Service is a Grid Service that is used by a client to create other Grid Service Instances. When a client needs to create a new instance of a particular Grid Service it locates a corresponding Factory Service, invokes its `createService` operation, and receives a unique identifier that it can use to access the newly-created Instance. See Section 5.4 for information about Grid Service identifiers.

A Factory Service may choose to implement either the Factory portType defined by OGSI or another portType which serves some more specialized function. For example, the NotificationSource portType described in Section 5.7 and the ServiceGroupRegistration portType both create new Instances, so they follow the Factory pattern and, by definition, any Grid Services that implement them are Factory Services.

5.3. Lifetimes of Instances

In many cases, the artefacts (the Grid Service Instances) that are created to accomplish a task are only needed for a limited time. They may represent physical resources such as storage allocations, processor and bandwidth reservations, or the right to use sensors and databases. In such cases, an explicit lifetime can be associated with an Instance when it is created.

Requesters may sometimes need to be aware of the lifetimes of the Instances they use or create. If the initial lifetime proves insufficient, they may periodically prolong the lifetime of an Instance by extending its terminationTime. Often, the lifetime will be significant in the case of any failure in the progress of a task. In that case, when their lifetimes expire, the Instances can be safely destroyed, and the resources they represent can be recycled. This convention is essential in a large-scale, distributed system such as a Grid, where it is not possible to take into account all the possible consumer dependencies when sweeping up Service Instances.

5.4. Identity: Handles, References and Locators

5.4.1. Grid Service Handles

When a Grid Service Instance is created, it is given a globally-unique identity, known as a Grid Service Handle (GSH), which can be used by consumers to refer to the state associated with that Instance.

A GSH is a standard Universal Resource Identifier (URI) – it indicates how to *locate* the Instance, but not how to *communicate* with it. Before the GSH can be used, it must be resolved into a Grid Service Reference.

5.4.2. Grid Service References

Although a Grid Service Handle uniquely identifies an Instance, the Handle by itself does not provide all the information needed to allow the Requester to send messages to the Instance. What is needed is a description of a binding that can be used to translate application-level operations into network messages. In OGSI, these Handle-specific bindings are called Grid Service References (GSRs), and they can be described in WSDL. A Reference is created by passing a Handle to a special service called a Handle Resolver.

In the case of the Counter Service example, a GSR is simply the WSDL description of the Service with the service, port and binding elements of the information fully specified,

including an address to which the client should send requests. These components are summarised in Figure 4-4. To be useful to a client, the binding description must be recognised by the client-side infrastructure, as described in Section 7.1, which allows the infrastructure to construct and transmit the message containing invocation parameters provided by the client application.

The OGSI Specification also allows GSRs which are not described by WSDL. This facility allows distributed programming architectures such as CORBA [8] to be used without change yet still within the framework of OGSI services. In the case of CORBA, the GSR would be a CORBA Object Reference which is recognisable to clients using CORBA infrastructure. The means by which such a client retrieves a non-WSDL GSR from a Handle Resolver and the way it is subsequently processed is beyond the scope of this Primer.

Unlike a Handle, which is unique for all time, a Reference is a *temporary* identifier: it has a limited lifetime, and should be refreshed periodically if the client has a long-term need to use the associated Instance. The Reference's goodUntil property indicates its expiration time, and while it may continue to work after that time the client should refresh it to ensure that the most appropriate binding is being used. If the Reference has become invalid – which may happen if the Instance has been relocated to a different server – any attempt to use it will result in a failed operation.

A single Handle may resolve to several different References, each with a different set of characteristics. If so, the client can choose which one to use. For example:

- A Reference that enables a high-performance binding might be available to clients that are local to the Instance, while clients running on remote systems have to use a lower-performance network binding.
- A client that is local to the Instance may use a Reference representing a binding that describes unencrypted transmission, while remote clients are required to use encryption.
- If multiple copies of a single Instance exist, the Handle Resolver may return a separate Reference for each copy. Service Instance copies are explained in Section 5.4.5.

Many handle resolution schemes are possible; the OGSI Specification does not restrict or mandate any particular one, and allows multiple schemes to be used in parallel. For example, a client may prefer a Handle described by a secure resolution scheme such as SGNP [15] to safeguard the integrity of Reference which is returned.

5.4.3. Passing Handles and References: the Grid Service Locator

Handles and References may be passed to or returned from Grid Service operations, and it is often convenient to pass a *collection* of References and Handles so that the eventual user of the target Instance (a client application or its supporting middleware) can choose the most appropriate way of calling it. OGSI defines the Grid Service Locator as a container for a collection of Handles and References: a single Locator may contain any number of Handles and/or References, and its recipient may choose to use them in any appropriate way.

For example, in response to a client's successful createService request, a Factory service returns a Locator that might contain several Handles and valid References. The client may store the Locator before using it and, when it is needed, discover that the references are no longer useable. It passes the Locator to a Handle Resolver Service, which returns an updated Locator containing the original Handles plus one or more new References. The client can choose whichever of the References it considers appropriate.

A Locator may also contain the name (a fully-qualified name in the XML namespace) of the interface for the target Service. This can be used by the recipient of the Locator to discover the interface description, to check that the Handles and References refer to implementations that are suitable for the client, and to allow the client infrastructure to enable access to the service.

5.4.4. Identifying Grid Service Instances

From the discussion above it should be clear that each Grid Service Instance is distinguished from all others by means of its Handles and References. Use of the same Handle to invoke (via resolution) operations on multiple occasions, even by different clients, must result in all of the operations acting on the same Instance.

However, a Grid Service can have multiple Handles and/or References, and a client may receive different versions from several sources as Locators are passed around. Comparing versions of these received from different sources is not useful as a way of identifying the target Instance as being the same one, as the next section illustrates. If a Service requires clients to be able to establish equivalence of several Locators, Handles or References, it must provide an operation to do this. The details of the comparison are dependent on the Service.

5.4.5. Service Instance Copies

A given Grid Service Instance is an association of a Service with some particular state – the value of a counter or the status of a physical resource, for example. If there is likely to be contention for the use of a particular Instance, it may be possible to improve performance by creating multiple copies of the same Instance, perhaps spread across multiple processors or servers. This is a service-oriented analogy of multi-threading a traditional program and is illustrated in Figure 6-4 on page 42.

For example, our Counter service may expect frequent access to each Counter Instance by multiple clients. One way of implementing the service is to distribute the load by establishing multiple copies of the same Instance, and resolving the Instance's GSH to multiple (different) references to those copies. Each reference is equivalent to all of the others – *provided* that the copies are correctly synchronised during updates, and that they always refer to the same state values. It is up to the Service designer to determine how best to synchronise operations and to determine if the Handle Resolver should pass all of the References to a particular client, allowing the client to choose which one to use, or to give each client only a Reference to a particular copy, perhaps on a round-robin basis for each request.

A service *may* also place additional information in its Handles to distinguish copies of the service from each other. The Counter Instance may provide different Qualities of Service (such as fast or slow response times) to different clients based on information in the Handle, yet still preserve the semantics of the Counter service across all clients of the Instance.

The semantics of the service description may, as in the case of the counter, require that the service moves through a series of well-defined states in response to a particular sequence of messages, thus requiring state coherence regardless of how Handles are resolved to References. However, other service descriptions may be defined that allow for looser consistency between the various members of the distributed service implementation. The copies of the Service Instance, identified by different Handles and References, may respond differently in some respects, yet still be the same Instance.

5.5. Introducing serviceData

The idea of stateful Grid Service Instances, which was introduced in Section 4.3, is based on the ability to assign a global identity and a lifetime to a piece of 'state' that is managed by the Instance. ServiceData is a way to describe some parts of the state. The Counter Service example introduced in Section 4.3.2 shows how a serviceData element (SDE) can be introduced to expose the value of the state – in this case, the counter. In general, a Grid Service can include declarations in its Service Description that expose to its clients as much or as little of its state as necessary.

The advantages of using serviceData are:

- Generic operations are provided to get and set the values of the SDEs and retrieve collections of values from the different SDEs.
- Interfaces are available that allow clients to receive notification of changes to the value of an SDE during the lifetime of a Service Instance (see Section 5.7).
- SDEs may have lifetimes, which can help to qualify the validity of information in a distributed system.

ServiceData is designed to support the general requirements for information management in a Grid, including requirements for the discovery, introspection, monitoring and management of Grid Service Instances. SDEs can provide descriptive information about an Instance, including both meta-data (information about the Service Instance, such as the interfaces it supports and the way it is configured) and state data (runtime properties of the Instance, such as the counter value).

5.5.1. The Structure of serviceData

The serviceData Declarations in a Service Description define the structure of an XML document which is delivered to a client in response to a query operation, to report the values of the serviceData Elements. However, an implementation of OGSI may provide a query API that returns values in their natural forms – rendering integers as binary values, for example – so that the client never has to deal directly with the XML.

The returned report document has a 'serviceDataValues' element at its root, and each serviceData Declaration (SDD) in the Service Description describes the structure of subsidiary elements of the report. For example, the declaration for the counterValue SDE from Section 4.3 is as follows:

```
<sd:serviceData name="counterValue"
                      type="xs:positiveInteger"
                      minOccurs="1"
                      maxOccurs="1"
                      mutability="mutable">
        <sd:documentation>
           The value of the counter.
        </sd:documentation>
      </sd:serviceData>
```

Part of the structure of the XML document generated by the Counter Service to report the value of the SDEs is shown graphically in Figure 5-1.

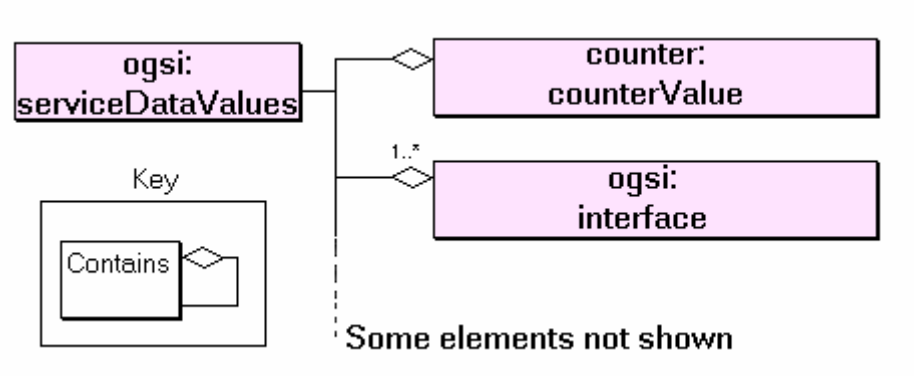

Figure 5-1: Example serviceDataValues

The declaration of the counterValue SDE causes the inclusion of one element named 'counter:counterValue' which contains an integer. Since the counter service extends the basic GridService portType, all of the SDEs declared in GridService are also present in the counter. These include the 'osgi:interface' SDE, whose value consists of one or more elements each of which contains the name of an interface (portType) implemented by the Instance. This, and the other SDEs defined by the GridService portType, are described in Chapter 8.

To be clear about the terminology, note that the singular 'value of an SDE' may actually consist of several 'SDE value elements'. For example, in the following fragment, the value of the 'interface' SDE consists of two value elements with values 'ogsi:GridService', and 'ns2:counterPortType'.

```
<ogsi:serviceDataValues ...>
     <ogsi:interface>
          <value>ogsi:GridService</value>
     </ogsi:interface>
     <ogsi:interface
         <value>ns2:counterPortType</value>
     </ogsi:interface>
</ns1:serviceDataValues>
```

The serviceData Declaration may use attributes to describe the structure of the serviceData values. Some of these attributes parallel those of XML schema elements because, like XML declarations, they describe the structure of XML documents. The following attributes may be associated with a given SDE:

- **name** and **type**.
- **minOccurs** and **maxOccurs.** These constrain the number of occurrences of the element. Simple values such as the counterValue are singletons (minoccurs=1 and maxoccurs=1). Compound values such as ogsi:interface consist of an unordered collection of elements.
- **nillable** (true or false). This can provide a special 'nil' value for the element. The meaning of this nil depends on the particular SDE.
- **modifiable**. If false, the serviceData element should be regarded as 'read only' by the requester, though its values may change as a result of other operations on the Service's interface, or simply because the underlying state of the service changed. If true, the setServiceData operation may be used to modify the value.

Note that in the Counter Service example the counterValue is technically not modifiable, because its value is not changed via the setServiceData operation, but via the 'increase' operation. The differences in the semantics of these operations are discussed in Section 5.5.4.

- **mutability**. This describes the way in which the serviceData is initialised, and whether its value can change during the lifetime of the Instance. An SDE declared with a mutability attribute of '**static**' is initialized with a value that is specified in the WSDL description, while a '**constant**' declaration indicates that the value will be initialised by the Service Instance when it is created. In both of these cases the SDE keeps the same value for the lifetime of the Instance.

  An '**extendable**' SDE can have new value elements added during the lifetime of the service, but none of them can be subsequently changed or removed.

  A '**mutable**' SDE has no constraints on its value elements: any and all may be changed or removed and new ones may be added at any time – provided the other constraints such as maxoccurs are still valid.

In the Counter Service example, the 'counter:counterValue' is mutable, because its value changes over time. The 'ogsi:interface' SDE is constant.

5.5.2. Dynamic Addition and Removal of serviceData Elements

Many SDEs are most naturally defined in a Service's interface definition can be supplied with initial values that are also included with the definition. For example, the initial value of the static findServiceDataExtensibility SDE in the ogsi:GridService portType is defined by:

```
<sd:staticServiceDataValues>
<findServiceDataExtensibility inputElement="ogsi:queryByServiceDataNames"/>
</sd:staticServiceDataValues>
```

However, situations can arise where it is useful to add or remove serviceData elements during the lifetime of an Instance. The means by which such updates are achieved is not defined by OGSI; for example, a Service Instance may implement operations for adding a new SDE. The GridService portType includes SDEs that allow a client to discover a Service's serviceData definitions dynamically, and to find their values.

5.5.3. The Validity of serviceData Element Values

ServiceData Element values may carry attributes defining their period of validity. These are:

- ogsi:goodFrom: Declares the time from which the value is valid. This is typically the time at which the value is copied into a report document or query response.
- ogsi:goodUntil: Declares the time until which the value is valid.
- ogsi:availableUntil: This declares the time until which the serviceData Element itself is expected to be available, perhaps with updated values. After this time, a client may no longer be able to get a copy of this element.

When supplied, these attributes are intended to be useful to the client, but the Service is not required to supply them, and the client is free to use or ignore them as it sees fit. For example, a Service that reports the temperature of a device might poll the device's temperature sensor once every minute. When a client requests the temperature, the Service might report the temperature value with a goodUntil value that corresponds to one second before the time of

the next reading. With knowledge of the Service's semantics the client might then store the value, and continue to use it until the value has expired, reasoning that if it were to make another request before that time it would receive the same value.

5.5.4. ServiceData Operations

The OGSI GridService portType defines operations which 'find' and 'set' the state reported by the serviceData Elements. The form of the operation and the relationship of the SDE values and the state were introduced in Section 4.3.2, and are shown in Figure 5-2 .

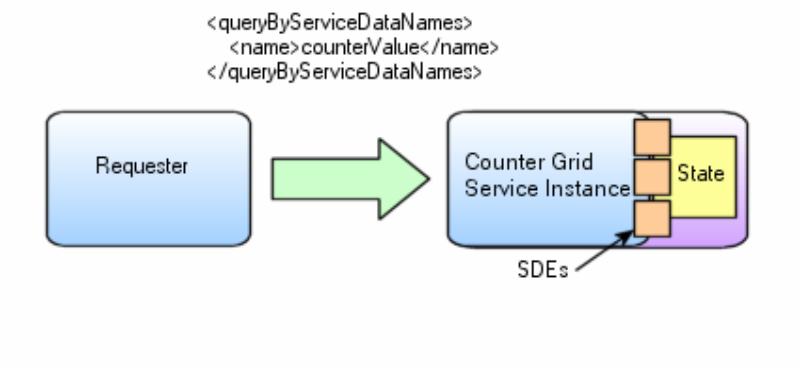

Figure 5-2: ServiceData Operations

Here, the Counter value is defined as an SDE using the XML in Section 4.3.2. A query operation uses the name of the SDE to select the value, and the Service returns a document similar to the following:

```
<ns1:serviceDataValues ...">
     <ns3:CounterValue ...
          xsi:type="ns2:CounterValueType"
          goodFrom="2002-04-27T10:20:00.000-06:00">
         <value xmlns="">111</value>
     </ns3:CounterValue>
</ns1:serviceDataValues>
```

The 'findServiceData' operation produces a report of the values of SDEs associated with the state of the Instance, and has two significant features. Firstly, the syntax of the operation which selects the values to be returned may be defined by the Service. OGSI defines default syntax, illustrated in Figure 5-2, which contains the names of the SDEs that are required, but each Grid Service can define its own syntax to provide greater power in the selection of reported values. This is an example of 'operation extensibility', which is described in more detail in Section 5.6. Secondly, the values that are reported need not be from the same point in time. Each element of the report may carry its own 'goodFrom' and 'goodUntil' attribute to indicate the freshness and longevity of the value. Even when times are not provided, the consumer of the report should make no assumptions about the coherence of the values.

The 'setServiceData' operation has the same features as 'findServiceData'; it also has an input syntax which is extensible and which can be defined by the Grid Service with defaults, defined by OGSI, called setByServiceDataNames and deleteByServiceDataNames. These systems use a collection of SDE names and values, and the values are used to update the state of the Instance. In the default systems, the client cannot assume that the replacement of any service data elements is carried out with any coherency: the way in which multiple values in the input data are treated, and their effect on the state of the Instance and on subsequent

findServiceData operations, is defined as if each value is independent of all others, and independent of other processes which may be updating or querying the state of the Instance.

As an example, consider the effect of defining the counterValue SDE of the Counter Grid Service with the attribute 'modifiable=true'. This enables a setServiceData operation on an instance of the Counter Grid Service.

```
<ogsi:setByServiceDataNames>
<counterValue> 100 </counterValue>
</ogsi:setByServiceDataNames>
```

The implementation of the Service may store the new value in a temporary buffer before applying it to the underlying state of the Instance. A findServiceData operation which is invoked just after the completion of the setServiceData may report a counterValue that is different from 100 because the state has not, at that time, been updated. In the case of the Counter Service, the `increase` and `getValue` operations are provided to manipulate the counter in a synchronized way.

5.6. Extensibility of Grid Service Operations

Several OGSI operations accept an input argument that is an untyped extensibility element. This allows for a general pattern of behaviour to be expressed by the Service Description, including a specific default refinement of the pattern, while allowing for other refinements, via extensions, which may be appropriate to particular services or tasks. An extensible operation is a useful way to define interfaces, such as the OGSI portTypes, which are intended to be extended by more concrete and specific portTypes. In order to allow a client to discover the extensions that are supported by such an operation, OGSI defines a common approach for expressing extensible operation capabilities via static serviceData values.

The findServiceData and setServiceData operations described in Section 5.5.4 and the NotificationSource::subscribe operation described in Section 5.7 are all examples of extensible operations. An extensible operation may contain one extensible input parameter, which can contain an XML element of a type described by the values of an SDE associated with the operation name. For example, the input arguments for the setServiceData operation are described by the serviceData values in the GridService portType:

```
<ogsi:setServiceDataExtensibility
 inputElement="ogsi:setByServiceDataNames" />

<ogsi:setServiceDataExtensibility
 inputElement="ogsi:deleteByServiceDataNames" />
```

These declarations provide two initial value elements for the SDE named setServiceDataExtensibility, which can have any number of additional values, and has a mutability attribute of 'static'. The two values permit two choices for input to the operation, and the service determines which kind of input is being used from the name attribute of the XML element used when the input message is received.

5.7. Introducing Notification

Notification is a common software messaging style that allows clients (sinks) to be notified of changes that occur in a service (source). For example, instead of a client periodically asking a Counter Grid Service Instance if there are any changes to its counter value (a polling approach), it makes one single call to the Service Instance, requesting to be notified whenever a state change occurs. From then on the Counter Instance will contact the client as soon as any

change occurs, and send it a notification message that may specify the type of change and/or information relating to the changed state.

The notification framework allows for asynchronous, one-way delivery of messages from a source to a subscribed sink. Any service that wishes to support subscription of notification messages must support the NotificationSource interface. OGSI allows for notification on the service data elements of a Grid Service Instance. As part of its serviceData, the NotificationSource maintains a set of SDEs to which a requester may subscribe for notification of changes. If notification on change of implementation-specific internal state is desired, then additional SDEs may be defined for that purpose.

To start notification from a particular service a client must invoke the subscribe operation on the notification source interface, giving it the Handle of the notification sink. A subscription request also contains a subscription expression that is an XML element that describes what messages should be sent from the source to the sink, as well as when messages should be sent, based on changes to values within a Service Instance's serviceData values.

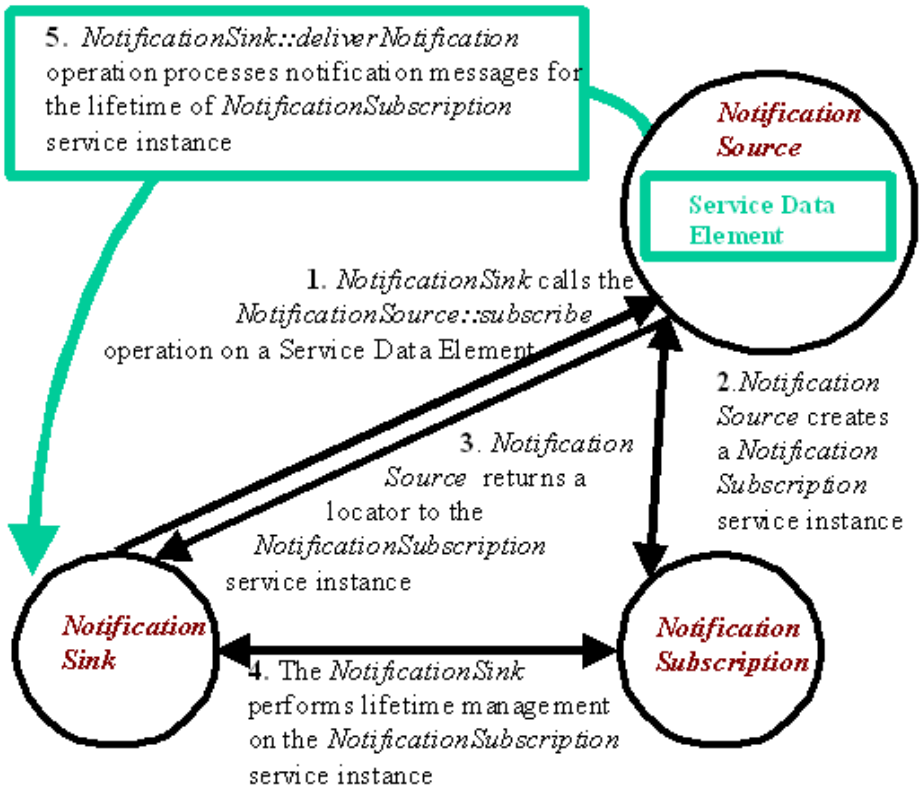

Figure 5-3: Typical Notification scenario

As illustrated in Figure 5-3, a subscription request causes the creation of a Subscription Grid Service Instance that can be used by the client to manage the subscription, and to discover its properties. A Locator to this Subscription Instance is returned as part of the output from the subscribe operation, as well the currently-planned termination time.

Following a successful subscription, a stream of notification messages then flows from the source to the sink as changes occur to the subscribed SDE's, until the subscription is either explicitly destroyed or is timed out.

For example, in the last Counter example from Section 3.2, the value of the counter is exposed through an SDE and is accessed through the queryByServiceDataNames operation (pull model). With notification support, the Counter Instance will automatically notify all sinks (subscribed clients) whenever changes occur to its value. The following example adds notification support to the counter example from Section 3.2

```
    <ogsi:portType
       name="NotificationcounterPortType"
       extends="counter:CounterPortType ogsi:NotificationSource"/>
```

Any client that is interested in receiving notification only has to support the NotificationSink portType. That is, a Web Service is not required to also implement the GridService portType in order to act as a notification sink. This enables the notification clients to live outside an OGSI hosting environment and still subscribe to notification messages.

The OGSI Notification framework defines one simple type of subscription, as defined by the subscribeByserviceDataNames operation extensibility. For a subscribeByServiceDataNames subscription, the notification message that is sent from the source to the sinks is simply a serviceDataValues element.

A client (sink) that uses a subscribeByServiceDataNames to the CounterValue SDE of a NotificationCounter Instance will receive a serviceDataValues element similar to the one below whenever the value (111 in the example) in the CounterValue SDE is changed.

```
<ns1:serviceDataValues ...">
     <ns3:CounterValue ...
         xsi:type="ns2:CounterValueType">
         <value xmlns="">111</value>
     </ns3:CounterValue>
</ns1:serviceDataValues>
```

5.8. Operation Faults

A common procedure is used for handling operation faults generated by Grid Services. This simplifies problem determination by having a common base set of information that all fault messages contain. It also allows for "chaining" of fault information up through a service invocation stack, so that a recipient of a fault can drill down through the causes to understand more detail about the reason for the fault. OGSI defines a base type (ogsi:FaultType) for all fault messages that Grid Services must return. This approach simply defines the base format for fault messages, without modifying the WSDL fault message model. All Grid Service operations must return the ogsi:fault (which is of type ogsi:FaultType) in addition to any operation-specific faults. For example an application-specific fault (myFault) could be included in the definition as follows

```
<wsdl:definitions ...>
   <types>
      <xsd:schema ...>
         <xsd:complexType name="MyFaultType">
            <xsd:complexContent>
               <xsd:extension base="ogsi:FaultType"/>
            </xsd:complexContent>
         </xsd:complexType>
```

```
         <xsd:element name="myFault" type="tns:MyFaultType"/>
      </xsd:schema>
   </types>
   <message name="myFaultMessage">
      <part name="fault" element="tns:MyFault"/>
   </message>
   <ogsi:portType ...>
      <wsdl:operation ...>
         <input ...>
         <output ...>
         <fault name="myFault" message="tns:myFaultMessage"/>
         <fault name="fault" message="ogsi:faultMessage"/>
      </wsdl:operation>
   </ogsi:portType>
</wsdl:definitions>
```

A Grid Service operation may return a more refined fault (i.e., an extension derived through the XML schema mechanisms) in place of a particular fault that is required by an operation definition. For example, if an operation is specified to return a fault with the element myFault under some circumstances, then a particular implementation of that operation may return an extension of myFault in its place. This should be done by returning the myFault message with an xsi:type of the more refined fault.

It is recommended that all faults from a Grid Service either use the FaultType directly or use an extension of FaultType. However there may be circumstances in which faults from a Grid Service are not derived from FaultType, such as when the Grid Service uses legacy portTypes in addition to OGSI portTypes.

5.9. Basic Services and Their Roles

Figure 5-4 shows an interaction between components defined by OGSI that is a typical pattern for the use of Grid Services. The Services are introduced in the subsections which follow, and are described in more detail in subsequent chapters.

The starting point for the interaction is that the Requester is an end user or Web Server acting on behalf of end users, and has installed applications which need to make use of remote services. The Requester has configuration information which provides references to a Registry Service.

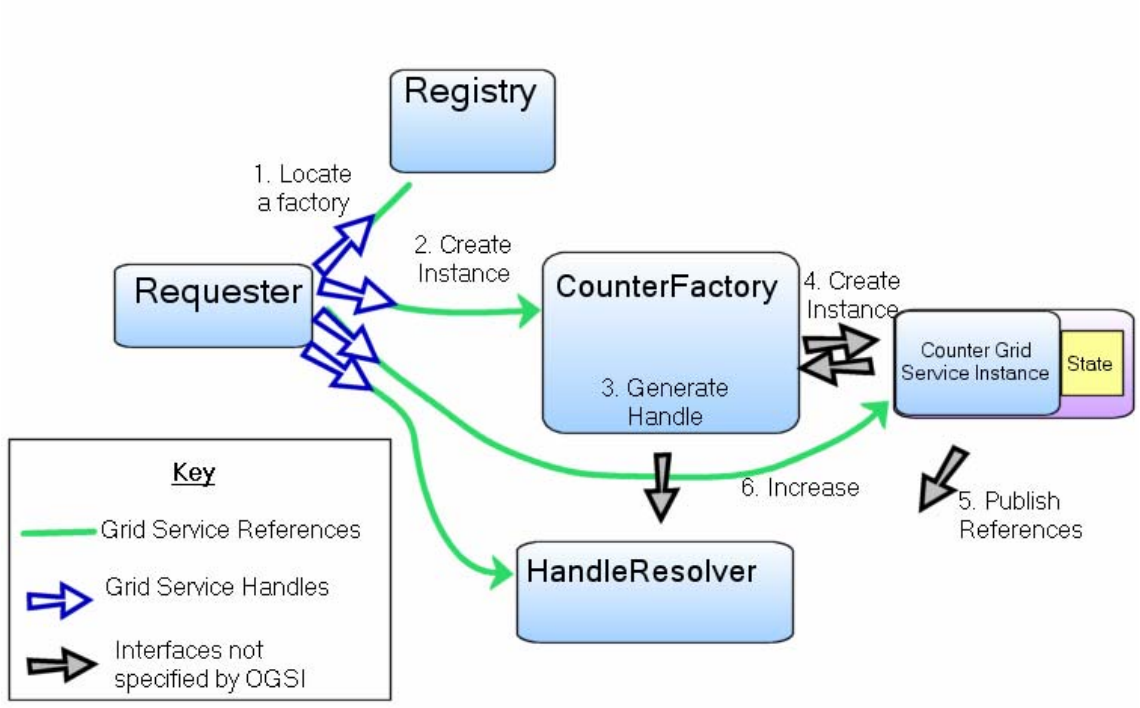

Figure 5-4: Main Components of OGSI Services

5.9.1. The Registry Service

Although OGSI does not specify a Registry Service, a Registry can be described by using or extending the ServiceGroup or ServiceGroupRegistration portTypes. A Registry contains references to other Services, and a Requester can search for a Service that it needs by supplying the Registry with details of the Interface that it needs. The Registry responds with details of all the Services it knows about that match the search criteria.

For example, a Requester that needs to create a Counter Service may search for a CounterFactory portType and receive back information, including a Handle, which identifies the Factory.

5.9.2. The HandleResolver Service

It is convenient to the Requester for Services which supply Handles, such as Registry Services, to supply a WSDL document describing GSRs along with the Handle, but the GSRs can also be discovered from a HandleResolver.

A HandleResolver Service is a fundamental part of an OGSI Grid. Since Requesters need access to a HandleResolver before they can access any other service, they must be bootstrapped with information about which HandleResolver to use. This is typically done using configuration data.

The HandleResolver is used to retrieve up-to-date or otherwise improved References to a Service identified by a Handle. A Requester can use the HandleResolver at any time to obtain the best available binding.

5.9.3. The Factory Service

The Factory Service was described earlier, but is included here for completeness.

A Requester invokes a createService operation on a Factory, and receives in response Handles and/or References for the newly-created Instance. See Section 5.2 for more information about Factory Services.

6. Implementing Web and Grid Services

In this chapter, we examine the relationship between OGSI and the existing and developing Web Services framework (description languages, tools and implementation platforms) on which it builds and depends. We examine both the server-side programming patterns for Grid Services and several conceptual hosting environments for Grid Services. The patterns described in this chapter are enabled but not *required* by OGSI; we discuss them here to help put into context concepts and details described in the other parts of the Primer and in the Specification. In particular, the discussion of implementing robust services provides more information about the relationship between handles and references, and the discussion of replicated Service Instances answers some questions about the issue of identity.

6.1. Server-Side Programming Patterns

The OGSI Specification does not dictate any particular server-side implementation architecture. A variety of approaches are possible, ranging from implementing the Grid Service Instance directly as an operating system process to using a sophisticated server-side component model such as J2EE. In the former case, most, or even all, support for standard Grid Service behaviours (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 behaviours will be supported by the hosting environment and invoked as needed.

To illustrate a range of possibilities for implementation, and the features of the Specification that enable this range, we describe different hosting patterns of increasing complexity in the following sections.

6.1.1. Monolithic Service Implementation

In Figure 6-1 we depict a scenario where the entire behaviour of the Grid Service, including the demarshalling/decoding of the network message, has been encapsulated within a single executable. The *protocol termination* represents a communication component such as an http server; the Grid Service is implemented directly, using the servlet interface. Although this approach may be efficient, it requires the protocol handling to be performed individually by each Grid Service, with no opportunity for Grid Services to reuse existing functionality.

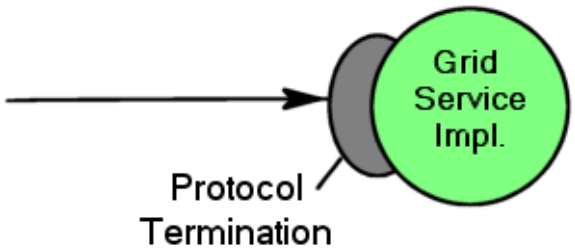

Figure 6-1: Simple monolithic Grid Service

6.1.2. Implementation within a Container

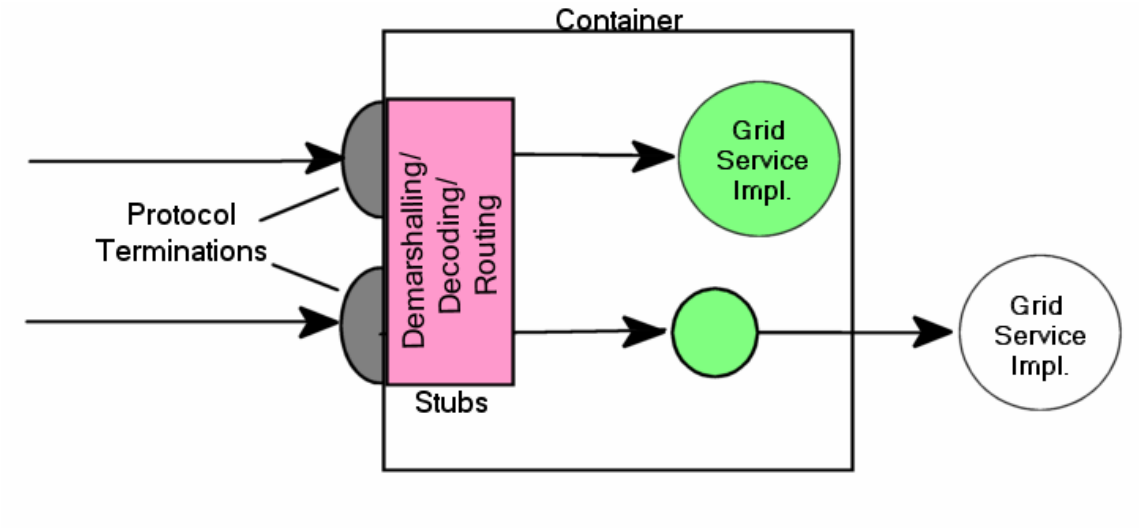

Figure 6-2: An approach to the implementation of argument demarshalling functions.

In Figure 6-2, the invocation message is received at a network protocol termination point (e.g., an http server, as is the case for many Grid Services). This converts the data in the invocation message into a format that can be handled by the hosting environment. We illustrate two Grid Service Instances (the shaded circles) implemented as container-managed components (for example Enterprise Java Beans (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 a native programming language. Demarshalling from protocol terminations to the native language is accomplished by generated components called *stubs* or *skeletons* . In some circumstances (the upper circle), the entire behaviour of a Grid Service is completely encapsulated within the component. In other cases (the lower circle), a component will collaborate with other server-side executables to delegate certain operations to standard components, or perhaps through an adapter layer designed to translate language invocation syntax, to complete the implementation of the Grid Service behaviour.

6.1.3. Container-managed State

A container implementation may provide a range of functionality beyond simple argument demarshalling. For example, it may provide lifetime management functions, automatic support for authorization and authentication, or request logging. It might also intercept lifetime management functions and terminate Service Instances when a service lifetime expires or an explicit destruction request is received. Thus, we avoid the need to re-implement these common behaviours in different Grid Service implementations.

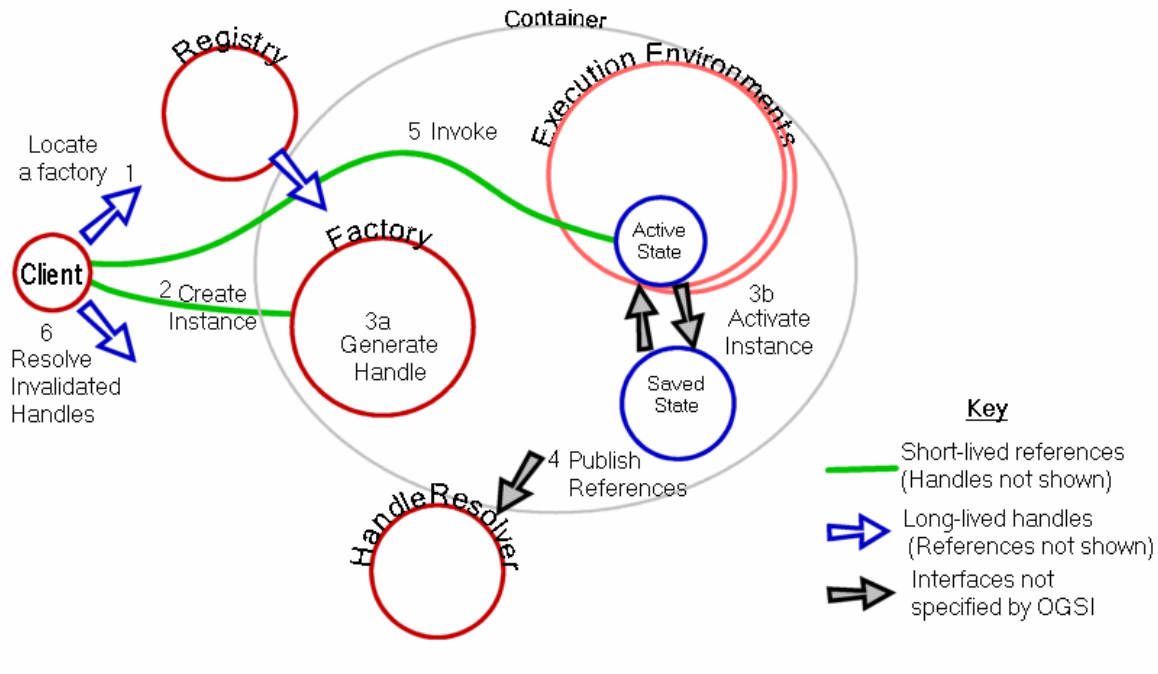

Figure 6-3: Container with state management

Figure 6-3 shows a container that is made up of multiple execution environments, any of which can be selected to host a Grid Service Instance. This is a possible implementation of the J2EE container described in the EJB specification [18]. In this implementation the state of the instance can be saved by the container at operation boundaries, and restored in case a Service Instance fails.

The diagram includes Grid HandleResolver and Registry services so that the order of events required to create and access the service can be seen. The client has a permanent reference to the HandleResolver and uses it to resolve service References that become invalid. The client also has a Handle to the registry service that it can use (1) to locate a suitable factory and, via the factory (2), create a Grid Service Instance (3a).

The container is responsible for allocating the execution environment for the instance, establishing the service Reference and supplying this to the HandleResolver (4), and initialising the service state and managing the store/load operations. The factory returns to the client a Locator containing a Handle and a Reference to the new service Instance that can be used to invoke service operations (5).

If the execution environment fails, the container may allocate a new environment with a new service Reference, load the state and supply the new Reference to the HandleResolver. The client uses the original Handle to obtain the updated Reference (6) from the HandleResolver.

6.1.4. Replicated Copies of a Service Instance

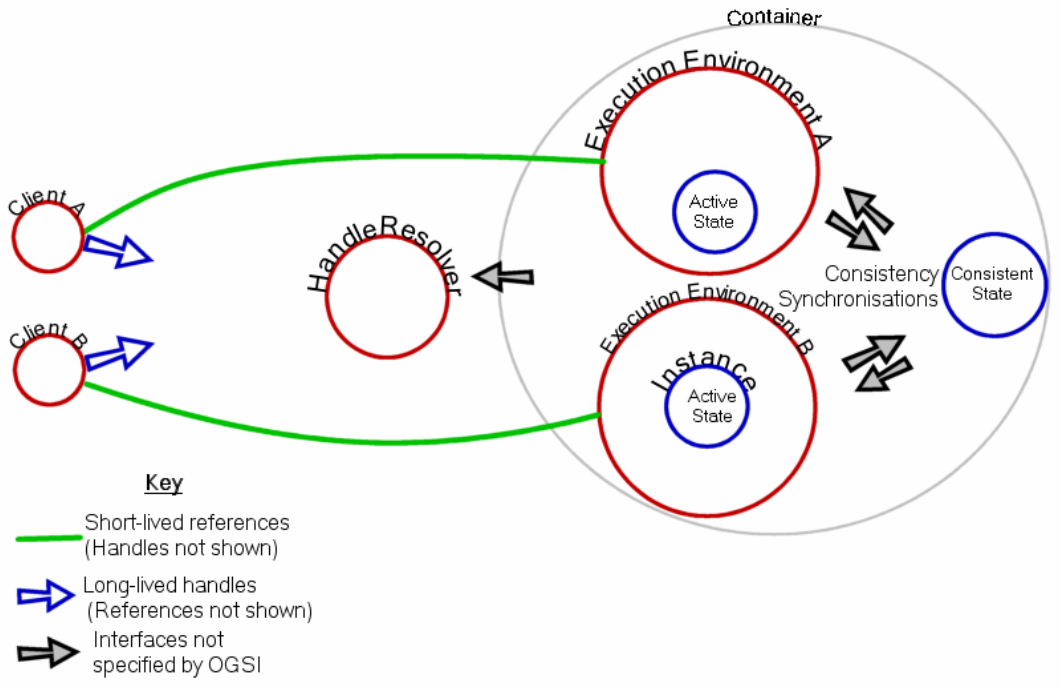

Figure 6-4: Replicated copies of a Service Instance

Figure 6-4 shows a container hosting multiple copies of a single Grid Service Instance in multiple execution environments. The environments may be hosted on different servers and therefore may provide scalability for a service which needs to serve many clients concurrently. The Execution environments can be considered caches for the underlying state of the Grid Service Instance: they must synchronise their own active state with the underlying state to preserve consistency among the caches. Although each client has the same Grid Service Handle, the distribution of workload among the execution environments may be achieved by resolving the Handle to different References, each describing a different protocol endpoint attached to its own execution environment. The resolution is controlled by a Handle Resolver which is closely coupled to the Container, and therefore is aware of the load distribution amongst the execution environments

The events which synchronise a cache with the underlying, consistent state are dependent on the container and the semantics of the Grid Service. For example, the same Counter Service Instance may receive *increase* messages at two different copies of the active state. An accurate counter can be provided by synchronising the new value of the underlying state before any response to a *getValue* is issued. However, an alternative approach might be to provide an *approximateCounter* service by synchronising at fixed time intervals.

As this description of an approximateCounter shows, the semantics of the particular service defines the level of consistency that can be expected when operations are issued by two different clients. The OGSI Specification says nothing about this issue. Similarly, it says nothing about responses received by the same client from different copies of the same Grid Instance referenced by two Grid Service References. If it is an important feature of a service that a client be able to identify two references as providing consistent results, a specific operation must be provided to confirm the equivalence of the References.

7. Using Grid Services

This chapter introduces the following concepts that are needed in order to use the Service Instances described by the previous chapter:

- Locating services and communicating with them.
- Ways in which collective information is used.

We examine the client-side programming patterns for Grid Services. The patterns described in this section are enabled but not required by OGSI. We discuss these patterns in this section to help put into context concepts and details described in the other parts of the Primer and in the Specification. Ways of locating Grid Service Instances, example use of a simple registry of factories, and the initial information needed when starting to use a Grid is also examined.

7.1. Client-Side Programming Patterns

An important issue that requires some explanation, particularly for those not familiar with Web Services, is how OGSI interfaces are likely to be invoked from client applications.

OGSI exploits an important component of the Web Services framework: the use of WSDL to describe abstract interfaces and, separately, multiple protocol bindings, encoding styles, messaging styles (RPC vs. document-oriented), security tokens and so on, for a given Web Service. The *Web Services Invocation Framework* [WSIF] and *Java API for XML RPC* [JAXRPC] are examples of infrastructure software that provide the capability to separate interface from bindings and provide multiple bindings. In each case, the basic structure is similar, and it provides a client-side architecture that can also be used to access OGSI services.

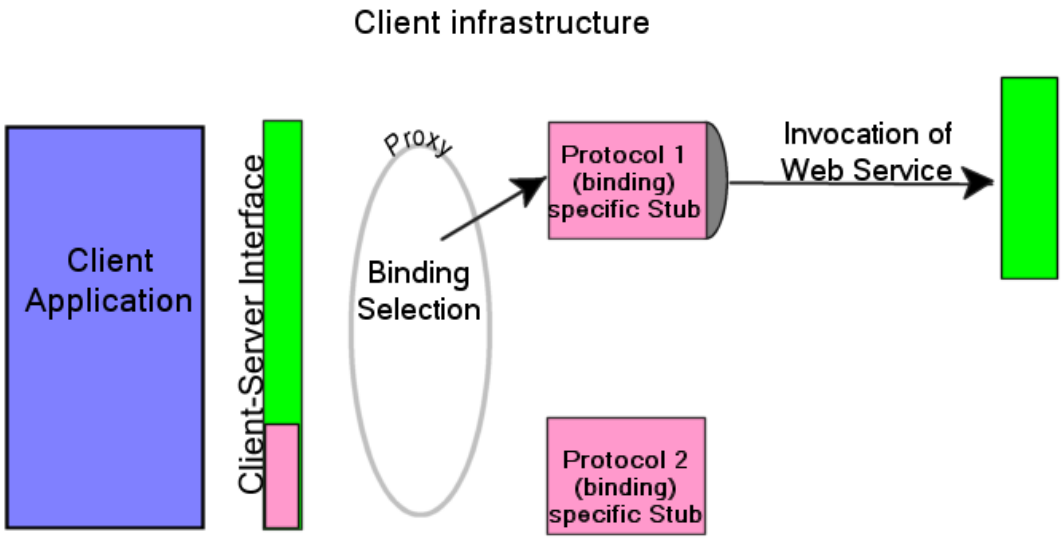

Figure 7-1: A client-side runtime architecture

In Figure 7-1, the client application accesses a client-side representation of the Web Service, sometime called a *proxy*. The proxy calls, or uses, components (shown as protocol-specific *stubs*) that marshal parameters for the invocation of the Web Service over a chosen binding,

and adds other necessary information such as protocol headers or security tokens obtained from the client's runtime environment.

There is a clear separation between the client application and the stub by the client-side interface. The interface comprises the operations, parameters and exceptions needed to invoke the service and to receive responses and handle any faults generated by the service. The client interface may also define exceptions that describe, in a generic way, errors arising from the binding mechanisms. These are dependent on the particular client infrastructure.

The client's interface to the service can be generated by taking the WSDL description of the Web Service interface and transforming it into interface definitions in a programming language specific way (e.g. Java interfaces or C#) suitable for the particular client-side architecture. The parameter marshalling and message routing stubs are generated from the various binding options provided by the WSDL. Generation of the interface and stubs can take place at various stages of development, deployment and execution of the client application: dynamic Web Service invocation leaves the generation of the binding and service address until execution time.

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.

It is possible, but not recommended, for developers to build customized code that directly couples client applications to fixed bindings of a particular service. Although certain circumstances demand potential efficiencies that can be gained by this style of customization, this approach introduces significant inflexibility into a system and therefore should only be used under extraordinary circumstances.

We expect that the stub and client side infrastructure model that we describe will be a common approach to enabling client access to Grid Service Instances. This includes both application-specific services and common infrastructure services that are defined by the OSGI Specification and described in this Primer. So, for most developers using Grid Services, the infrastructure and application level services appear in the form of a class library or programming language interface that is natural to the caller.

The WSDL and the GWSDL extensions required by OGSI can be supported by heterogeneous tools and enabling infrastructure software by means of translation of GWSDL into WSDL [23].

7.2. Reasons for Registries

Registries are sources of information about services in a Grid. Registries are themselves Grid Services, and their function is to provide references to other services, perhaps based on some selection criteria such as the portType, or the capacity for resources to be allocated for a particular task. The criteria are not determined by the OGSI Specification, but depend on the individual registry and the kind of information it provides. The following examples give a flavour of how registries and service information can be organized:

- Central directories are the starting point for enquiries about Grid Services and resources. Their locations may be well-known to potential clients and they may keep lists of more local registries.
- Partner Catalog registries [19] sit behind an organization's firewall and contain only approved, tested, and valid Web Service descriptions from legitimate business

partners. The business context and metadata for these Web Services can be targeted to the specific requester.

- Local directories can contain references to factories capable of creating services instances (including resource allocations, for example). Local information can contain more detailed, precise and/or dynamic information.
- Factories may keep a list of the Service Instances they have created, and allow themselves to be queried directly about those services.

Figure 7-2 shows the last two of the alternatives described above, as an example of the way information in registries can be related.

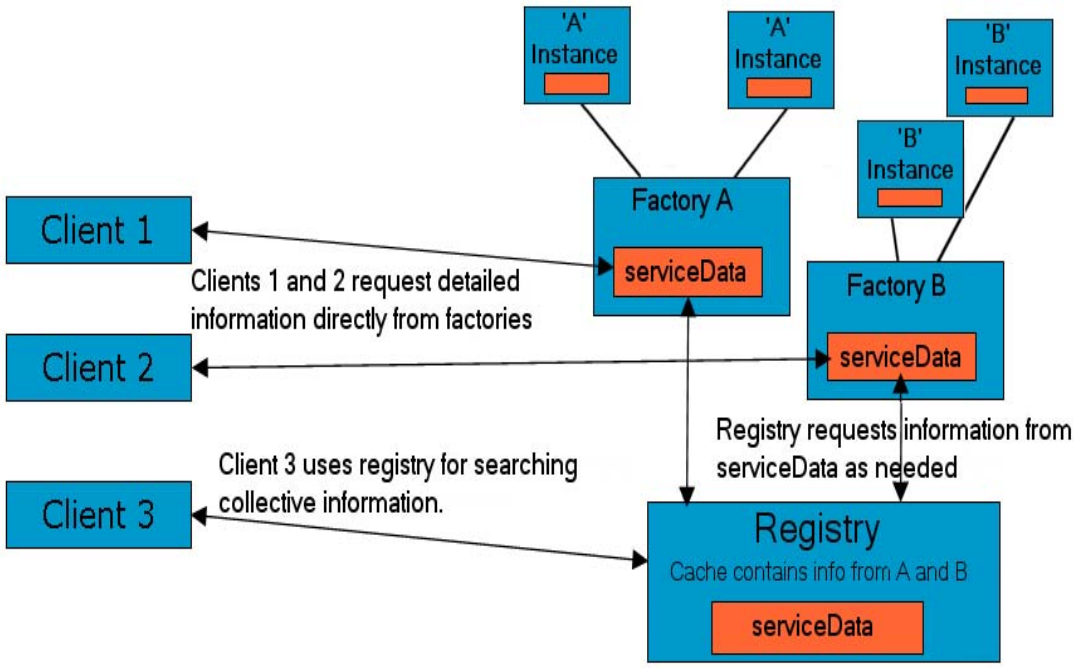

Figure 7-2: Factories and a dedicated Registry as information sources

The example of a factory which also acts as a registry shows how a registry can be just one of several functions provided by a given service.

7.2.1. The OGSI portTypes

OGSI defines three portTypes that may be used to build registries: ServiceGroup. ServiceGroupEntry and ServiceGroupRegistration.

A ServiceGroup is a collection of entries, where each entry points to a member Grid Service Instance as well as to a Grid Service implementing the ServiceGroupEntry interface that is used to manage the individual entries in the ServiceGroup (see Figure 7-3). Each Grid Service Instance belonging to a ServiceGroup must conform to (i.e. either be or be a subtype of) at least one of the portTypes listed in the membershipContentRule SDE. Each ServiceGroupEntry contains a Locator for the member Grid Service Instance and information (content) about that service. The content element is an XML element advertising some information about the member service. The content model forms the basis on which search

predicates can be formed and executed against the ServiceGroup, using the findServiceData operation.

The ServiceGroupRegistration portType extends the ServiceGroup portType and provides a management interface (add and remove operations) for a ServiceGroup. The add operation creates a ServiceGroupEntry and adds it to the ServiceGroup, while the remove operation removes each ServiceGroupEntry that matches an input expression.

More details about the three ServiceGroup portTypes can be found in Chapter 10.

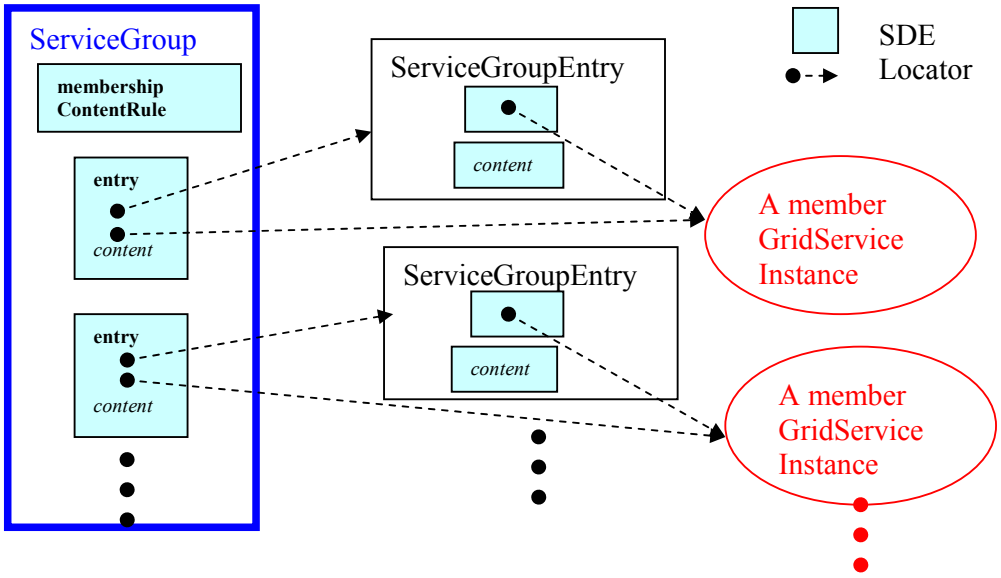

Figure 7-3: ServiceGroup structure and its relation to ServiceGroupEntry

7.2.2. Example: A Simple Registry of Service Factories

This example registry records the service factories currently running in one service container and is based on a *ServiceGroup* portType, not a *ServiceGroupRegistration* portType. It is a way of keeping track of current activity on a single server. Any kind of factory can be registered: the registered services need have nothing in common except what is required by the OGSI standard. The main metadata of interest (createServiceExtensibilityInfo in the example to follow) are the sets of portTypes provided by each Service Instance created by the factories. Hence, the metadata in this registry comes from the standard service data elements prescribed by OGSI for the Factory portType. The ogsi:CreateServiceExtensibility elements in a factory portType lists as QNames all the portTypes of the Service Instances that the factory can create.

For this example, suppose that there are just three factories, all for variants of a counter service:

- The basic counter factory which creates Instances that accept messages defined by the counter portType and the GridService portType.

- The "private" counter factory, whose Instances enforce access control for a restricted set of users, and which accepts the messages defined by the SecureCounter portType and the GridService portType.
- The "shared" counter factory, whose Instances are intended for concurrent use by more than one client, and which accepts messages defined by the Counter portType, the NotificationSource portType and the GridService portType.

As part of its entry serviceDataValues this registry (ServiceGroup) could contain the following values:

```
<!-- Specific values for this registry. -->
<ns:serviceDataValues xmlns:ns=…>
   <!-- Description of basic-counter factory. -->
   <entry xmlns:ogsi= …>
      <serviceGroupEntryLocator nil="true"/>
      <memberServiceLocator>
        <ogsi:handle>
          http://exemplar.org/ogsa/services/CounterFactory
        </ogsi:handle>
      </memberServiceLocator>
      <content>
         <createServiceExtensibilityInfo>
            <createsInterface>Counter</createsInterface>
            <createsInterface>GridService</createsInterface>
         </createServiceExtensibilityInfo>
      </content>
   </entry>

   <!-- Description of private-counter factory. -->
   <entry>
      <serviceGroupEntryLocator nil="true"/>
      <memberServiceLocator>
         <ogsi:handle>
            http://exemplar.org/ogsa/services/PrivateCounterFactory
         </ogsi:handle>
      </memberServiceLocator>
      <content>
         <createServiceExtensibilityInfo>
            <createsInterface>SecureCounter</createsInterface>
            <createsInterface>GridService</createsInterface>
         </createServiceExtensibilityInfo>
      </content>
   </entry>

   <!-- Description of shared-counter factory. -->
   <entry>
      <serviceGroupEntryLocator nil="true"/>
      <memberServiceLocator>
         <ogsi:handle>
             http://exemplar.org/ogsa/services/SharedCounterFactory
         </ogsi:handle>
      </memberServiceLocator>
      <content>
         <createServiceExtensibilityInfo>
            <createsInterface>Counter</createsInterface>
            <createsInterface>GridService</createsInterface>
            <createsInterface>NotificationSource</createsInterface>
         </createServiceExtensibilityInfo>
      </content>
```

```
      </entry>
  </ns:serviceDataValues>
```

Notes on the example above:

- Elements with the 'ogsi' prefix are defined in the OGSI standard.
- The serviceGroupEntryLocator elements are all annulled. The content of this registry is built using some unspecified operations defined by the service interface: it does not use the *add* operation from the ServiceGroupRegistration portType.
- The memberServiceLocator elements contain handle elements that give the GSHs of the registered factories. The given GSHs follow the HTTP scheme for GSHs used by Globus Toolkit 3; other forms of handles are possible.

A client can use the registry to select factories for particular kinds of counter as follows.

The client invokes findServiceData on the registry's GridService port. The client includes this as the queryExpression parameter of the invoked operation:

```
<ogsi:queryByServiceDataNames>
   <name>entry</name>
</ogsi:queryByServiceDataNames>
```

The result of the operation is a dump of all the metadata for all the services (all entry elements) as shown in the serviceDataValues document above.

The client runs these metadata through an XPath search-engine (an XSLT processor is a likely implementation of this) using this query:

```
//ogsi:entry[content/createServiceExtensibilityInfo/createsInterface="Notif
icationSource"]
```

to find all the factories for counters that do notification. The client is then left with one entry element:

```
<entry>
   <serviceGroupEntryLocator nil="true"/>
   <memberServiceLocator>
      <ogsi:handle>
        http://exemplar.org/ogsa/services/SharedCounterFactory
      </ogsi:handle>
   </memberServiceLocator>
   <content>
      <createServiceExtensibilityInfo>
         <createsInterface>Counter</createsInterface>
         <createsInterface>GridService</createsInterface>
         <createsInterface>NotificationSource</createsInterface>
      </createServiceExtensibilityInfo>
   </content>
</entry>
```

The client runs a further XPath search using query:

```
//ogsi:handle
```

to extract the GSH of the selected service.

7.3. Initial Service Discovery and Invocation

As we saw earlier, a client gains access to a Grid Service Instance through a Grid Service Handle (GSH) and Grid Service References (GSR). A Grid Service Handle can be thought of as a permanent network pointer to a particular Grid Service Instance, but does not provide sufficient information to allow a client to access the service; the client needs to "resolve" a GSH into a GSR.

The first points of contact for a client as it begins to use Grid Services are often basic services such as the registry, described above, and a HandleResolver, described in Section 5.9.2. In order to communicate with these services the client must either have GSRs to them, or have GSHs that can be resolved to GSRs.

The initial GSH or GSR for a registry can be supplied to the client in various ways, generally requiring some out-of-band mechanism defined by local conventions – for example, configuration during the installation of software components or use of a commonly available protocol such as ftp or http to retrieve the information from a central source.

In the case of a HandleResolver, an initial GSH is not an option since a GSR to the resolver would be needed to resolve the GSH. To avoid this circular dependency, the initial GSR for a HandleResolver must be provided by a similar kind of out-of-band mechanism as might be used for GSHs for other services.

8. The GridService portType

To be a Grid Service a service must implement the *GridService* portType, which defines common, basic message exchange patterns for:

- Asking a service to describe itself (Introspection)
- Managing the lifetime of a Service Instance

This chapter describes the operations and serviceData associated with the *GridService* portType.

8.1. ServiceData Elements

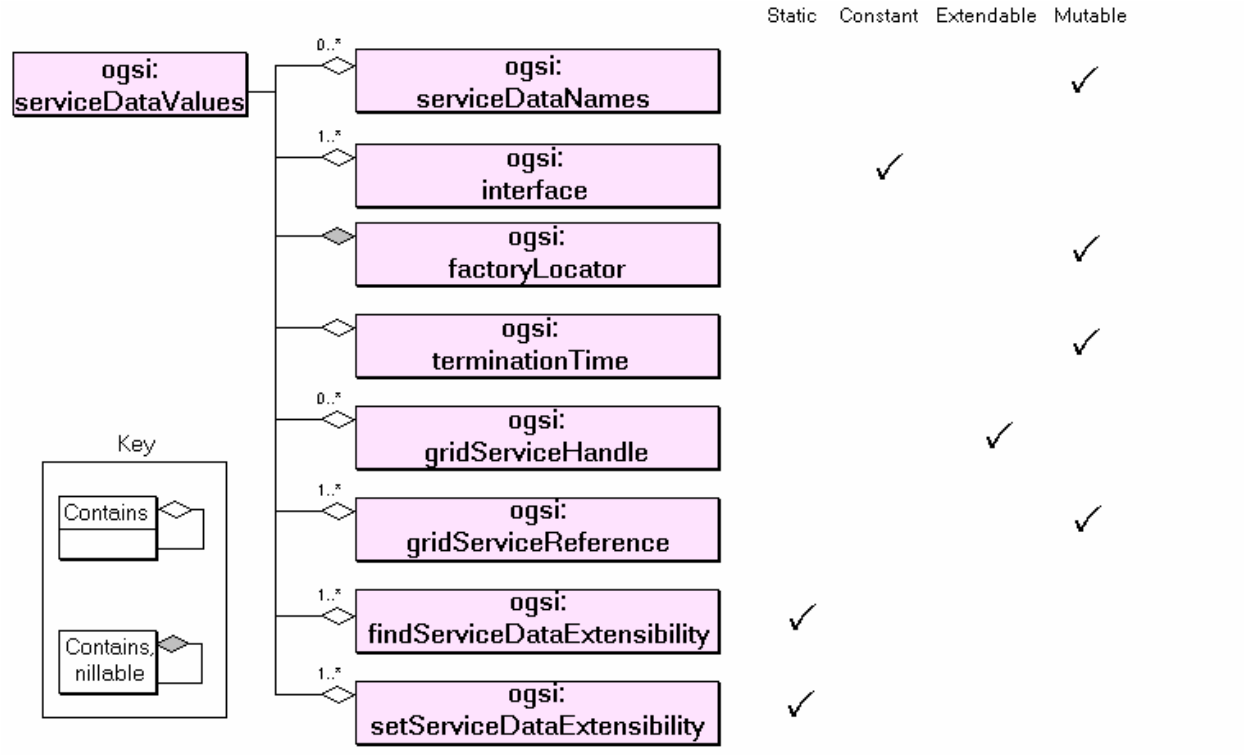

Figure 8-1: ServiceData Elements in the GridService portType

Figure 8-1 summarizes the Service Data Elements (SDEs) defined by the GridService portType. All Grid Services must extend this portType, so all Grid Services must report values for these SDEs. The meanings of the attributes of the SDEs ('Static', 'Constant' and so on) which control how and when the SDEs can be modified were described in Section 5.5.1

- **serviceDataName**
  The complete set of names of the SDEs supported by an Instance of a Grid Service. This must include the names in Figure 8-1 which are defined by the *GridService* portType (including its own name). However, Grid Services generally implement some portType which extends the GridService portType, so the serviceDataName SDE includes the names of SDEs defined by the GWSDL of the extended service in addition to the names shown in Figure 8-1. It also includes SDEs added dynamically by the Instance.

- **Interface**
  The names of all the portType(s) within the Service Instance's complete interface definition. For a basic GridService portType this would simply be the value "ogsi:GridService". However, Grid Services generally implement an extension of GridService. The Interface SDE values include the names of the extended portType and all the names of any portTypes directly or indirectly referenced by means of the 'extends' attribute, of course including "ogsi:GridService".
- **factoryLocator**
  A service locator to the factory that created this Grid Service Instance. If the instance was not created by a factory, this value must be xsi:nil. This locator must refer to the same factory service throughout the lifetime of this Grid Service Instance, although the handles and references contained in this locator may change during the Service Instance's lifetime.
- **gridServiceHandle**
  The Grid Service Handles for the Grid Service Instance.
- **gridServiceReference**
  Grid Service References to this Grid Service Instance. One of the value elements must be a WSDL representation. Other value elements may represent other forms of the GSR.
- **findServiceDataExtensibility**
  A set of operation extensibility declarations for the findServiceData operation. Any conforming inputElement declared by values of this SDE can be used by the client as a QueryExpression parameter to the instance's findServiceData operation, and implies the query input syntax and return values. The GridService portType declares an initial value for this SDE of "ogsi:queryByServiceDataNames" which provides a default, name-based syntax for the operation in all Grid Services.
- **setServiceDataExtensibility**
  A set of operation extensibility declarations for the setServiceData operation. Any conforming inputElement declared by values of this SDE can be used by the client as the UpdateExpression parameter to the instance's setServiceData operation, and implies the update syntax and return values. The GridService portType declares an initial values for this SDE of "ogsi:setByServiceDataNames" and "ogsi:deleteByServiceDataNames" which provides default, name-based syntax for the operation in all Grid Services.
- **terminationTime**
  The termination time for this Service Instance.

8.2. GridService Operations

The operations defined by the GridService portType are:

- findServiceData and setServiceData are described in Section 5.5.4.
- requestTerminationBefore, requestTerminationAfter and destroy which control the lifetime of an instance.

8.2.1. Destroy

The destroy operation is one way of terminating an instance and takes no parameters. It requests destruction of the Service Instance and, if the instance accepts the request, it sends an acknowledgement. However, the Service may not be immediately destroyed because there may be internal synchronizations and termination processes that need time to complete. The Instance will generally not respond to new requests once the destroy operation has been acknowledged.

The Destroy operation may return a *ServiceNotDestroyedFault* if the Instance did not intiate self-destruction.

8.2.2. RequestTerminationBefore

The requestTerminationBefore operation can be used to shorten the lifetime of a Service Instance requests that the termination time of the Service Instance be changed. The request specifies the latest desired termination time. Upon receipt of the request, the Service Instance may adjust its termination time, if necessary, based on its own polices and the requested time. Upon receipt of the response, the client should discard any responses that have arrived out of order, based on the timestamp in the response.

The operation can return *TerminationTimeUnchangedFault*: which means that the Service Instance ignored the request.

8.2.3. RequestTerminationAfter

The requestTerminationAfter operation can be used to extend the lifetime of a Service Instance. It requests that the termination time be changed by specifying the earliest desired termination time. Upon receipt of the request, the Service Instance may adjust its termination time, if necessary, based on its own polices and the requested time, and it returns the new termination time as a response to the operation. Upon receipt of the response, the client should discard any responses that have arrived out of order, based on the CurrentTimestamp in the response.

The operation can return *TerminationTimeUnchangedFault*: which means that the Service Instance ignored the request.

9. Handle Resolution

As discussed in Chapter 4, a Grid Service Instance is identified by one or more Grid Service Handles (GSHs). However, a GSH does not contain enough information for a consumer to communicate with the identified Grid Service Instance. The consumer needs to get hold of at least one Grid Service Reference (GSR) which contains the necessary information for communicating with the identified Grid Service Instance (Chapter 4).

Grid Service Instances that implement the HandleResolver portType, also referred to as *handle resolvers*, can be used to map a GSH to one or more GSRs. A GSH consists of a URI, and, like any URI, it consists of a 'scheme' name 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. The OGSI Specification is silent on the semantics or the mechanisms of the resolution because there are many ways to design Handles. For example, a secure resolution scheme such as SGNP [15] may be needed, or the Handles may be designed to accommodate information specific to a particular discipline, where a naming convention is already in existence.

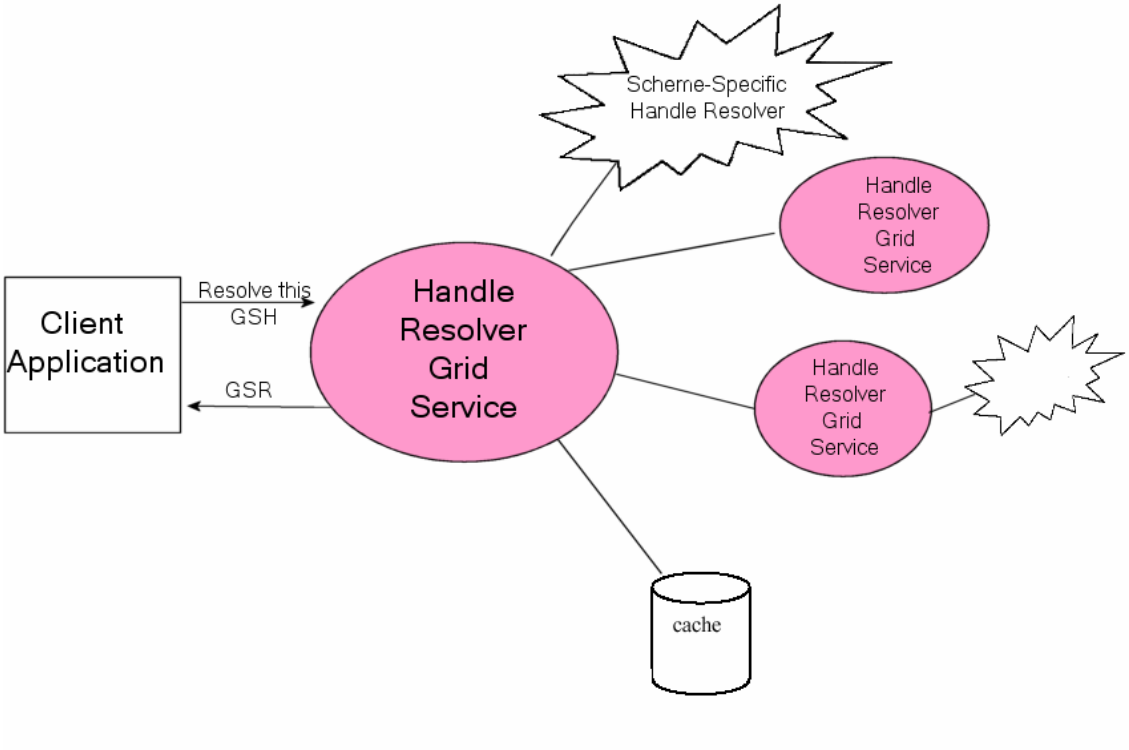

Figure 9-1: Resolving a GSH

A client need not be aware that GSHs belonging to multiple different schemes are being used because the handle resolver interface recognises any valid URI and will attempt to resolve it. Figure 9-1 shows a client using the handle resolver service which can resolve multiple kinds of GSH. This generic service delegates resolution to one of several, more specific services each of which may deal with a particular scheme, or with a particular part of a name space, and may in turn delegate to more specific services. The diagram also shows that a resolver

service may cache the results of a resolution, managing the cache based on the gooduntil times associated with a GSR.

The OGSI Specification only requires a handle resolver to adhere to the following:

- It must include a Service Data Element called *handleResolverScheme*; and
- It must support an operation called *findByHandle*.

9.1. The handleResolverScheme SDE

A handle resolver advertises the URI schemes that it understands through the *handleResolverScheme* SDE. For example, a factory for counter Grid Service Instances may use GSHs of the form "counter:global-unique-identifier," which are specific to counters, while another factory may use the generic http scheme, with GSHs such as "http://www.counter.com/grid/service/instance/10".

However, a particular handle resolver may only understand GSHs of a particular URI scheme. A consumer of a Grid Service Instance implementing the HandleResolver portType may get the names of the supported URI schemes through the *handleResolverScheme* SDE.

For example, a handle resolver for GSHs identifying counter Grid Service Instances will set the value of the *handleResolverScheme* SDE to "counter". A handle resolver that also supports GSHs using the "http" scheme will have two SDEs, with the values "counter" and "http":

```
<serviceDataValues>
   <handleResolverScheme>counter</handleResolverScheme>
   <handleResolverScheme>http</handleResolverScheme>
</serviceDataValues>
```

The service data declaration of the *handleResolverScheme* SDE specifies that its value can change throughout the lifetime of the handle resolver (mutability="true"), but only the handle resolver itself can change it (modifiable="false"). A value of xsi:nil is allowed, indicating that the handle resolver may be able to resolve all URI schemes.

9.2. The findByHandle operation

9.2.1. Input

The only operation that is declared as part of the HandleResolver portType is the findByHandle operation, which accepts as input a Locator called the 'HandleSet'. This contains one or more GSHs. As the definition of the Locator dictates, all the GSHs contained in the HandleSet must point to the same Grid Service Instance. The handle resolver is free to choose any of the GSHs in the HandleSet for the resolution process.

The GSRExclusionSet is an optional input to the findByHandle operation. It is a Locator that contains one or more sets of GSRs that the handle resolver should *not* return as part of the result of the resolution process. This enables the client to insist that the resolver return different (and possibly more suitable) GSRs than the ones already available. The handle resolver does not care *why* these GSRs had to be excluded – if the GSRExclusionSet Locator contains any GSHs, they will be ignored by the handle resolver.

9.2.2. Output

The output of the findByHandle operation is a Locator that contains one or more GSRs, the outcome of the resolution process. All the GSHs that were included in the input HandleSet will also be included in the output Locator. The handle resolver may also include additional GSHs to the same Grid Service Instance.

A client may pass a Locator which has been received from a handle resolver back to the same (or another) handle resolver at a later time for the purpose of refreshing the GSRs contained within it.

9.2.3. Faults

The following faults may be generated as a result of calling the *findByHandleOperation* operation:

- The *InvalidHandleFault* is returned when one of the GSHs in the HandleSet input parameter violates the syntax of its URI scheme.
- When the resolver cannot return a GSR that is not already contained in the *GSRExclusionSet* input parameter, the *NoAdditionalReferencesAvailableFault* is returned.
- If the *GSRExclusionSet* input parameter is empty but the resolver is still unable to return a GSR, the *NoReferencesAvailableFault* is returned. The fault may be extended with any of the following faults to provide more information about the error: *NoSuchServiceFault*, *NoSuchServiceStartedFault*, *ServiceHasTerminatedFault*, or *TemporarilyUnavailableFault*.
- It is possible that the handle resolver knows of another handle resolver that may be capable of performing the resolution. In this situation the *RedirectionFault* is returned, along with the GSH of the new handle resolver.

10. Finding Services: ServiceGroups and Registries

In Chapter 7, the rationale for Registries was given. The ServiceGroup, ServiceGroupEntry and ServiceGroupRegistration portTypes were introduced and the use of an example registry of factories, from the client perspective, was given. This chapter is mostly concerned with the problem of creating and managing registries of Grid Services. Some explanation of how registries are organised and the details of the ServiceGroup portTypes are given. The Open Grid Services Architecture (OGSA), which builds on OGSI to provide operational standards for Grid systems, will build on these base ServiceGroup portTypes provided by OGSI. OGSA registries are also likely to implement other portTypes such as the NotificationSource portType which allows clients to subscribe to registry state changes.

10.1. The Registry Interfaces

There are two components to a registry – Registration and Discovery. Registration places a service Locator in the registry and Discovery allows a client to retrieve it. Though it is logical to think of these as two parts of the same service, this is not necessarily the best way to organize the information. The Discovery service can use the registered Locators to find out more information about the services, for example dynamic information such as the current resource capacity and it may pass this on to a third party that provides selection among a collection of similar services. A Discovery service can also provide qualities of its own, such as scalability, which are independent of the qualities of the registration.

10.1.1. The purpose of registries in OGSA

A registry is a very general concept. In computing, we may define it as a catalogue of objects, described and indexed by the objects' metadata. The objects may be either inside the computing system (i.e. the objects consist entirely in data); or they may be real-world items, with only metadata inside the computing system; or they may be computing-system proxies for real-world objects. In all cases, the essence of the registry pattern is that a registry stores some part of an object's metadata without storing either the object itself or (for a real-world object) its proxy.

Given a registry, clients may inspect the metadata and use them to reason about the catalogued objects. Typically, clients then use the metadata to locate and access some of the catalogued objects. Often, the only use of the registry is to locate the objects.

OGSI is concerned with registries of Grid Services. Grid Services can be used to build other kinds of registries, but OGSI has specific portTypes that can be used to build registries where the entries describe Grid Services.

The essential requirement of a service registry is that the metadata include a Locator for each catalogued service. The registry itself need not resolve the Locators to include valid GSRs, although the service running the registry may also include a HandleResolver portType for this purpose. From OGSA's point of view, "locating" a Service Instance through a registry means finding its persistent Handle.

10.1.2. OGSI Support for Service Registries

In OGSA, service registries have two uses:

- finding service factories from which to make new instances;
- finding existing instances.

Both uses are covered by the following three OGSI portTypes:

- serviceGroup
- serviceGroupEntry
- serviceGroupRegistration

A ServiceGroup portType maintains information about a group of other Grid Services. These services may be members of a group for a specific reason, such as being part of a federated service, or they may have no specific relationship, such as the services contained in an index or registry operated for discovery purposes. A container registry that holds a fixed list of member services intended for discovery purposes only, could be defined by a portType that extends the base behaviour described by ServiceGroup. The ServiceGroup portType extends the GridService portType.

The serviceGroupEntry portType defines the interface through which the lifetime of individual entries in a service group may be managed. Each ServiceGroupEntry service refers to (is a proxy of) a Grid Service Instance that is a member of the ServiceGroup. Membership registration through a ServiceGroupEntry instance is a soft state operation that must be periodically refreshed, thus allowing discovery services to deal naturally with dynamic service availability. The ServiceGroupEntry portType extends the GridService portType.

The ServiceGroupRegistration portType provides a management interface (add and remove operations) for a ServiceGroup. The add operation creates a ServiceGroupEntry and adds it to the ServiceGroup. The remove operation removes each ServiceGroupEntry that matches the MatchExpression input parameter of the remove operation. The ServiceGroupRegistration portType extends the ServiceGroup portType.

10.1.3. Factory and Instance Registration

An OGSI factory is normally created when the hosting environment is first started and lives for as long as the container lives. Registration of factories and other persistent services is best carried out during instantiation of the Registry service.

Following a createService operation from a client, a factory creates a new Grid Service Instance and returns a Locator to the client. OGSI does not prescribe the way in which the new Instance is registered, or even whether it should be. Either the factory or the newly created Instance could use the ServiceGroupRegistration add operation to register the new Instance in a registry.

10.1.4. Making Discoveries

The operations inherited from GridService portType may be used to query service data in a registry as shown in the example in Chapter 7. The example illustrates the use of a simple queryByService on the content of a registry. More powerful discoveries could be built using XPath or XQuery. The Registry may implement other specialised operations for the purpose of querying its content.

10.2. Grouping services: The ServiceGroup portType

Below is a diagram of the WSDL markup elements that describe the ServiceGroup portType.

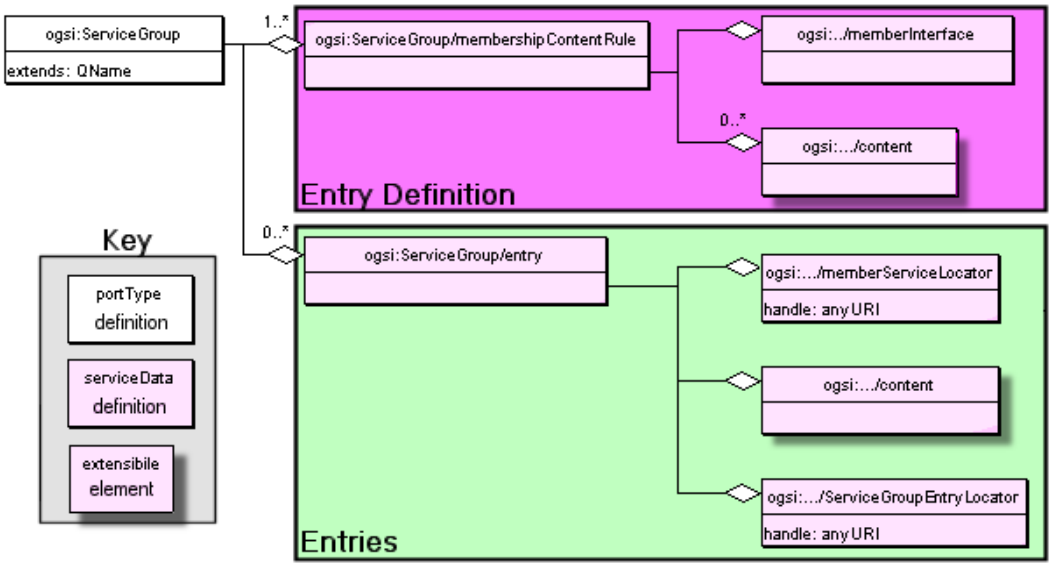

Figure 10-1: GWSDL Description of Service Group.

ServiceGroup allows a collection of Grid Services to be maintained and managed. This portType has no operation and consists of two SDE elements: membershipContentRule and entry as well as the SDE's and operations inherited from GridService.

10.2.1. The membershipContentRule SDE:

This SDE can be used to restrict the membership Grid Service Instances in a ServiceGroup, by specifying the "rules" that must be satisfied before membership is allowed. The rules, as shown in Figure 10-1, specify the following:

- A list of interfaces the member Grid Service Instances may implement. Each Service Instance that is a member of a ServiceGroup must implement one or more interfaces from this list.
- For each interface, zero or more contents which describe the member. Typically these are names of its SDEs, or a set of XSD elements identifiable through QNames, that are needed for a member to become a part of this group. (The contents of QNames are described in the ServiceGroupEntry).

This service data has a mutability value of "constant" and hence these rules are created by the runtime on ServiceGroup service startup.

The example in Chapter 7 could enforce its restriction to only register factories and also include information about the portTypes supported by the created instances as follows:

```
<membershipContentRule>
<memberInterface>ogsi:Factory</memberInterface>
<content>createServiceExtensiblityInfo</content>
</ membershipContentRule >
```

10.2.2. The entry SDE:

The entry SDE is a compound element made of three components as shown in Figure 10-1:

- a Locator element pointing to the registered service itself;
- an extensible content element that contains metadata describing the registered service. The content element is an XML element advertising some information about the member service.
- a Locator element pointing to the registration proxy for the registered service.

Given the membershipContentRule above, an example entry for the "shared" counter factory from Chapter 7 could have the following value:

```
<entry>
   <serviceGroupEntryLocator nil="true"/>
   <memberServiceLocator>
      <ogsi:handle>
        http://exemplar.org/ogsa/services/SharedCounterFactory
      </ogsi:handle>
   </memberServiceLocator>
   <content>
      <createServiceExtensibilityInfo>
         <createsInterface>Counter</createsInterface>
         <createsInterface>GridService</createsInterface>
         <createsInterface>NotificationSource</createsInterface>
      </createServiceExtensibilityInfo>
   </content>
</entry>
```

10.3. Proxies for ServiceGroup members: The ServiceGroupEntry portType

The registry facility also has one ServiceGroupEntry Instance for each registered member. The intention is to have the appearance of a registration proxy service in order to use the lifetime management facilities of the GridService portType and limit the lifetime of registration. In practice, a registry may implement these proxy services in some light-weight form in order to save resources. That is, a registry for a trivial number of services may use conventional Service Instances for its proxy services, using ordinary service containers, while a registry intended for thousands of registrants may be built as a specialized service container with a more scalable implementation of the proxies. The ServiceGroupRegistration portType provides an operation which creates the ServiceGroupEntry Instances. The ServiceGroupEntry portType extends the GridService portType to which it adds two new SDE: memberServiceLocator and content.

10.3.1. The memberServiceLocator SDE:

This SDE contains a service Locator to the Grid Service Instance whose membership of the ServiceGroup is managed by this ServiceGroupEntry.

10.3.2. The content SDE:

This SDE contains information about the member Grid Service Instance. Since membership of a ServiceGroup is restricted by membershipContentRule SDE , this content SDE must satisfy any restrictions specified in the membershipContentRule SDE. If the member Grid

Service Instance implements several memberInterfaces included in the membershipContentRule SDE values, all content elements associated with any of these memberInterfaces must be included in this content SDE.

10.4. Managing ServiceGroup membership: The ServiceGroupRegistration portType

In a registry implementing the ServiceGroupRegistration portType, clients can call the add operation of the ServiceGroupRegistration portType to register a service and the remove operation to de-register services. Each call to add registers one service. Each call to remove deregisters a set of related services. A registration proxy can be set to terminate at a certain time, in the normal manner of OGSI services, thus limiting the lifetime of a registration.

The ServiceGroupRegistration portType extends the ServiceGroup portType by adding two new SDE: addExtensibility and removeExtensibility and two new operations: add and remove. The separation of the Registration operations from the Discovery operations of the ServiceGroup can be used to provide different access controls on these two aspects of registries.

10.4.1. The addExtensibility SDE:

This contains a set of operation extensibility declarations for the add operation. Any inputElement declared by the value of his SDE may be used as the Content parameter to the instance's add operation, and implies the add semantics and return values that may result from the add operation. These addExtensibility elements may be different from the elements listed in the membershipContentRule SDE because the content may be derived from them or referenced by them.

10.4.2. The add Operation:

The add operation creates a ServiceGroupEntry and adds a corresponding entry in the Service Group. This operation takes two mandatory parameters (Service Locator and Content) and an optional parameter (TerminationTime), outputs two parameters (ServiceLocator and CurrentTerminationTime) or may generate a number of Faults.

The ServiceLocator input parameter is a Locator for the member Grid Service Instance. The Content input parameter must conform to an input element declaration denoted by one of the addExtensibility SDE values. The add operation converts the value in the Content argument into an element conformant with one or more of the entries contained in the membershipContentRule. This transformation is based on the type of Grid Service as inferred by the ServiceLocator and processed according to ServiceGroup specific semantics. The resulting transformed Content becomes the ServiceGroupEntry Content SDE. Finally the TerminationTime input parameter is used to specify the earliest and latest initial termination times of the created ServiceGroupEntry Service Instance.

The ServiceLocator output parameter refers to the newly created ServiceGroupEntry. The CurrentTerminationTime time refers the ServiceGroupEntry's currently planned termination time.

The set of Faults that may be returned from the add operation are: ExtensibilityNotSupportedFault, ExtensibilityTypeFault, ContentCreationFailedFault, UnsupportedMemberInterfaceFault, AddRefusedFault and the generic OGSI Fault that encompasses all other Faults.

10.4.3. The removeExtensibility SDE:

This contains a set of operation extensibility declarations for the remove operation. Any inputElement declared by value of his SDE may be used as the MatchExpression parameter to the instance's remove operation, and implies the remove semantics and return values that may result from the remove operation. OGSI defines one standard removeExtensibility SDE value: matchByLocatorEquivalence.

10.4.4. The remove Operation:

The remove operation removes all entries from the ServiceGroup that matches the operation's input argument. This operation takes one input parameter (MatchExpression), has no output parameters except acknowledgement of the operation completion and may generate a number of Faults.

The MatchExpression input parameter must conform to an inputElement declaration denoted by one of the removeExtensibility SDE values. The MatchExpression is evaluated against all entries in a ServiceGroup. Each entry that matches is removed. For example the matchByLocatorEquivalence expression compares one or more Locators for textual equality with the canonical XML form of the memberServiceLocator of each entry contained in the ServiceGroup and removes the entries for which there is a match. This operation has no effect on the member Grid Services referred to in the removed entries.

The set of Faults that may be returned from the add operation are: ExtensibilityNotSupportedFault, ExtensibilityTypeFault, MatchFailedFault, RemoveFailedFault and the generic OGSI Fault that encompasses all other Faults.

11. Creating Transient Services: The Factory

From a programming model perspective, a factory is an abstract concept or pattern. A consumer may use a factory to create an instance of a Grid Service and receive a Locator for it. Any service may be a factory of other Grid Service Instances without having to adhere to a specific interface. Nevertheless, the OGSI Specification defines a standard portType for a factory Grid Service Instance for those service implementers wishing to provide a general, rather than service-specific, interface for creating new instances.

The OGSI Specification suggests that any newly created Grid Service Instance should be registered with a handle resolver in order to receive a GSH. The mechanism by which this may take place is implementation-specific.

As with all Grid Services, the factory must extend the GridService portType; it also introduces one SDE and one operation.

11.1. The createServiceExtensibility SDE

As explained in Section 5.6, for each extensible operation in a portType there should be a Service Data Declaration of type OperationExtensibilityType. The createServiceExtensibility SDE is such a declaration. Its type is CreateServiceExtensibilityType, which extends OperationExtensibilityType to include the set of portTypes that the Grid Service Instance to be created should support. Implementers of the Factory portType may decide to provide additional extensibility elements that should be declared through this SDE.

11.2. The createService operation

11.2.1. Input

The *createService* operation of the Factory portType accepts two arguments as input:

- TerminationTime. This is optional and allows the client calling the createService operation to specify the earliest and latest initial termination times of the Grid Service Instance to be created. The factory will select an initial termination time from the window specified by the client and will return it.
- CreationParameters. This, too, is an optional argument and must conform to one of the extensibility parameter declarations defined as values of the createServiceExtensibility SDE. It is used when the requester wishes to send additional information to the Service Instance implementing the factory portType. For example, a factory may advertise through the createServiceExtensibility SDE more than one supported combinations of interfaces that created Service Instances can support. A requester can use the CreationParameters to select which of the combinations the new Service Instance must support.

11.2.2. Output

As a result of requesting the creation of a new Service Instance, the requester receives:

- A Locator to the newly created Service Instance.
- The CurrentTerminationTime, which is an element identifying the planned termination time for the newly-created Service Instance. Of course, consumers may request for this value to change through the operations provided by the GridService portType

The Service Instance implementing the *Factory* portType may also decide to return an *ExtensibilityOutput* element which is specific to the Service Instance created.

11.2.3. Faults

The faults that may be communicated back to the consumer are:

- *ExtensibilityNotSupportFault*. The type of the *CreationParameters* was not supported by the Service Instance that received the request.
- *ExtensibilityTypeFault*. The value of the *CreationParameters* input element could not be validated against its declared type.
- *ServiceAlreadyExistsFault*. This fault can occur if the creation parameters for the instance include its intended identity. If the Service Instance that was supposed to be created already existed, this fault is returned, and also carries the Locator to the existing Service Instance .
- *Fault*. Any other fault that may occur.

12. GridService Notification

12.1. Notification Source and Notification Sink

A Grid Service which wants to notify its clients of changes to its state should implement NotificationSource portType. Such a Grid Service (also called a *source*) can send notification messages to any number of clients. A client (also called a *sink*) that wishes to receive notification messages should implement NotificationSink portType.

A notification message is an XML element that is sent from the source to the sink. The type of the notification message is determined by the subscription expression. A subscription expression also describes what messages should be sent and when they should be sent based on changes to values within the serviceDataSet of the Grid Service Instance implementing NotificationSource interface.

12.2. Subscription Model

To enable notifications, a Grid Service must implement the Notification Source portType. The following example adds notification support to the counter example explained in Section 5.7

```
<definitions>
……
<gwsdl:portType
       name="NotificationcounterPortType"
       extends="counter:CounterPortType ogsi:NotificationSource"/>
……
</definitions>
```

The NotificationSource portType extends the GridService portType and has two SDEs and one operation.

12.3. NotificationSource : Service Data Declarations

NotificationSource defines two SDEs to control the scope and nature of subscriptions.

12.3.1. notifiableServiceDataName

The notifiableServiceDataName SDE contains the QNames of the service data elements to which a client can subscribe for notification of changes.

12.3.2. subscribeExtensibility SDE

The subscribeExtensibility SDE is used for expressing the nature of subscription. Every Grid Service Instance that implements the NotificationSource portType must support a subscribeExtensibility SDE value of subscribeByServiceDataNames. subscribeByServiceDataNames allows clients to receive notifications whenever the corresponding value(s) of SDE is changed.

However, a Grid Service can choose to support other subscription expressions. For example, subscribeBySDEAndXPathRestriction as defined below can be supported by a Grid Service to allow the clients to specify an XPath expression based on which notifications can be sent when the values of the SDEs change.

```
<xsd:element name="subscribeBySDEAndXPathRestriction">
<xsd:complexType>
<sequence>
<xsd:element name="subscribeBySDEAndXPath"
type="tns:SDEXPathSubscriptionType"/>
</sequence>
</xsd:complexType>
</xsd:element>
<xsd:complexType name="SDEXPathSubscriptionType">
<sequence>
<xsd:element name="name" type="QName"/>
<xsd:element name="xpath" type="string"/>
</sequence>
</xsd:complexType>
```

12.4. NotificationSource :: subscribe operation

Notification Clients can use this operation to subscribe to changes to the target instance's service data. The life-time of the subscription is managed by a separate Grid Service Instance called *subscription instance*.

This operation takes as input

- A subscription expression
- The Locator of the notification sink to which messages will be delivered
- Initial expiration time of the subscription as requested by the client

This operation outputs

- The Locator to the subscription instance
- Initial termination time of the subscription as decided by the source

Notification clients can discover the subscription expression types supported by a Service Instance by performing a findServiceData request on the instance, using a queryByServiceDataNames element, which contains the name “ogsi:subscribeExtensibility”. Clients can then choose a subscription expression in the subscribe operation to control the scope and nature of the subscription. The *subscribeByServiceDataNames* subscription expression, which is supported by all Notification Source Grid Service Instances, can be used to subscribe for notification messages whenever there are any changes in service data elements.

The subscribeByServiceDataNames allows the client to set the minimum and maximum intervals between notification messages.

A client invokes a subscribe operation on the Grid Service's NotificationSource port. As an example, a client can include this as the input parameter of the subscribe operation:

```
<ogsi:subscribebyServiceDataNames>
 <name minInterval=PT15M   maxInterval=PT30M>counterValue</name>
```

```
</ogsi:subscribebyServiceDataNames>
<sink>
 <ogsi:handle>
  http://localhost/ogsa/services/CounterNotificationClient
 </ogsihandle>
</sink>
<ogsi:expirationTime>2003-12-12-26T10:20:00.000-06:00</ogsi:expirationTime>
```

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 a minInterval. 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 service data elements' values, then the source will resend the same values. When the value is "infinity" the source need never resend service data values if they do not change. If this property is not specified, then the notification source may choose a maxInterval.

The sink element contains the Locator of the notification sink to which messages will be delivered. This Locator may contain only references. Specifying references will allow plain Web Service that implements the NotificationSink portType to subscribe for notifications. Please note that Web Services does not have the concept of handles.

The subscribe operation results in the creation of a subscription instance of type NotificationSubscription. The expirationTime element allows the client to specify an initial termination time of the newly created NotificationSubscription instance.

A response from the NotificationSource Grid Service (for the invoking the subscribe operation as given above) might look like

```
<ogsi:subscriptionInstanceLocator>
<ogsi:handle>http://localhost/ogsa/services/CounterNotificationSubscription
</ogsi:handle>
</ogsi:subscriptionInstanceLocator>
<ogsi:currentTerminationTime>2003-12-12-26T10:20:00.000-06:00
</ogsi:currentTerminationTime>
```

12.5. Lifetime Management of the NotificationSubscription

The subscribe operation discussed above results in the creation of a subscription instance of type NotificationSubscription. Notification Clients are responsible for managing the life-time of subscription using the soft-state life time management as explained in Section 2.2. Additionally, clients can discover properties of the subscription such as subscriptionExpression and sinkLocator by querying the respective SDEs of the NotificationSubscription instance.

NotificationSubscription portType extends Grid Service portType and contains no operations.

12.6. Notification Clients

A Notification Client can be a Grid Service or a Web Service and should implement the NotificationSink portType. The NotificationSink portType defines only one operation called deliverNotification which is used by the source to deliver the XML messages.

12.7. NotificationSource :: deliverNotification operation

This operation has one input, and it does not return an output or faults.

Input:

- Message: An XML element containing the notification message that flows from the source to the sink. The XML type and the frequency of the message are dependent upon the subscription expression.

For example, for the subscription request described above, a notification message might look like this:

```
<ns1:serviceDataValues ...">
     <ns3:CounterValue ...
         xsi:type="ns2:CounterValueType">
         <value xmlns="">111</value>
     </ns3:CounterValue>
</ns1:serviceDataValues>
```

A Web Service implementing the NotificationSink portType does not need to implement the GridService portType. This enables Web Services living outside an OGSI hosting environment to subscribe to notifications.

12.8. Reliable Notifications

Different protocol bindings can be used with the notification interfaces described above. For example, to achieve reliable notifications we can use protocol bindings which use reliable message delivery mechanisms. Some of the commercially available message queue implementations support guaranteed message delivery. These commercial messaging systems can also act as notification intermediaries by transferring or filtering the notification messages before delivering it to the sink.

12.9. Notification Usage Scenarios

Notifications can be used in a variety of scenarios. Event oriented systems which need asynchronous delivery of events can benefit greatly from the OGSI Notification framework. For example, a Grid Service which is published in a Grid Service registry can use notifications to update the registry about its status. The status of the Grid Service (for example) can change from “Service_Active”, “Service_Busy”, “Service_Error”, “Service_ShuttingDown”, “Service_Offline”, “Service_Maintenance” and by using notifications the registry can be kept up-to-date.

13. Grid Services Security

This chapter describes the requirements for security functions in the context of the services enabled by the OGSI Specification. The Specification says very little about security: it simply states that security is not its concern. This chapter of the Primer explains this statement and summarises the information contained in related documents which are concerned with security for Grid Services.

13.1. Architecture Layering

The OGSI Specification is built on assumptions made about services provided by underlying facilities (called the hosting environment) and communications protocol layers which are separate from the application in the requester and provider. The communications layers have been described in this Primer in terms of bindings, which can be described in WSDL (see Figure 4-1) and which relate abstract interface descriptions to the transport protocols. Operation of security functions can be achieved in these underlying layers based on secure communications and services which construct message components independently from the applications. The Anatomy of the Grid [2] and the OGSA Security Roadmap [24] describe the relationship between the protocol layers, the services and infrastructure which enable security functions and the applications which use them.

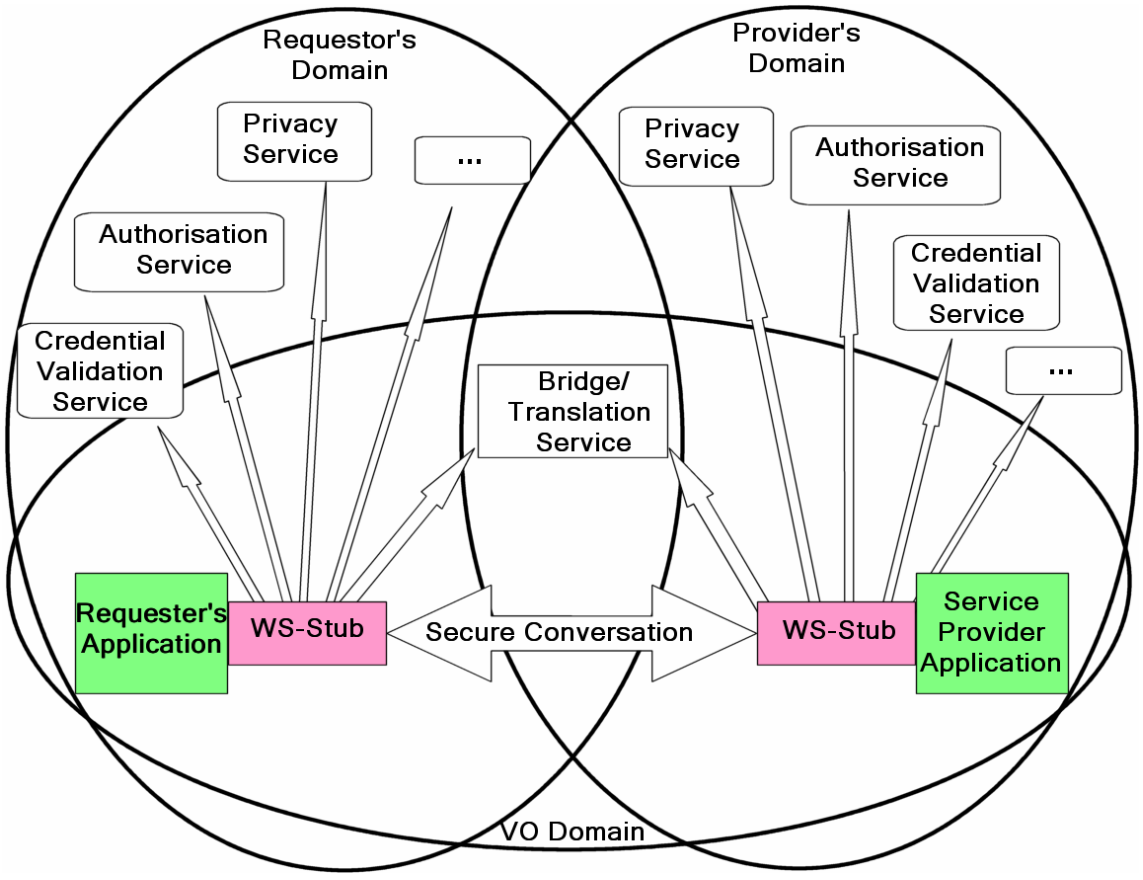

Figure 13-1: Relationship of security functions to other components.

Figure 13-1, based on the OGSA Security Roadmap, shows the relationship of the security functions to the components of Grid Service implementations introduced in Figure 6-2 and Figure 7-1. In the requester's system, the security functions are not called directly by the requesting application, but are managed by a stub. The function of the stub is:

- To analyse the WSDL description of the service and orchestrate the processing of security functions required by it. For example, there may be functions which apply to all messages, or to parts of some messages only. These characteristics are decided by the policy information which is contained in or referenced by the WSDL, but which, like binding information, is also separate from the application-level message formats.
- To gather security information such as the userid from the requester's environment.
- To engage any necessary communications with third party services. These services might be the validation of the providers' credentials, checking of access permissions, logging, or translation of information, such as userids or authentication tokens which may be needed in the provider's domain.
- To create any necessary encoding of security information in the message.

The figure also shows an important concept in the description of Grid Services called the *Virtual Organisation*. Traditionally, organisations are established around stable administrative structures which are used to establish trust and give permission for a requester to use a Service. In the Grid world, the relationship must be more flexible and dynamic and the term *Virtual Organisation* describes this more dynamic facility.

Like OGSI, the operation of security in OGSA is constructed using Web Services standards. The Security Roadmap summarises these standards and their exploitation in a Grid environment.

14. Glossary of Terminology

These terms and abbreviations are used in the text and are defined where they are first used. The list below may help you if you dip into the text without reading from the beginning.

Web Service: A software component identified by a URI [RFC 2396], whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web Service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols. (This is the definition from the Web Services Architecture [19].)

Web Service Consumer: A software component that sends messages to a Web Service.

Stateful Web Service: A Web Service that maintains some state between different operation invocations issued by the same or different Web Service Consumers.

Grid Service: A general term used to refer to all aspects of OGSI. The term "Grid Service" is sometimes used to refer to a Grid Service Description document and/or a Grid Service Instance for a particular service.

Grid Service Description: A WSDL-like document that defines the interface of Grid Service Instances. The defined interface must extend the OGSI GridService portType.

Grid Service Instance: A stateful Web Service whose interface adheres to that defined by a Grid Service Description and whose lifetime management properties are well defined.

Service Data Element: An attribute-like construct exposing state information through operations defined by the GridService portType.

Grid Service Handle: A URI that permanently identifies a Grid Service Instance.

Grid Service Reference: A binding-specific endpoint with a potentially limited lifetime that provides access to a Grid Service Instance.

15. Author Information and Acknowledgements

Tim Banks (Editor)
IBM
Hursley Park, Winchester, UK. SO21 2JN.
Email: tim_banks@uk.ibm.com

Dr. Abdeslem Djaoui,
CLRC
Rutherford Appleton Laboratory, UK, OX11 0QX.
Email: a.djaoui@rl.ac.uk

Dr. Savas Parastatidis,
North-East Regional e-Science Centre
School of Computing Science, University of Newcastle upon Tyne, UK
Email: http://savas.parastatidis.name

Anbazhagan Mani
IBM India Software Lab
Golden Enclave, Airport Road, Bangalore 560 017, INDIA
Email: manbazha@in.ibm.com

Also we acknowledge the authors of the OGSI Specification that provided source material and review comments: Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, Carl Kesselman, Tom Maguire, Thomas Sandholm, Dr. David Snelling and Peter Vanderbilt.

15.1. Acknowledgements

We are grateful to numerous colleagues for contributions and discussions on the topics covered in this document, in particular (in alphabetical order, with apologies to anybody we've missed):

Marc Brooks, John Carter, Chris Dabrowski, Kate Keahey, Guy Rixon, Krishna Sankar, and Jem Treadwell.

16. Document References

[1] *Open Grid Services Infrastructure (OGSI) Version 1.0.* June 27, 2003. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, D. Snelling, P. Vanderbilt, Graham, C. Kesselman, D. Snelling, P. Vanderbilt. Available at: http://forge.gridforum.org

[2] *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, I. Foster, C. Kesselman, S. Tuecke, Authors. International Journal of High Performance Computing Applications, 15 (3). 200-222. 2001. Available at http://www.globus.org/research/papers/anatomy.pdf.

[3] Globus Toolkit 3. Available from http://www-unix.globus.org/toolkit/

[4] *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, I. Foster, C. Kesselman, J. Nick, S. Tuecke, Authors. Globus Project, 2002. Available at http://www.globus.org/research/papers/ogsa.pdf

[5] *What is Service-Oriented Architecture?* Hao He. Available at http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[6] *A Model, Analysis, and Protocol Framework for Soft State-based Communication*. Suchitra Raman, Steven McCanne.

[7] *RFC 1831. RPC: Remote Procedure Call Protocol Specification Version 2*. Available at http://www.ietf.org/rfc/rfc1831.txt

[8] *The Object Management Group CORBA specification.* Available from the OGM Web Site: http://www.omg.org/corba/

[9] *Java RMI specification.* Available at: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html

[10] *DCOM Architecture*. Markus Horstmann and Mary Kirtland. Available from http://msdn.microsoft.com

[11] *Hypertext Transfer Protocol. RFC 2616*. Available at http://www.w3.org/Protocols/rfc2616/rfc2616.html

[12] *Web Services Description Language (WSDL) 1.1 W3C Note 15 March 2001* Available at http://www.w3.org/TR/wsdl

[13] *Web Services Description Language (WSDL) Version 1.2, W3C Working Draft 3. March2003, World Wide Web Consortium.* Available at http://www.w3.org/TR/2003/WD-wsdl12-20030303

[14] *J2EE Platform Specification.* Available at http://java.sun.com/j2ee/1.4/download.html

[15] *Secure Grid Naming Protocol (SGNP): Draft Specification for Review and Comment. GGF4 Submission May 3, 2002* Available at http://sourceforge.net/projects/sgnp/

[16] *Java (TM) API for XML-Based RPC (JAX-RPC).* Available at *http://java.sun.com/xml/jaxrpc/docs.html*

[17] *Welcome to WSIF: Web Services Invocation Framework* Available at *http://www.apache.org/wsif*

[18] *Enterprise JavaBeans TM Specification, Version 2.1.* Sun Microsystems. Available at http://java.sun.com/products/ejb/docs.html

[19] *The Web Services Architecture*. Available at http://www.w3.org/TR/ws-arch/wsa.pdf

[20] *Universal Description, Discovery, and Integration (UDDI) 3.0 specification*. Available at http://uddi.org/pubs/uddi-v3.00-published-20020719.htm

[21] The Web Services Interoperability Organisation. http://www.ws-i.org/

[22] The Grid Security Infrastructure. Described at http://www.globus.org/security/

[23] *GWSDL to WSDL translation*, Available at https://forge.gridforum.org/projects/ogsi-wg/document/GWSDL_to_WSDL_1.1_Transformation/en/2

[24] *The OGSA Security Roadmap*. Available at https://forge.gridforum.org/projects/ogsa-sec-wg/document/OGSA_Security_Roadmap/en/1

17. Copyright Notice

17.1. Intellectual Property Statement

The GGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the GGF Secretariat.

The GGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the GGF Executive Director (see contact information at GGF website).

GLOBAL GRID FORUM

office@gridforum.org

www.ggf.org

18. The Index