OGSI.NET Programmer’s Reference

Glenn Wasson

Updated: April 5, 2004

This document describes how to author and deploy a grid service in OGSI.NET. This is not a primer on grid services or OGSI and the reader is assumed to be generally familiar with terms and concepts from both OGSA [1] and OGSI [2]. The goal of the OGSI.NET project is to make programming a grid service as easy as programming a web service! The following sections describe how to take some service logic and turn it into an OGSI compliant grid service hosted by the OGSI.NET container.

1 Write Service Logic

The first step in writing a grid service for OGSI.NET is to write the logic for your grid service. At this point, concentrate on the functions and data that you want to expose to “the grid”. As an example, consider someone writing a weather forecasting service. Let’s say that this service would allow clients to determine the current forecast, for their local area, for a given date. The service would also provide access to data about the current weather conditions (we’ll choose temperature and wind speed). We’ll start off with the following C# code.

```
public class WeatherForecastService
{
   public double temperature;
   public int windSpeed;
   public int areaCode;

   public WeatherForecastService(int clientAreaCode)
   {
        areaCode = clientAreaCode;
        // configure the service for this location
   }

   public string GetForecast(DateTime forecastDay)
   {
        string forecastDescription;
        // compute forecast
        return forecastDescription;
   }
}
```

Figure 1. Initial Service Logic for the WeatherForecastingService

For simplicity, this code does not use any functionality specific to grid computing as per OGSA (e.g. notification). However, many grid services will want to take advantage of this functionality and OGSI.NET’s class libraries provide many useful functions for grid service authors. These are discussed in Appendix A.

2 Annotate the Service Logic

In general, what separates a grid service from service logic is metadata about how that logic (and accompanying data) should be exposed to “the grid”. This includes what methods/data should be exposed as well as the terms of the exposure (i.e. policy). In OGSI.NET, this metadata is defined by a set of C# Attributes that can be placed on classes, functions or data members. These attributes are used by both the OGSI.NET container at runtime and by static tooling (i.e. our WsdlGenerator) to determine how the container makes the grid service available to clients.

The attributes used by OGSI.NET can be roughly divided into three categories, those that deal with methods, those that deal with data and those that deal with policy. We will cover each of these areas in detail.

2.1 Attributes for Methods

There are two ways to expose methods in an OGSI.NET grid service, by writing a port type and then having your service include that port type, or by allowing OGSI.NET to build a port type for your service directly from the service logic. There are advantages to each and both involve some common attributes. We'll start with defining your own port type. Returning to the WeatherForecastService example, let's create a port type (`WeatherForecastPortType`) and have it contain the single method of the service (`GetForcast`). All port types must derive off of the `GridServiceSkeleton` class provided with OGSI.NET. Each method in the port type must be decorated with one or more attributes.

The easiest way expose methods is by using the `[WsdlBaseName]` attribute in conjunction with the `[WebMethod]` attribute. `[WsdlBaseName]` provides a "base name" which (when appended with various endings) is used to provide names for various elements throughout the WSDL and binding generation process. The `[WsdlBaseName]` attribute has Name and Namespace parameters whose values will be used for various element names and targetNamespaces in the WSDL and Bindings generated by OGSI.NET. When using the `[WsdlBaseName]`, any method marked with the `[WebMethod]` attribute will be exposed.

Figure 2 shows example usage of these attributes for the WeatherForecastPortType (ignore the `[WebService]` and `[WebServiceBinding]` attributes for the moment).

```
[WsdlBaseName(Name="WeatherForecast",
              Namespace="http://gcg.cs.virginia.edu/weatherforecast")]
[WebService]
[WebServiceBinding]
public class WeatherForecastPortType : GridServiceSkeleton
{
   public double temperature;
   public int windSpeed;
   public int areaCode;

   public WeatherForecastPortType()
   {
   }

   [WebMethod]
   public string GetForecast(DateTime forecastDay)
   {
        string forecastDescription;
        // compute forecast
        return forecastDescription;
   }
}
```

Figure 2. Annotating Methods using [WsdlBaseName]

This port type will have a single method, GetForecast.

While [WsdlBaseName] and [WebMethod] are the easiest way to expose methods in a grid service, there are times when the grid service author is writing a service to conform to pre-existing WSDL or WSDL bindings. For these situations, OSGI.NET provides some additional attributes that override the default names that OGSI.NET's WsdlGenerator associates with the `[WsdlBaseName]` attribute.

The first is the `[WebService]` attribute which tells the WsdlGenerator to create WSDL definitions for this port type. When used with no parameters (as in Figure 2) it is just a tag to tell the WsdlGenerator to create a WSDL definition for this port type. In this case, the name attribute of the <definitions> element of the WSDL will be "*basename*PortType" (i.e. whatever was specified for the Name parameter of [WsdlBaseName] with "PortType" appended to it). The targetNamespace attribute will be set to the Namespace parameter of [WsdlBaseName]. However, if the [WebService] attribute is used with Name and Namespace parameters, those values will be used for the <definitions> element of the port type's WSDL instead.

Similarly, the [WebServiceBinding] attribute tells the WsdlGenerator to create (in this case SOAP) bindings for the port type. When used with no parameters, the name attribute of the <definitions> element of the Binding file will be "*basename*Binding" (in this case WeatherForecastBinding) and targetNamespace attribute will be equal to the Namespace parameter used in [WsdlBaseName] plus "/bindings" (in this case http://gcg.cs.virginia.edu/weatherforecast/bindings). However, if the [WebServiceBinding] attribute is used with Name and Namespace parameters, those values will be substituted for the [WsdlBaseName] defaults.

Finally, it may be necessary to specify binding information for individual methods. In this case, the [SoapDocumentMethod] attribute can be used (along with [WebMethod]. The attribute takes two parameters, the SOAPAction name and the Binding name. The SOAPAction name sets the value that must be present in the SOAPAction HTTP header of a SOAP request to access the method. The Binding name is a means of grouping all "binding" information for a particular messaging type. If the [WebServiceBinding] attribute is used, the Binding parameter for each [SoapDocumentMethod] attribute must match the Name parameter for the [WebServiceBinding] attribute. Figure 3 shows an example of using [SoapDocumentMethod].

```
[WebService("WeatherForecastService", "http://gcg.cs.virginia.edu/weatherforecast")]
[WebServiceBindingAttribute(Name="WeatherForecastPortTypeSOAPBinding",
                            Namespace="http://gcg.cs.virginia.edu/weatherforecast")]
public class WeatherForecastPortType : GridServiceSkeleton
{

        ...

   [WebMethod]
   [SoapDocumentMethod("http://gcg.cs.virginia.edu/weatherforecast#getForecast",
                       Binding="WeatherForecastSOAPBinding")]
   public string GetForecast(DateTime forecastDay)
   {
        ...
   }
}
```

Figure 3. Using [SoapDocumentMethod] to Fully Specify Binding Information

It should also be noted that other attributes related to the .NET XML processing architecture can be applied to methods and their parameters (e.g. the [return] attribute). Since these are not specific to OGSI.NET (but effect all Web Services and .NET applications) they are not covered here. Examples can be seen in the various sample services provided with OGSI.NET, e.g. the BasicCounterService.

We have now defined a new port type called WeatherForecastPortType. This port type can be used by any number of services by using the [OGSIPortType] attribute. This attribute allows the service author to say "this service implements this port type". Consider the following service definition.

```
[WsdlBaseName(Name=”WeatherForecast”,
              Namespace=”http://gcg.cs.virginia.edu/weatherforecast”)]
[WebService]
[WebServiceBinding]
[OGSIPortType(typeof(WeatherForecastPortType))]
[OGSIPortType(typeof(GridServicePortType))]
public class WeatherForecastService : GridServiceSkeleton
{
   public WeatherForecastService(int clientAreaCode)
   {
        WeatherForecastPortType pt =
               base.GetPortTypeInstance(typeof(WeatherForecastPortType)) as
               WeatherForecastPortType;
        pt.areaCode = clientAreaCode;
   }
}
```

Figure 4. WeatherForecastService Definition using OGSIPortType Attributes

The WeatherForecastService implements two port types, the WeatherForecastPortType that we defined previously and the GridServicePortType, which all grid services must implement (and which is included in the OGSI.NET libraries). The `[OGSIPortType]` attribute is an attribute that is placed on the service class. This attribute takes a single parameter that is the (C#) System.Type of the port type that this service should expose. The System.Type can be obtained using the `typeof()` function as shown. Also, note that the WeatherForecastService, like the WeatherForecastPortType, must derive off of the GridServiceSkeleton.

In the above example, all the logic for the WeatherForecastService’s methods is contained in the WeatherForecastPortType. This is useful when the WeatherForecastPortType will be used by many services. However, sometimes it is too clumsy to define separate port types and then services to utilize those port types. In this case, OGSI.NET allows a service class to be defined with methods that are annotated like those used in defining a port type. OGSI.NET will create a port type for these methods and have the service implement that port type. Using this method, our WeatherForecastService would look like this:

```
[WebService(Name=”WeatherForecastService”,
            Namespace=”http://gcg.cs.virginia.edu/weatherforecast”)]
[WebServiceBinding(Name=”WeatherForecastServiceSOAPBinding”,
                   Namespace=”http://gcg.cs.virginia.edu/weatherforecast”)]
[OGSIPortType(typeof(GridServicePortType))]
public class WeatherForecastService : GridServiceSkeleton
{
   public double temperature;
   public int windSpeed;
   public int areaCode;

   public WeatherForecastService(int clientAreaCode)
   {
        areaCode = clientAreaCode;
   }

   [WebMethod]
   [SoapDocumentMethod(“http://gcg.cs.virginia.edu/weatherforecast#getForecast”,
                       Binding=”WeatherForecastSOAPBinding”)]
   [return: XmlElement(“forecast”, Namespace=”http://gcg.cs.virginia.edu”)]
   public string GetForecast(DateTime forecastDay)
   {
        string forecastDescription;
        // compute forecast
        return forecastDescription;
   }
}
```

Figure 5. WeatherForecastService Definition without [OGSIPortType] Attribute

The GetForecast method will still be exposed by this service, but there was no need to define a separate port type for it. Of course, this also means that there is no WeatherForecastPortType that can be imported into other services. Note that we could also have used the `[WsdlBaseName]` attribute in this definition.

2.2 Attributes for Data

Grid services will often want to expose certain parts of their internal state to clients. The OGSI specification [2] addresses this with Service Data Elements (SDEs). In OGSI.NET, there are several ways to use SDEs. They can be declared at the class level (either on the service class or port type class) as well as the data member level. By using the `[SDE]` attribute, service data elements can automatically be declared and made accessible to the service's clients using the OGSI specification-defined methods for accessing service data. All the grid service author has to do is manipulate the values of those SDEs (see Appendix A for information on the functionality provided by OGSI.NET for this purpose).

The `[SDE]` attribute can be used with a number of different parameters as described in the following table. Familiarity with SDEs and their configuration, as defined in Section 6 of the OGSI Specification [2], is assumed.

Pos	Name/Type	Description
1	sdeName System.string	The name of the SDE, e.g. currentTemp or windSpeed, to be used in a QueryByServiceDataName.
2	sdeType System.Type	The C# type of the values of the SDE, e.g. typeof(int) means the SDE has integer values.
3	isNillable System.bool	Determines whether this SDE can have the value of null.
4	Mutab ServiceDataTypeMutability	The mutability type of this SDE. Can be static, constant, mutable, or extendable.
5	isModifiable System.bool	Determine whether this SDE can be modified by calls to SetServiceData.
6	minOccurs System.string	String representation of the minimum number of values this SDE can have.
7	maxOccurs System.string	String representation of the maximum number of values this SDE can have.
8	xsdType System.string	The name to be used as the xsd:type when values from this SDE are serialized into XML. This attribute is optional and if not used, OSGI.NET will try and determine this from the sdeType. However, for certain C# types, the xsd:type is ambiguous and so this attribute is provided.

If the WeatherForecastService wanted to expose two SDEs representing the current temperature and wind speed, this can be done using the `[SDE]` attribute at the class level because this data is part of the state of the WeatherForecastPortType.

```
[SDE("currentTemp", typeof(double), false, ServiceDataTypeMutability.mutable,
      false, "1", "1")]
[SDE("windSpeed", typeof(int), false, ServiceDataTypeMutability.mutable,
      false, "1", "1", "int")]
public class WeatherForecastPortType : GridServiceSkeleton
{
```

Figure 6. SDE Declarations using Attributes on a Port Type

Note that the xsdType parameter for the windSpeed SDE is not strictly necessary because the SDE's sdeType is `typeof(int)` (and thus the xsd:type can be inferred).

These two attributes will add 2 SDEs to the WeatherForecastPortType. The grid service author can now access those SDEs using provided helper-routines (see Appendix A) and clients can invoke any of the find/set/delete ServiceData functions of the GridServicePortType on those SDEs (subject to the constraints specified in the SDE attribute's parameters, e.g. modifiability, mutability, min/max occurs, etc.).

OGSI.NET also provides a short-hand mechanism for turning class data members into SDEs. The SDE attribute can again be used for this purpose. In this usage, the attribute is simpler to use than in the above example, but does not provide as much configurability. Suppose the WeatherForecastPortType wanted to expose the currentTemperature member.

```
public class WeatherForecastPortType : GridServiceSkeleton
{
   [SDE]
   public double temperature;

   public int windSpeed;
   public int areaCode;

   public WeatherForecastPortType()
   {
   }
```

Figure 7. Exposing a Public Data Member as an SDE

The simple `[SDE]` annotation on a public member variable (the container only permits this attribute on public members) causes an SDE to be created with the same names as the member variable. The remainder of the SDE's parameters are:

- sdeType = type of the member variable
- isNillable = true
- Mutability = mutable
- isModifiable = false
- minOccurs = 1
- maxOccurs = 1

When a client makes a findServiceData call on the temperature SDE, the container will return the current value of the temperature member variable. To understand how this is done, we introduce the concept to SDE Get and SDE Set Handlers.

An SDE Get Handler is a callback that is called whenever a client makes a findServiceData call on the SDE. This callback returns a set of values for the SDE that are returned to the client. Get Handlers can be useful when the value(s) of an SDE need to be dynamically determined at the time of the findServiceData call. An SDE Set Handler is a callback invoked when a client performs a setServiceData call on the SDE. Set Handlers can be used to perform tasks like saving the new SDE values to permanent storage or sending notification that a change has occurred.

Service authors can write their own Get and Set Handlers and annotate them with an additional attribute `[SDEHandler]`. The function prototypes for these callbacks are:

```
public static ArrayList GetHandler(OGSIServiceData osd, GridServiceSkeleton gss);

public static void SetHandler(OGSIServiceData osd, Object newElem, bool adding,
                              GridServiceSkeleton gss);
```

Figure 8. Prototypes for Get and Set SDE Handler Functions

The first parameter for the Get Handler, `osd`, is the service data element being retrieved (OGSIServiceData is the wrapper class for SDEs used by OGSI.NET). The second parameter, `gss`, is the GridServiceSkeleton for the service instance. Using the GridServiceSkeleton, the Get Handler can access any of the service instance's functions or data members (see Appendix A for more details). The Get Handler returns an ArrayList of values that are returned to the client who called findServiceData.

The Set Handler has both the OGSIServiceData and GridServiceSkeleton parameters as well as two others. The `adding` parameter is a Boolean that is true when values are being added to this SDE and false when they are being deleted from the SDE. The `newElem` parameter is an object that contains the new value to be added or the old value to be deleted.

A function can be designated as a Get or Set Handler is one of two ways. If the function is not defined within the service or port type declaring the SDE, then the `[SDEHandler]` attribute can be placed on the service or port type class. If the Handler is defined within the service or port type, the `[SDEHandler]` attribute can be put on the method itself. For example:

```
[SDE(“windSpeed”, typeof(int), false, ServiceDataTypeMutability.mutable,
     false, “1”, “1”)]
[SDEHandler(SDEHandlerAttribute.HandlerOperation.Get, “windSpeed”,
            typeof(UVa.Grid.OGSIGridServices.SDEValueGetter), “GetCallback”)]
public class WeatherForecastPortType : GridServiceSkeleton
{
```

- or -

```
[SDE(“windSpeed”, typeof(int), false, ServiceDataTypeMutability.mutable,
     false, “1”, “1”)]
public class WeatherForecastPortType : GridServiceSkeleton
{
   int windSpeed;

   [SDEHandler(SDEHandlerAttribute.HandlerOperation.Get,
               “windSpeed”, “http://gcg.cs.virginia.edu”)]
   public static ArrayList CalcWindSpeed(OGSIServiceData osd, GridServiceSkeleton gss)
   {
        ArrayList returnValues = new ArrayList();
        int currentWindSpeed;
        // compute current wind speed
        returnValues.Add(currentWindSpeed);

        return returnValues;
   }
```

Figure 9. Two Options for Specifying SDE Handlers

The first example in Figure 9 shows how the `[SDEHandler]` attribute can be placed on a class. The parameters to the attribute are:

- the type of Handler it designates. The values are either *Get*, *PreSet* or *PostSet* from the SDEHandler.HandlerOperation enumeration. The difference between a PreSet handler and a PostSet handler is that the PreSet handler is called before making the requested modification to the SDE's values while the PostSet handler is called after.)
- the name of the SDE to which it corresponds
- the (C#) Type of the class the implements the Handler method
- the name of the method

The second example in Figure 9 shows the `[SDEHandler]` attribute being used on a member function. In this case, the attribute requires fewer parameters (just the type of the Handler and the name of the SDE). Note that the parameter for calling the Set Handler before or after changing the SDE values is optional and, if omitted, defaults to true. Also note that all Handler functions must be *public* and *static*. The `[SDEHandler]` attribute, in either usage, has an optional namespace parameter (as shown in the second example of Figure 7). Normally, the namespace of any SDE is defined by the port type or service that that `[SDE]` attribute decorates. The `[SDEHandler]` attribute assumes that if it is not given a namespace for its corresponding SDE, it should use the namespace of the class it decorates (even if it decorates a method within that class). These namespaces come from either the `[WsdlBaseName]` or `[WebServiceBinding]` attributes as described above. However, in some cases it may be necessary to specify the name and namespace of a handler's corresponding SDE and this is done with the optional namespace parameter.

Returning to the WeatherForecastPortType example in Figure 7, the temperature SDE will automatically be assigned a Get Handler that uses reflection to retrieve the current value of the `temperature` variable. This allows the grid service author to modify only the `temperature` variable and those changes will be automatically reflected in the temperature SDE.

2.3 Attributes for Policy

OGSI.NET provides annotations to specify role-based security policy for grid services using a set of policy attributes. Currently, 2 attributes, `[Confidentiality]` and `[Authorization]` are used. These attributes can be placed on a service class, port type of method. Policy attributes on a method apply to that method only. Policy attributes on a port type class apply to all methods of that port type, unless overridden by policy attributes on specific methods. Policy attributes on a service class apply to all port types of that service (and thus all the service's methods), unless overridden by policy attributes on port type classes or individual methods. We describe the policy attributes below.

A service deployer (separate from the service author) can define the security credential(s) that can be used to verify that a given client can play the role specified in the policy attribute. For example, a service author might use attributes to specify that a function can only be invoked by an "administrator", while the service deployer would define that the administrator is Joe Smith. Currently, both X509 and Username tokens can be used to prove a client is in a particular role. See section 4 for information on defining security credentials for roles.

2.3.1 Authorization Attribute

The `[Authorization]` attribute allows the grid service author to specify that an entity playing a given "role" that can access a service, port type or function. The `[Authorization]` attribute takes a single string parameter called "roleName". RoleName is an arbitrary string which the service deployer will use in the ogsi.config file when defining the necessary security credentials to prove membership in the role. The `[Authorization]` attribute causes the GridServiceWrapper to verify that any incoming message to the service, port type or method is digitally signed by one of the credentials specified by the deployer. If multiple, distinct role names (and thus multiple, distinct role definitions by the deployer) are desired, multiple `[Authorization]` attributes can be used.

```
[Authorization("weatherman")]
public class WeatherForecastService : GridServiceSkeleton
{


<service name="WeatherForecastService">
...
   <parameter name="weatherman" value="X509:"/>
</service>
```

Figure 10. Authorization Attributes Decorating the WeatherForecastService

Figure 10 shows the definition of the WeatherForecastService and a corresponding deployment section for the service in the ogsi.config file (see Section 4 for deployment details). The `[Authorization]` attribute says that the WeatherForecastService can only be accessed by clients who can fill the "weatherman" role. The deployment section defines the weatherman role to be anyone who can sign a message with an X509 certificate (essentially, the service deployer has said that all messages to this service must have verifiable integrity and non-repudiation, but has not placed any restrictions on who can access the service). Section 4 provides complete information of service deployment and how to define policy roles.

2.3.2 The Confidentiality Attribute

This attribute allows the grid service author to specify encryption requirements for incoming messages to their service, port type or method. It has the same parameter as the `[Authorization]` attribute and is processed the same way by the container, except that it specifies policy with respect to encryption and not digital signatures. So, one of the security credentials defined for the role specified by a `[Confidentiality]` attribute must be used to encrypt incoming messages to the service, port type or function that the attribute is placed on.

Note that both the `[Confidentiality]` and `[Authorization]` attributes can use the same role names. In that case, the same set of credentials (as defined by the service deployer) must be used to both sign and encrypt a message. However, if the role can be fulfilled by multiple security credentials, then the credential used to sign and the credential used to encrypt need not be the same.

While it is possible to handle authorization based on the credential that was used to perform encryption, OGSI.NET does not perform both of these functions via the `[Confidentiality]` attribute. It is imagined that the `[Authorization]` attribute will have "tighter" role definitions that the `[Confidentiality]` attribute. For example, a service's policy might be that all messages be encrypted with X509 encryption (i.e. using any X509 certificate), but only X509 certificates with a specified subject DN can access a certain function.

3 Using the OGSI.NET Tools

After appropriately annotating a service, the OGSI.NET WsdlGenerator can be used to generate the WSDL for the service and client stubs.

In the {OGSI.NET_install_dir}/Bin directory is WsdlGenerator.exe. This program takes two arguments, an assembly file (specified with –a) and a class name (specified with –c). The assembly file must contain the class name specified. The WsdlGenerator will produce port type, service and/or bindings files depending upon the attributes used to decorate the class (see section 2.1). For example:

```
bash-2.05b$ WsdlGenerator -a c:/OGSIdotNet-current/OGSIdotNet/Bin/UVa.Grid.OGSI
SampleServices.dll -c UVa.Grid.OGSISampleServices.SDE.SDESampleService
```

See the WsdlGenerator help for more options.

4 Deploying the Service

Services are deployed in the OGSI.NET container by putting the service DLL and WSDL in the appropriate location and then modifying the OGSI.config file to reflect the new service. The DLL generated by compiling your service with VS.NET should be placed in the {OGSI.NET_install_dir}/Bin directory. The WSDL files generated by the WsdlGenerator should be put within the schema virtual directory hierarchy you created when you installed OGSI.NET.

The last step is to modify the OGSI.config file, which is in the {OGSI.NET_install_dir}/Configuration directory. This file contains one entry for every service that is to be started when the OGSI.NET container starts. Typically, these are services that are needed whenever the container is running, such as the HandleResolverService or various Factory services that create new grid service instances. The OGSI.config file that comes with OGSI.NET has many examples of how to deploy services.

The file begins with a <deployment> element which contains two sub-elements, <globalConfiguration> and <service>. Each of these sub-elements contains a number of name/value pairs denoted by <parameter> elements. The <globalConfiguration> parameters will be passed to every service (and service instance) that is started within the container. These parameters can be anything that the container administrator wants to provide to each service. It is up to the service's logic to use these values (see Appendix A for details on how to access them). Two parameters that are used by the container itself are "schemaRoot" and "httpPort"[1]. The value of the schemaRoot parameter is the path to the virtual directory in which the schema virtual directory was created when OGSI.NET was installed. This is often "http://*<your fully qualified domain name>*/" because the schema virtual directory is typically installed at the root of the virtual directory system. The httpPort value is typically "80" (IIS's default port for HTTP traffic).

The <service> elements provide information to allow the container to load the service. They can also provide arbitrary name/value pairs that will be passed to new service instances on startup. Each <service> element has a name attribute (this is an attribute in the sense of .NET's XmlElements and not in the sense of the C# attributes discussed above). This name will be used as the GSH of the service with `{schemaRoot}:{httpPort}/ogsa/services` prepended.

The <parameter> sub-elements of the <service> element can be arbitrary name/value pairs, but each <service> element must have 3 well-known parameters named schemaPath, class and messageHandlers. The schemaPath parameter is the path to the service's WSDL when appended to the schemaRoot. For example, consider the following <service> element.

```
<service name="core/handle/HandleResolverService">
   <parameter name="schemaPath"
        value="schema/core/handle/handle_resolver_service.wsdl"/>
   <parameter name="class" value="UVa.Grid.OGSICoreServices.HandleResolverService,
        OGSICoreServices.dll"/>
   <parameter name="messageHandlers"
        value="UVa.Grid.SoapMessageHandlerLib.SoapMessageHandler,
        SoapMessageHandlerLib.dll”
</service>
```

Figure 11. Example Service Deployment Element for the OGSI.config File

If the schemaRoot for this container was "http://localhost", then the GSH of this service would be http://localhost:80/core/handle/HandleResolverService and its WSDL would be located at http://localhost/schema/core/handle/handle_resolver_service.wsdl. The "class" parameter is a string whose format is `“{.NET class name}, {.dll name}”`. The .NET class name is the fully qualified type name of the class that implements this service. The .dll name is the name of the DLL that contains the .NET class name's implementation (if a full path is not provided, OGSI.NET will load the DLL from the Bin directory). In this case, the type UVa.Grid.OGSICoreServices.HandleResolverService implements this service and that class is defined in the file OGSICoreServices.dll. The final parameter, "messageHandlers", specifies that classes that handle the various messaging formats that the service supports. OGSI.NET provides a SOAP message handler and the value of this parameter shown above will be sufficient for most services. The format of the value is the same as the format of the value of the "class" parameter.

There are an additional set of 3 parameters that apply to a common type of service, the FactoryService. Many grid service instances are created by factories and OGSI.NET provides a FactoryService that can be used to create instances of most services. Consider the following <service> element for a FactoryService that will create instances of the WeatherForecastService.

[1] A third parameter "remotingPort" is part of the experimental support for Microsoft's remoting protocol and is not covered in this document.

```
<service name=”samples/weather/WeatherForecastFactoryService>
   <parameter name=”schemaPath” value=”schema/core/factory/factory_service.wsdl”>
   <parameter name=”class” value=”UVa.Grid.OGSISamples.Factory.BasicFactoryService,
                                  OGSISampleServices.dll”>
   <parameter name=”messageHandlers”
              value=”UVa.Grid.SoapMessageHandlerLib.SoapMessageHandler,
                     SoapMessageHandlerLib.dll”>
   <parameter name=”createSchemaPath”
              value=”schema/weather/WeatherForecastService.wsdl”>
   <parameter name=”createClass”
              value=”UVa.Grid.SampleServices.WeatherForecastService,
                     SampleServices.dll”>
   <parameter name=”createMessageHandlers”
              value=”UVa.Grid.SoapMessageHandlerLib.SoapMessageHandler,
                     SoapMessageHandlerLib.dll”>
</service>
```

Figure 12. Example <service> Element for the WeatherForecastService's Factory

The first three parameters define OGSI.NET’s BasicFactoryService. The next three parameters tell the factory what to create. They are essentially equivalent to the first 3 parameters (except with the word “create” prepended to their names), but they refer to the service instances to be created rather than to the factory itself. In this example, the WSDL of the created instances can be found at *schemaRoot*/schema/weather/WeatherForecastService.wsdl, the implementation for the service is UVa.Grid.SampleServices.WeatherForecastService (in SampleServices.dll) and the new service will use the SOAP message handler provided with OGSI.NET. Notice that there is no name provided for the newly created services. This is because the name for new services is actually part of the createService call to the factory and so is specified by the calling client.

After editing the OGSI.config file and placing the WSDL and .dll files in the right locations, restart the OGSI.NET container by going to the “Services” control panel (i.e. Control Panels -> Administrative Tools -> Services). Left click on the “OGSIContainerService” and then select “restart” (or select “stop” and then select “start”). Your new service (or its factory) should now be running!

4.1 Policy Role Definitions

OGSI.NET supports role-based security policy for digital signing and encryption of messages. As discussed in Section 2.3, the `[Authorization]` and `[Confidentiality]` attributes define “security roles” and the service deployer defines the credentials that can be used to prove that an entity is able to play a given role.

Roles are defined using the <parameter> sub-element of the service’s <service> element in the ogsi.config file. The “name” of the parameter must correspond to the role name defined in one of the service’s policy attributes. The “value” of the parameter defines the acceptable tokens that can be used. Currently, both X509 and Web Service Enhancements (WSE) Username tokens are supported and both can be used in defining a given role[2].

X509 tokens can be defined in terms of Subject DN and Issuer DN. Consider the following examples.

[2] Username tokens can not be used for encryption in the WSE and therefore cannot be used to define `[Confidentiality]` roles.

```
<parameter name=”role1” value=”X509:subject=C=US, O=University of Virginia, OU=UVA
Standard PKI User, E=gsw2c@Virginia.EDU, CN=Glenn S. Wasson 2” />

<parameter name=”role2” value=”X509:issuer=C=US, ST=Virginia, L=Charlottesville,
O=University of Virginia, E=pkimaster@virginia.edu, CN=UVA Standard Assurance SKP 1”/>

<parameter name=”role3” value=”X509:subject=... || issuer=... />

<parameter name=”role4” value=”X509:subject=...;X509:subject=... />

<parameter name=”role5” value=”Username:username=Glenn Wasson />

<parameter name=”role6” value=”Username:username=Glenn Wasson;X509:issuer=... />

<parameter name=”role7” value=”X509:;Username:” />

<parameter name="UsernameTokenPasswordClass"
value="PKGServiceNS.PKGUsernameTokenManager, PKGService" />
```

Figure 13. X509 and Username token-based Policy Role Definitions

All X509 roles are defined by <parameter> elements whose “value” is a string that starts with “X509:”. This can be followed by “subject=” (for defining X509 tokens by subject DN) or “issuer=” (for defining X509 tokens by the issuing CA). The first <parameter> element of Figure 13 shows a role definition based on a particular subject DN. The second parameter element shows a role definition based on an issuer DN. The third <parameter> element shows how subject and issuer DNs can be combined in defining a token (using the “||” separator). Finally, multiple X509 tokens can be specified by separating their definitions with a semi-colon and beginning the next definition with “X509:”, as shown in the fourth <parameter> element.

OGSI.NET can use Microsoft’s Web Service Enhancements 2.0 [3] Username tokens as well. A Username token consists of a username and password and can be used to digitally sign a message (there is no notion of message encryption with Username tokens). The fifth <parameter> element in Figure 13 shows how to specify a Username token. The value portion of the parameter starts with “Username:username=” and is followed by the string specifying the username that will be embedded in the token. The sixth <parameter> element shows that Username and X509 tokens can be combined in defining a role by using a semi-colon between definitions. The seventh <parameter> element shows how to define a role that merely requires some form of credential to be used. Role7 is defined as being played by anyone who signs a message with an X509 certificate or a Username token. This type of “loose” role definition may be wanted for services that are more interested in accounting than in restricting access.

OGSI.NET does not allow service deployers to specify the password for a Username token in the OGSI.config file because that file is stored in cleartext and because it is not necessary given the operation of the WSE pipeline. Instead, another service parameter named “UsernameTokenPasswordClass” is used to specify a class that given a username can determine the correct password. The WSE pipeline will use this class to verify that the password in any message signed with a username token is correct. If it is not correct, the WSE pipeline will have throw an exception before the OGSI.NET container would have begun verifying that the message complies with the stated policy. If the password is correct, then the WSE has already validated the password and it is not necessary for OGSI.NET to perform any further verification. The value portion of the UsernameTokenPasswordClass parameter is similar to the specification of classes in the messageHandler parameters; it consists of a fully qualified class name (i.e. name and namespace) and the assembly that contains that class. However, when specifying the assembly name, **do not** use the .dll extension. While it is appropriate to use this extension when specifying the messageHandlers, it is **not** appropriate when specifying the UsernameTokenPasswordClass and will cause the WSE to not be able to load the correct class. The final <parameter> element of Figure 13 shows an example.

5 Appendix A

This section describes the various ways that the OGSI.NET container's functionality can be used programmatically within a service's service logic (as opposed to handled via attributes).

How to call a port type's members from within a service:
Sometimes, a grid service might want to access one of the members (functions/data) of a port type that the service supports. For example, suppose a service supported the NotificationSourcePortType (via `[OGSIPortType(typeof(NotificationSourcePortType))]`). The following code provides access to the port type's members.

```
NotificationSourcePortType ns= GetPortTypeInstance(typeof(NotificationSourcePortType))
                               as NotificationSourcePortType;

XmlElement message = MessageHandler.SerializeObject(myValue, typeof(int),
                                                    new XmlQualifiedName("value", ""))

XmlQualifiedName topicName =
       new XmlQualifiedName(“Update”, “http://gcg.cs.virginia.edu”);

ns.DeliverNotification(topicName, message);
```

Figure 14. Accessing a Port Type imported with [OGSIPortType]

The GetPortTypeInstance() function returns a port type of the type given as a parameter. Recall that each service will expose only one port type of a particular type.

How to Serialize My Own Arbitrary Object:
OGSI.NET provides a helper function for serializing arbitrary objects into XML. This function is a wrapper for the .NET XmlSerializer, so all the usual attributes that are used to control XML serialization can be used on an arbitrary type defined by the grid service author. OGSI.NET's function can be seen in Figure 14 and is a static function defined on the MessageHandler class. This function takes three parameters, the type of the object to be serialized, the actual object itself and an XmlQualifiedName which can be null. The result of this call is an XmlElement that represents the serialization of the object with its XmlRootAttribute set to the value of the XmlQualifiedName (if provided).

How to Deliver a Notification:
Notifications are an important part of OGSI and OGSA. They are asynchronous messages delivered in a publish/subscribe model. OGSI.NET provides a DeliverNotification() function that can be used to send a given message to all subscribers to a particular topic. This function can be seen in Figure 14. The function takes two parameters, an XmlQualifiedName that denotes the topic of the message (the message will be delivered to all subscribers to that topic) and the actual message itself as an XmlElement.

How to Inspect and/or Modify an SDE Declared at the Class Level:
When an SDE is declared by annotating a data member with the `[SDE]` attribute, the value of that SDE can be inspected/modified by manipulating that data member itself. However, when an SDE is declared at the class level (either on the service class or one of the port type classes), then you can accessed it in OGSI.NET using the GetServiceData function of the GridServiceSkeleton.

```
OGSIServiceData mySDE = GetServiceData(new XmlQualifiedName(“myName”, “myNS”));

// getting an SDE’s values
ArrayList theValues = mySDE.SDValues;

// setting / modifying an SDE’s values
mySDE.Add(newValue);
mySDE.Remove(oldValue);
mySDE.RemoveAt(2);
mySDE.Replace(new ArrayList {...});
mySDE.Clear();
mySDE[1] = newValue;
```

Figure 15. Inspecting and Modifying the Values of an SDE Declared at the Class Level

The GetServiceData function takes a single XmlQualifiedName, which is the unique name of the SDE and corresponds to the sdeName and sdeNS parameters of the SDE’s declaration (see section 2.2). The values of an SDE can be retrieved in an ArrayList using the SDValues property of the OGSIServiceData object returned by GetServiceData. There are a number of ways to modify the service data element’s values. All the functions in Figure 15 will call the SetHandler (if appropriate) and will fail if the number of elements after the requested alteration would conflict with the minOccurs or maxOccurs of the SDE. The Add() function adds a single new value of the SDE’s value list. Remove() removes the object that matches from parameter from the SDE’s value list. RemoveAt() removes the value at a particular index in the SDE’s value list. Replace swaps the current value list for the one passed in as a parameter. Clear() removes all SDE values. Finally, the individual elements of the SDE value list can be modified via standard array notation. Note that you cannot “get” the values of individual elements this way (i.e., x = mySDE[1] is illegal) because that should call the GetHandler and that function returns a list not a single value. SDE values should be retrieved via the SDValues property.

How to Get the SoapContext Object for Messages Processed by the WSE Pipeline:
How to Get Other Information Contained in a SOAP Message’s Headers:
OGSI.NET is compatible with the Web Service Extensions (WSE) 2.0 [3]. Programmers familiar with writing web services in VS.NET will be familiar with the SoapContext object, which holds the results of processing a message’s soap headers (for example, processing the WS-Security headers). OGSI.NET provides the same functionality and any request message to a service will be processed by the WSE. To access the SoapContext object, use the following.

```
SoapContext requestContext = InputMessageHeaders["SoapContext"] as SoapContext;

XmlQualifiedName qn = new XmlQualifiedName(“invocationID”, “http://gcg.cs....”);
InvocationID invocationID = InputMessageHeaders[qn] as InvocationID;
```

Figure 16. Using the InputMessageHeaders Hashtable

OGSI.NET provides a hash table called InputMessageHeaders. The container populates this hash table with:

- the SoapContext object that results from running the WSE pipeline over a message’s headers
- the deserialization of the contents of any remaining SOAP headers
- the HTTP headers (if HTTP was used as the messages transport)

The key for the SoapContext object in the hash table is “SoapContext” as shown in Figure 16. The keys for the deserialized objects that were encoded in other SOAP headers are the XmlQualifiedNames of those headers. So for example, the SOAP header <ns1:invocationID xmlns:ns1=”http://gcg.cs....”> *serialized InvocationID object* </ns1:invocationID> would cause the deserialized InvocationID object to be associated with the key qn, as shown above.

In order to describe to the OGSI.NET container how to deserialize (arbitrary) data placed in SOAP headers, the attribute `[OGSISoapHeader]` is used on a service class or port type. This attribute maps a headers qname to a data type for the header’s value. When `[OGSISoapHeader]` is used, the container examines the headers of all incoming messages from the qname in the attribute. The contents of that header will be deserialized as the data type defined in the attribute and the resulting object will be placed in the InputMessageHeaders hash table with the qname as its key. For example, the attribute `[OGSISoapHeader(“invocationID”, “http://gcg.cs....”, typeof(InvocationID))]` means that the container should deserialize the contents of the <ns1:invocationID xmlns:ns1=”http://gcg.cs....”> header as an InvocationID object and place it in the InputMessageHeaders hash table under the key XmlQualifiedName(“invocationID”, “http://gcg.cs....”).

How to Add Data to the Outgoing Headers for a SOAP Message:
The GridServiceSkeleton class contains a hash table called OutputMessageHeaders. The keys in this hash table are XmlQualifiedNames and the values are System.Objects. Any key/value pair in this table will have the value serialized and placed in an outgoing SOAP message’s headers in an element whose name will be the QName of the key. An exception to this is when using WSE to construct output message headers. In this case, the service can create and populate a SoapContext object as normal (see [2] for documentation on the WSE) and then add it to the OutputMessageHeaders hash table under the key “SoapContext” as shown below. Note that in the case of a SoapContext object, the hash table key is a System.String. In all other cases, the key is an XmlQualifiedName.

```
int foo;
// compute foo
XmlQualifiedName headerName= new XmlQualifiedName(“foo”,“http://gcg.cs.virginia.edu”);
OutputMessageHeaders.Add(headerName, foo);

SoapEnvelope env = new SoapEnvelope();
SoapContext responseContext = env.Context();
X509SecurityToken token = GetSecurityToken(...);    // get a security token
Signature sig = new Signature(token);
responseContext.Security.Elements.Add(sig);
responseContext.Security.Tokens.Add(token);

OutputMessageHeaders.Add(“SoapContext”, responseContext);
```

Figure 17. Modifying the Headers of Outgoing SOAP Messages

How to Pass Parameters to a Service’s Constructor from a Custom Factory:
Many factory services provide exactly the same functionality; they only differ in the type of service that they create. For service instances that don’t need customized construction, OGSI.NET’s BasicFactoryService (and an appropriate deployment descriptor as shown in section 4) is sufficient. For factories that need to pass custom parameters to the constructors of the service instances it creates, OGSI.NET provides a callback mechanism that grid service authors can use to transform the creationParameters XmlElement passed into the FactoryPortType’s createService method into a set of (C#) parameters that will be used as arguments to the service instance’s constructor.

```
[OGSIPortType(typeof(FactoryPortType))]
public class SignedFactoryService : GridServiceSkeleton
{
   public SignedFactoryService()
   {
        FactoryPortType factory = GetPortTypeInstance(typeof(FactoryPortType))
                                 as FactoryPortType;
        factory.SetChildCreationCallback(new
                    FactoryPortType.ChildCreationCallback(childCreationCallback));
   }

   private object[] childCreationCallback(NameValueCollection childServiceParams,
                                        XmlElement creationParams)
   {
        // examine the service parameters and the creationParams element passed in
        // to the createService() call and determine the constructor params for
        // the new service instance
        return new object[] {...};
   }
}
```

Figure 18. Passing Constructor Parameters to a Service Instance Created by a Factory

A service implementing the FactoryPortType can call that port type's SetChildCreationCallback() function to define a method to be called when the FactoryPortType's createService() method is invoked. The createService() method will use this callback to retrieve an array of System.object elements that will be passed to the constructor of new services instances. The order of the elements in this array must match the order of the parameters in the service's constructor. In other words, a service constructor `public MyService(int p1, double p2)` would require a callback that returned an array with an int in position 0 and a double in position 2 (recall that all types derive from System.object and so any data type can be placed in a System.object array). Also, all constructor parameters must be Serializable (in the .NET sense) because they will be passed between ApplicationDomains from the Factory to the new service instance.

As shown in Figure 18, the creation callback must take a NameValueCollection and an XmlElement as parameters and return an object array. The NameValueCollection will be filled with the service parameters for the new to-be-created service from the factory's deployment descriptor (see section 4) as well as the FactoryLocator stored under the key "factoryLocator". The XmlElement parameter is the element that is passed in the createService() call from the client.

So, in order to create a factory that sends constructor parameters to the services it creates, first create a service that imports the FactoryPortType with the OGSIPortType attribute. Then define a callback with the function prototype shown in Figure 18. Finally, in the factory's constructor, get the FactoryPortType object and call SetChildCreationCallback(), passing in the name of callback.

How to Return an extensibilityOutput Element from the createService Method of a FactoryPortType:
If a factory service wishes to return an extensibilityOutput element from the createService method, (see the OGSI FactoryPortType in [2]), the service being created by the factory needs to implement the interface `UVa.Grid.OGSIGridServices.IExtensibilityOutput`. This interface has a single method: `XmlElement GetExtensibilityOutput()` which should return the extensibilityOutput element. If a service class implements this interface, the createService method of the FactoryPortType will call `GetExtensibilityOutput()` when creating a new instance of that service and return the extensibilityOutput element to the client issuing the createService request.

How to Create Co-Located Services:
Normally, each service instance in OGSI.NET is stored in a separate AppDomain within the container process. While this provides excellent separation of services and protects services from

each other, it increases the per service memory footprint. Co-located services are services that exist in the same AppDomain as other services. These service instances can reuse the infrastructure set up for the AppDomain (e.g. shared libraries) and so has a smaller memory footprint. In all other ways, they are the same as other grid services. In order to make a factory create its service instances in the same AppDomain (i.e. in order to make a factory create co-located services), add the following line to the factory's configuration section of the OGSI.config file.

```
<parameter name="createServiceDomain" value="shared"/>
```

Sometimes it is useful for a service A to create another service B, when A is not a factory, i.e. A does not implement the FactoryPortType. For example, it can be useful to have the serviceGroup service create serviceGroupEntry services or for the notificationSource service to create notificationSubscription services. In these cases, it is efficient to create the new service in the same AppDomain as the "parent" service because there may be many such child services and each often contains a small amount of data (often 1 or 2 SDEs). In order to support this, OGSI.NET provides the `InternalServiceFactory`, whose use is shown in Figure 19.

```
namespace UVa.Grid.ColocatedSample
{
   [WsdlBaseName("My", "http://www.cs.virginia.edu/GCG/Samples")]
   [WebServiceBinding]
   [OGSIPortType(typeof(GridServicePortType))]
   public class MyPortType : GridServiceSkeleton
   {
      private InternalFactoryService childServiceFactory;

      public MyPortType()
      {
         childServiceFactory = new InternalFactoryService(typeof(ChildService));
      }

      [WebMethod()]
      [return: XmlElement("childLocator")]
      public LocatorType doSomething()
      {
         // create a child service, the child's default constructor gets invoked and
         // the initial termination time is the same as the parent service.
         GridServiceSkeleton childService = childServiceFactory.CreateService(new 
object[] {});
         LocatorType childLocator = new LocatorType();
         childLocator.handles = new string[] { childService.ServiceHandle };
         return childLocator;
      }
    }

    [WsdlBaseName("child", "http://www.cs.virginia.edu/GCG/Samples")]
    [WebService]
    [OGSIPortType(typeof(GridServicePortType))]
    public class ChildService : GridServiceSkeleton
    {
       public ChildService()
       {
       }
    }
}
```

Figure 19. Using the InternalFactoryService to Create Co-Located Services

How to Access the Global and Service Parameters from the OGSI.config File:
Global parameters and service parameters for each service can be specified in the OGSI.config file. These parameters are accessible to a service's service logic using the GridServiceSkeleton's GlobalParams and ServiceParams properties. These properties are NameValueCollections. So, a service parameter `<parameter name="param1" value="someValue">` from the OGSI.config file would be accessible to the service as `ServiceParams["param1"]`.

How to Create Client Stubs for a Service:
Currently, client stubs generated via VS.NET's "Add Web Reference" mechanism are not guaranteed to work with OGSI.NET (although they may). To generate client stubs, run the GWsdlFlattener (GWsdlFlattener.exe) on the service's GWSDL generated by OGSI.NET's WsdlGenerator. The flattened WSDL can then be feed into VS.NET's wsdl.exe and client stubs generated normally.

However, any WSDL bindings generated by the WsdlGenerator will still refer to the original GWSDL files. If when deploying the service, you put the flattened WSDL into the schema directory, you will need to edit the WSDL bindings file to point to the flattened WSDL. This should be a change in a single line that includes the GWSDL file to instead include the WSDL file (typically you'd change xyz.gwsdl to xyz.wsdl).

How to Throw Faults in OGSI.NET Services:
The OGSI specification defines a number of types of faults that services may wish to throw. Grid Service authors may define their own types of faults (which may derive off the OGSI-specification defined FaultType or not). No matter what type of fault a service throws, the OGSI.NET container should send an appropriate XML representation of the fault type to the client that invoked the fault-causing function. This can be accomplished as follows:

```
if (/*can not find service*/)
{
   throw new GridServiceException(typeof(NoSuchServiceFaultType),
                               "Could not find the service");
}
```

OR

```
if (/*that service exists already*/)
{
   ServiceAlreadyExistsFaultType fault = new ServiceAlreadyExistsFaultType();
   fault.existingService = /*some locator type*/;

   throw new GridServiceException(fault);
}
```

OR

```
if (/*something weird happens*/)
{
   MyStrangeFaultObject msfo = new MyStrangeFaultObject();
   throw new GridServiceException(msfo);
}
```

The GridServiceException class (which is derived off the System.Exception class) has a number of constructors and the actions of the OGSI.NET container when throwing a GridServiceException depend on the constructor used. The first constructor takes a System.Type and a string message. The value for the Type parameter must be a type that is derived off of UVa.Grid.OGSICoreServices.Bindings.GridService.FaultType. This is the C# type that corresponds to the base type for all faults defined in the OGSI specification. When a GridServiceException constructed in this way is thrown, an object of the given Type will be serialized into the "detail" element of a SOAP fault message which is then sent to the client. The OGSI.NET container will automatically "fill in" various fields of the base FaultType (adding a timestamp, a LocatorType of the service throwing the exception, a chained array of FaultTypes representing the exception chain and an XmlElement called "extensions" in which the stack trace is placed). The "description" field of the base FaultType is set to the value of the "message" parameter given in the GridServiceException constructor. Use this option when throwing one of the OGSI specification faults that differs from the base fault type in name only (i.e. it add no new fields).

The second option for throwing a GridServiceException is to create a new fault object (derived off UVa.Grid.OGSICoreServices.Bindings.GridService.FaultType) and populate its fields. This object is passed to the GridServiceException constructor and when that GridServiceException is thrown, the given object will be sent to the client wrapped in the "detail" element of a SOAP fault. In this case, the OGSI.NET container will only fill in fields of the base FaultType, and only if they are still set to null (i.e. if the grid service author fills them in with a value, the container will just pass it along). Use this option when throwing an exception that extends the base OGSI fault types with custom data fields.

Finally, a grid service author may wish to send an arbitrary message to a client when an exception occurs and not one of the OGSI specification faults (or a fault derived of the base fault type). To use this option, use the GridServiceException constructor that takes a single System.Object as a parameter. When this exception is thrown, the OGSI.NET container will serialize the object into the detail element of a SOAP fault. No other processing is done.

How to Catch Faults in Clients for OGSI.NET Services:
The OGSI.NET client libraries provide some helpful routines for catching faults thrown by grid services. These routines deserialize SOAP fault messages into C# types that represent the thrown exception.

```
try
{
   loc = factory.createService(requestTT, null, out serviceTT, out eo);
}
catch (System.Web.Services.Protocols.SoapException e)
{
   object o = FaultMapper.GetFaultType((XmlElement) e.Detail);
   /* test o’s Type(i.e. if (o.GetType() == typeof(ServiceAlreadyExistsFaultType)) */
}
```

Figure 20. Catching Faults on the Client-side

The FaultMapper.GetFaultType() method can be passed the detail element of a SOAP message. The output will be a deserialized object that represents the C# type of the fault if the thrown fault is one of the OGSI specification defined faults. This is done by testing the name of the root element of the detail element and deserializing the element to that type. If the thrown fault is not one of the OGSI defined faults, the method simply hands back the XmlElement representation of the detail element.

How to Perform Cleanup on Service Termination:
When a service instance is scheduled to be destroyed, a ServiceTerminationEvent is fired by the container. Services can register to receive this event and perform any service specific cleanup they require before the container removes them. A service termination event handler should have the following signature:

```
void handler (object sender, EventArgs e)
```

To have this event handler called when a ServiceTerminationEvent event is raised, place the following code in the service's constructor (or elsewhere, but the registration must occur before the handler can handle the event):

```
base.ServiceTermination += new ServiceTerminationEventHandler(handler);
```

where `handler` is the name of your method whose signature is shown above.

If a service wants to catch termination events for co-located services (see above), they can use the GridServiceSkeleton returned by the InternalFactoryService. For example:

```
InternalFactoryService childServiceFactory = new
               InternalServiceFactory(typeof(ChildService));
GridServiceSkeleton child = childServiceFactory.CreateService(...);
Child.ServiceTermination += new ServiceTerminationEventHandler(handler);
```

Figure 21. Registering a Handler for a Co-Located Service's Termination

6 References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S. 2002. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002. http://www.globus.org/research/papers/ogsa.pdf.
2. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maguire, T., Sandholm, T., Vanderbilt, P. and Snelling, D. 2003. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Draft Recommendation, 6/27/2003. http://www.globus.org/research/papers/Final_OGSI_Specification_V1.0.pdf.
3. Powell, M. 2003. Programming with Web Services Enhancements 2.0, July 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/programwse2.asp