MS.NETGrid OGSI User Manual

Project Title: MS.NETGrid

Document Title: MS.NETGrid OGSI User Manual

Document Identifier: MS.NETGrid-OGSIUserManual-v1.1

Authorship: Ally Hume, Daragh Byrne

Document History:

Personnel	Date	Summary	Version
EPCC	16th October 2003	EPCC Approved	1.1
EPCC	16th June 2003	EPCC Approved	1.0
MJJ	13th June 2003	Second Draft – Changes and Comments	0.2
ACH	10th June 2003	First Draft	0.1

Approval List: *EPCC: Project Leader, Technical Staff, Technical Reviewer, Coach*

Mike Jackson, Daragh Byrne, Ali Anjomshoaa, Neil Chue Hong

Approval List: *Microsoft Research Limited: Managing Director, University Relations x 3*

Andrew Herbert, Fabien Petitcolas, Van Eden, Dan Fay

Contents

1 Introduction

MS.NETGrid is an ASP.NET-based implementation of the Open Grid Services Infrastructure. This user manual is aimed at the Grid service developers and describes how to use the MS.NETGrid implementation to create Grid services. Details of how to write clients that use these services – in the form of running examples – are also included. This document complements [*MS.NETGridOgsiDesignOverview*] that is included with this distribution.

1.1 Product Licence

The MS.NETGrid software is released under the licence described in LICENSE.TXT under the root directory of the distribution.

1.2 Project WWW Site

The MS.NETGrid Project WWW site is at:

http://www.epcc.ed.ac.uk/~ogsanet

1.3 Support and Queries

Please note that our software is intended as a proof-of-concept to demonstrate the applicability of Microsoft .NET in general and ASP.NET in particular to the development of Grid services rather than as a fully-functioning product. In consequence we do not provide any support for our software. However please feel free to forward any comments, queries, suggestions or problems you have to:

ogsanet-queries@epcc.ed.ac.uk

2 Installation

The download consists of a zip file called `MS.NETGridOGSI-TP1.1.zip`. Extract this to a convenient directory of your choice. We recommend that you install at the root of your `C:` drive, so that upon unzipping you have a folder called `C:\MS.NETGrid`. At this location, there will be a directory called `MS.NETGrid`. This directory will have a number of subdirectories one of which is called `Ogsi.Container`. This `Ogsi.Container` directory contains the ASP.NET-based OGSI container, as well as all source code for the container.

The `Ogsi.Container` directory must be set up as a virtual directory under Internet Information Services (IIS). This can be done using Microsoft's Internet Services Manager; see the user documentation for this product for details of creating a virtual directory. **Call the virtual directory `ogsa`.** Further information about creating a virtual directory under IIS may be found at http://localhost/iishelp/iis/htm/core/iicodirv.htm if you have a standard installation of IIS and are logged in as an administrator.

Throughout the rest of this document the notation *<OGSIContainer>* refers to the full path name of the `Ogsi.Container` directory, e.g, `C:\MS.NETGrid\Ogsi.Container`. *<install_dir>* refers to the directory into which the `MS.NETGrid` directory was unzipped, e.g. `C:\`.

2.1 Configuration

The file `Web.config` in the directory *<install_dir>*`\MS.NETGrid\Ogsi.Container` can now be edited to suit the server machine. Open this file using an XML editor (notepad will do) and navigate to the XML node `configuration/gridContainer.config/containerProperties`. This node contains a number of child nodes that look like the following:

```
<containerProperties>
      ......
      <add key="persistentServiceProxyDirectory" value="persistent" />
      <add key="transientServiceProxyDirectory" value="transient" />
      <add key="serviceDirectory" value="services" />

      ......
     <!--
       This bit needs to be changed.
       -->
      <add key="domain" value="www.mydomain.org" />
      <add key="port" value="80" />
      ......
       <add key="addTextTraceListener" value="false" />
       <add key="traceFileRoot" value="C:\Log" />
   </containerProperties>
```

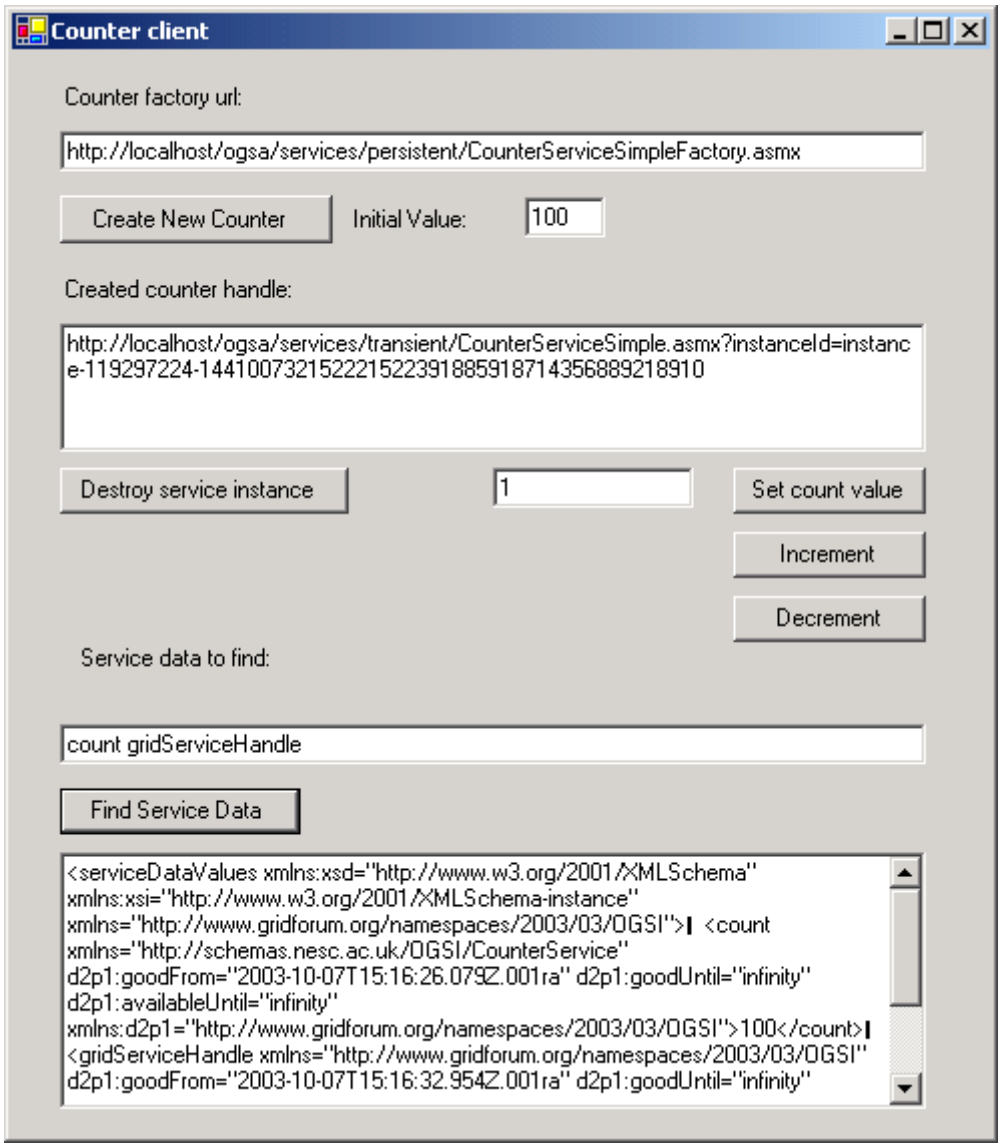
The domain container property needs to be changed to reflect the name of your domain. By default it is set to `localhost`, which is fine for doing development on your local machine. If you wish to expose grid services to other computers on your network, you need to change this value to the name of your computer, e.g. `PC10`. If you are exposing services on a public Web server, you need to enter the name of the Web server here. For example, if you are running the Grid server under www.mydomain.org, enter the value www.mydomain.org here.

You can enable tracing (logging and debugging) for the application by changing the value of the `addTextWriterTraceListener` from false to true. This allows the container to send debug output to a text file. The prefix/directory for the trace files may be specified by the value of the `traceFileRoot` property.

2.2 Running a Demonstration Client

The installation is configured to initialise a counter service factory that can be used in conjunction with a supplied client to test your installation. The client is located in `<install_dir>/MS.NETGrid/DemoServices/CounterService/CounterClient`. Go to the `bin` subdirectory and run the `CounterClient` executable. Click on the "Create New Counter" button. This creates a new instance of a transient counter service. You can set the initial value of this counter service instance upon creation by use of the “Initial Value” text box. The Grid Service Handle for this service instance should appear in the "Created counter handle" box. If it does not, please make sure that IIS is running and the `ogsa` virtual directory is visible from Internet Services Manager.

Several operations may be performed on this service instance. The count value may be set using the "Set count value" button. This sets the current value of the counter to be that of the box to the left of this button. The "Increment" and "Decrement" buttons may be used to adjust the value of the counter by the amount displayed in the box between the buttons. The "Find Service Data" button can be used to obtain service data by name. The `count` service data is found by default. The ServiceData values are displayed in the bottom textbox as pure XML. If this client operates correctly, the installation has been successful. The correctly operating client looks like the following screenshot.


Counter client
Counter factory url:
http://localhost/ogsa/services/persistent/CounterServiceSimpleFactory.asmx
Create New Counter
Initial Value:
100
Created counter handle:
http://localhost/ogsa/services/transient/CounterServiceSimple.asmx?instanceId=instanc
e-119297224-144100732152221522391885918714356889218910
Destroy service instance
1
Set count value
Increment
Decrement
Service data to find:
count gridServiceHandle
Find Service Data
<serviceDataValues xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.gridforum.org/namespaces/2003/03/OGSI"> <count
xmlns="http://schemas.nesc.ac.uk/OGSI/CounterService"
d2p1:goodFrom="2003-10-07T15:16:26.079Z.001ra" d2p1:goodUntil="infinity"
d2p1:availableUntil="infinity"
xmlns:d2p1="http://www.gridforum.org/namespaces/2003/03/OGSI">100</count>
<gridServiceHandle xmlns="http://www.gridforum.org/namespaces/2003/03/OGSI"
d2p1:goodFrom="2003-10-07T15:16:32.954Z.001ra" d2p1:goodUntil="infinity"

3 Class documentation

Source code documentation is available in Microsoft Help format at:

`<install_dir>/MS.NETGrid/doc/API/MS.NETGridHelp.chm`

4 Writing a simple persistent service

In this section we go through the stages required to write and deploy a simple persistent Grid service. You can follow these steps using Visual Studio.NET or Notepad. The required stages are:

1. Write the service skeleton class that acts as the main class for the service.
2. Write the classes that implement the functionality of the portTypes offered by the service.
3. Write the proxy class that exposes the service's public methods.
4. Compile the service classes and proxy class into an assembly.
5. Write a `.asmx` file for the service.
6. Configure the OGSI container.
7. Install the service.

The Grid service we write here will be a persistent Grid service. We will implement a single method called `hello()`, on the `MyService` portType. This method takes one argument called `name` and will return the string "`hello` *`name n`*" where *`name`* is the value of the `name` argument and *`n`* is an integer that starts at 1 and increments every time the method is called.

The Grid service will inherit additional methods (e.g. `findServiceData()` and `destroy()`) that correspond to the GridService portType operations that all Grid services must provide.

Visual Studio.NET

In the case that you are using Visual Studio.NET, the type of project you start should be of type “Class Library”. Make sure that no “hidden” files such as resource or CodeBehind files are created by Visual Studio.NET. If they are, delete them. Each of the files mentioned in the following steps can then be added to the project individually.

4.1 Write the service skeleton

The service skeleton is a class from which the functionality of the Grid service is “hung”. In general, no functionality is directly implemented on this class. Rather, it acts as a reference class for deploying your services, and does behind the scenes lifetime and serviceData management. The code for this class is as follows and should be saved in a file called `MyServiceImpl.cs`:

```
using Ogsi.Core;
namespace MyGridServices
{
  [OgsiPortType( typeof( MyGridServices.MyServicePortType ),
   "http://mydomain.com/myNameSpace",
   "MyGridService" ) ]
  public class MyServiceImpl : PersistentGridServiceSkeleton
  {

  }
}
```

This class inherits from the `Ogsi.Core.PersistentGridServiceSkeleton` class. All **persistent** Grid services must inherit from this class. In contrast, **transient** Grid services must inherit from the `Ogsi.Core.GridServiceSkeleton` class.

This service class does not contain any business logic. It does have a single `OgsiPortType` attribute. This attribute declares that the service exposes a portType called "MyGridService" whose default namespace is "http://mydomain.com/myNameSpace". The .NET type that implements this portType is "`MyGridServices.MyServicePortType`". The implementation of the portType class is discussed in the next section. More than one portType implementation can be associated with the service class by use of multiple `OgsiPortType` attribute. This attribute-based model encourages modularity and reuse of portType implementations.

As can be seen from the code, implementing the service class for a Grid service is very similar to writing any other class. The `Ogsi.Core.PersistentGridServiceSkeleton` and `Ogsi.Core.GridServiceSkeleton` classes give access to other Grid service infrastructure such as service data that the service may wish to use. This simple example does not use service data.

4.2 Write the portType class(es)

Next you must write the service logic, which normally resides on the portType classes. Create a file called `MyServicePortType.cs`. Its contents should be as follows:

```
using Ogsi.Core;
namespace MyGridServices
{
  public class MyServicePortType : PortTypeBase
  {
    private int count_ = 0;

    public string hello()
    {
      return "Hello, " + count_++;
    }
  }
}
```

4.3 Write the proxy class

The proxy class is used to expose the service's public methods, or, in other words, the operations of the Grid service.

The code for the proxy class – which should be saved in a file called `MyService.cs` – is as follows:

```
using System.Web.Services;
using Ogsi.Container.Instances.Proxy;

namespace MyGridServices
{
  public class MyService :
                      PersistentGridServiceInstanceAspProxy
  {
    [WebMethod]
    public string hello( string name )
    {
      object [] args = { name };
      return (string) CallMethodOnPortType(
                     "MyGridServices.MyServicePortType",
                     "hello",
                      args );
    }
  }
}
```

This class inherits from the `PersistentGridServiceInstanceAspProxy` class (in the namespace `Ogsi.Container.Instances.Proxy`). The `System.Web.Services.WebMethodAttribute` is used to tell ASP.NET to expose this method as a public Web method of the service. Other ASP.NET method attributes may be used to provide further customisation. See the ASP.NET user documentation for details.

Every public method on any of the portType classes that is to be exposed as part of the Grid service must have a corresponding method defined for it in the proxy class. The methods in the proxy class are all very similar. They simply put the arguments into an array and then pass these arguments and the method name to the `CallMethodOnPortType()` method that is a public method of the base class. This method essentially farms out the work requested by clients of the service to instances of the portType classes that are maintained by the container.

4.4 Compile the service classes and proxy class into an assembly

To compile the service class and proxy class into an assembly use the following command line instruction (which should be typed on a single line):

```
csc /t:library /out:MyServices.dll MyServiceImpl.cs
MyServicePortType.cs MyService.cs
    /r:<OGSIContainer>\bin\Ogsi.Container.dll
/r:<installDir>/MS.NETGrid/Ogsi.Common/bin/Ogsi.Common.dll
/r:System.Web.Services.dll
```

<OGSIContainer> is the path to the OGSI container installation. We need to reference this dll as it contains the base classes for our types.

If you are using the Microsoft Visual Studio development environment configure your project to output an assembly called `MyServices.dll`. You will have to add the following references to the project before building:

- *<OGSIContainer>*\bin\Ogsi.Container.dll

- *<installDir>*\MS.NETGrid\Ogsi.Common\bin\Ogsi.Common.dll
- System.Web.Services.dll

Regardless of whether you have used Visual Studio or the command line you should now have created an assembly called MyServices.dll, located in the current directory.

4.5 Write a .asmx file for the service

To utilise ASP.NET to deploy our Grid service we must create a .asmx file for the service. This simply maps the service name to the proxy class.

The .asmx file is (save this in a file called MyService.asmx):

```
<%@ WebService Class="MyGridServices.MyService"%>
```

The name of the .asmx file determines the Grid Service Handle (GSH) for the service. This file can be created by hand with Notepad.

4.6 Configure the OGSI container

For each Grid service hosted by the OGSI container an entry must be added to the container's Web.config file (this file is located at *<OGSIContainer>*\Web.config).

Inside the <gridServiceDeployment> element of the file add the following entry:

```
<gridServiceDeploymentDescriptor
            asmxProxyFileName="MyService.asmx"
            serviceClass="MyGridServices.MyServiceImpl"
            assembly="MyServices"
            persistence="persistent"
            >
</gridServiceDeploymentDescriptor>
```

This entry specifies:

- The name of the .asmx file generated in section 4.5, for example MyService.asmx.
- The full name of the service class created in section 4.1, for example MyGridServices.MyServiceImpl.
- The name of the assembly created in section 4.4, for example MyServices.
- Whether the service is persistent or transient.

4.7 Installing the service

To install the service:

- Copy the MyServices.dll assembly to *<OGSIContainer>*\bin.
- Copy the MyService.asmx file to *<OGSIContainer>*\services\persistent.
- Restart IIS.

- To restart IIS obtain the Control Panel, select Administrative Tools and then select Internet Services Manager. Select the appropriate web site (probably Default Web Site) and use the stop and start icons to stop and restart the server.

5 Writing a client of a persistent service

Writing a client to talk to a persistent Grid Service is identical to writing a client to talk to a Web Service. Firstly we must create a **client-side** proxy class (do not confuse this with the server-side Grid Service proxy) for the service and then use it in our own code.

If you are developing clients with Visual Studio.NET, you can use the "Add Web Reference" facility to automatically generate proxy classes for your use. You would direct Visual Studio.NET to add the reference from, for example, "http://localhost/ogsa/services/persistent/MyService.asmx".

To create a proxy class create a new directory and within that directory use the following command line instruction:

```
WSDL http://localhost/ogsa/services/persistent/MyService.asmx
```

If you are accessing our container from a remote host then you can replace localhost with the name of the host upon which our container is running.

This will create a proxy class called `MyService` in a file called `MyService.cs`. Note that this proxy class will be in the default namespace. `WSDL.exe` contains options for creating the class in a namespace of your choice; execute `WSDL` without any command line parameters for details. We can now write a client that uses the service. Code for an example client of our service is:

```
using System;

class MyClient
{
   static void Main()
   {
      MyService GridService = new MyService();
      Console.WriteLine( GridService.hello( "Horace" ) );
   }
}
```

Compile this file along with the `MyService.cs` file as shown in the following command line instruction:

```
csc /out:MyClient.exe MyClient.cs MyService.cs
```

This will produce an executable called `MyClient.exe` that will call the Grid service's `hello()` method when it is run producing the output shown below (identical results will be produced if you use Visual Studio.NET):

```
C:\Projects\ogsanet\client>MyClient
hello Horace 1

C:\Projects\ogsanet\client>MyClient
hello Horace 2

C:\Projects\ogsanet\client>MyClient
hello Horace 3
```

6 Writing a transient service and a persistent factory service

In this section we go through the stages required to write and deploy a simple transient Grid service with a corresponding persistent factory service. The factory service can be used to create instances of the transient Grid service. The stages are:

1. Write the transient service class that implements the functionality of the transient service, and the associated portType classes.
2. Write the factory service class.
3. Write the factory initialiser class that is used by the factory to initialise new service instances.
4. Write proxy classes for the service class and factory service class.
5. Compile classes into an assembly.
6. Write `.asmx` files for the transient service and factory service.
7. Configure the OGSI container.
8. Install the services.

The transient Grid service we write here is very simple. It is initialised with a single string argument called `greeting` provided to the factory service – via the Factory portType `CreateService` operation – that will create our transient Grid service and we will implement a single method called `hello()`. The method takes one argument called `name` and will return the string "*`greeting name n`*" where *`greeting`* is the value of the `greeting` argument the service was initialised with, *`name`* is the value of the name argument of the `hello()` method and *`n`* is an integer that starts at 1 and increments every time the method is called. The Grid service will inherit additional methods (e.g. `findServiceData()` and `destroy()`) that correspond to the GridService portType operations that all Grid services must provide.

6.1 Write the transient service class

The service class is the class that implements the specific functionality of the Grid service. The code for this class – which should be saved in a file called `MyTransServiceImpl.cs` – is as follows:

```
using Ogsi.Core;

namespace MyGridServices
{
  [OgsiPortType( typeof( MyGridServices.GreetingPortType ),
    "http://mydomain/myNameSpace",
    "GreetingService")]
  public class MyTransServiceImpl : GridServiceSkeleton
  {
  }
}
```

This class inherits from the `Ogsi.Core.GridServiceSkeleton` class. All transient Grid

services must inherit from this class.

The next step is to write the `GreetingPortType` class. This class looks like the following:

```
using Ogsi.Core;
namespace MyGridServices
{
  public class GreetingPortType : PortTypeBase
  {
    private int    count_     = 1;
    private string greeting_;

    public string Greeting
    {
      set
      {
        greeting_ = value;
      }
    }
    public string hello( string name )
    {
      return greeting_ + " " + name + " " +
            (count_++).ToString();
    }
  }
}
```

6.2 Write the persistent factory service class

The factory service class is the class that implements the functionality of the persistent factory service. The code – which should be saved in a file called `MyTransServiceFactoryImpl.cs` – is as follows:

```
using System;
using Ogsi.Core;

namespace MyGridServices
{
  [OgsiPortType(
      typeof(FactoryPortType),
      "http://localhost/MyTransServiceFactoryNS",
      "MyTransServiceFactoryPortType") ]
  public class MyTransServiceFactoryImpl :
                                PersistentGridServiceSkeleton
  {
  }
}
```

As can be seen this is a very simple class. The class is for a persistent service so it inherits from `Ogsi.Core.PersistentGridServiceSkeleton`. The service is to be a factory so the we have used the `OgsiPortType` attribute to specify that the service exposes the Factory portType. A generic, configurable factory portType implementation is provided with the MS.NETGrid-OGSI container. When we use the `OgsiPortType` attribute to specify that a

service supports the Factory portType the majority of the implementation is done for us – all we have to do is write an initialiser class to create instances of the service the factory creates and then specify the initialiser class is associated with the factory. These tasks are covered in sections 6.3 and 6.7 respectively.

The second and third arguments to the `OgsiPortType` attribute specify the namespace and name of the portType. These values are not used in the current release but are intended to provide a reference to the portType for use in future releases.

6.3 Write factory initialiser class

The factory initialiser class is used to initialise instances of transient services after they have been created by a factory. Due to the nature of the OGSI implementation we provide, it is not possible to pass information to the portType classes via a constructor. We use initialiser classes then to perform any custom initialiser role. The code – which should be saved in a file called `MyTransServiceInitialiser.cs` – is:

```
using System;
using System.Xml;
using Ogsi.Core;

namespace MyGridServices
{
  public class MyTransServiceInitialiser :
ITransientServiceInitialiser
  {
    public void InitialiseServiceFromParameters(
      GridServiceSkeleton instance,
      XmlElement creationParams )
    {
      GreetingPortType gpt =
instance.PortTypeProviders["MyGridServices.GreetingPortType"]
as GreetingPortType;
      if( gpt != null )
        gpt.Greeting = "Hello there, "; // or whatever
    }

  }
}
```

The factory creator class must implement the `ITransientServiceInitialiser` interface. This interface has only one method called `InitialiseServiceFromParams()`. Essentially what this method does is initialise the service object that has been instantiated by the factory.

The `creationParams` parameter of the `CreateService()` method can be used to pass an arbitrary XML element from the client to the factory. For examples of how to specify the XML type of this element, and how to use the information contained within it, please see the Counter Service example in *<installDir>/MS.NETGrid/DemoServices/CounterService*. Files of interest in this directory include `CounterServiceSimpleImpl.cs`, `CounterServiceSimpleInitialiser.cs` and `CounterPortType.cs`. It is also possible to restrict the type of the XML input element for the Factory portType to be of one of a number of certain types, and a demonstration of this is given in the `PostCreate()`

method of `CounterServiceSimpleFactoryImpl.cs`.

6.4 Write proxy classes

We have to write proxy classes for both the transient service and the factory service.

The code for the service class's proxy class – which should be saved in a file called `MyTransService.cs` – is:

```
using System.Web.Services;
using Ogsi.Container.Instances.Proxy;

namespace MyGridServices
{
  public class MyTransService :
                        TransientGridServiceInstanceAspProxy
  {
     [WebMethod]
     public string hello( string name )
     {
        object [] args = { name };
        return (string) CallMethodOnPortType(
          "MyGridServices.GreetingPortType",
          "hello", args);
     }
  }
}
```

This class is virtually identical to the proxy class for the persistent service discussed in section 4.3. The only difference here is that this class inherits from `TransientGridServiceInstanceAspProxy` because it is a transient rather than persistent service.

The code for the factory class's proxy class – which should be saved in a file called `MyTransServiceFactory.cs` – is:

```
using System.Web.Services;
using Ogsi.Container.Instances.Proxy;
using Ogsi.Core.Types;

namespace MyGridServices
{
  public class MyTransServiceFactory :
                      PersistentGridServiceInstanceAspProxy
  {

    [WebMethod]
    public LocatorType createService(CreationType creation)
    {
      FaultType fault = null;
      object [] arguments = { creation, fault };
      LocatorType retval = (LocatorType) CallMethodOnPortType(
        "Ogsi.Core.FactoryPortType",
        "CreateService",
        arguments );
```

```
        CheckFault( fault );
        return retval;
      }
    }
}
```

In this case, `CallMethodOnPortType` is used as the `CreateService` method is not defined directly on `MyTransServiceFactoryImpl`, but on the `FactoryPortType` provider object associated with `MyTransServiceFactoryImpl` using `OgsiPortTypeAttribute`. The first argument to `CallMethodOnPortType` is the name of the object type and is used to select the appropriate provider. The second argument is the name of the method to call and the third argument is the parameters to pass to the method. Note that one of the parameters of the implementation of `CreateService` on `FactoryPortType` is an `Ogsi.Core.Types.FaultType` object. This is used to pass any fault information back to the proxy client. The `CheckFault`() method is used to see if the fault is not null and throws an exception in a standard manner if a fault has occurred. This is a standard paradigm that you should use for operations that may return a fault.

6.5 Compile classes into an assembly

To compile all the classes into an assembly use the following command line (this should be typed on a single line):

```
csc /t:library /out:MyTransServices.dll
 MyTransService.cs MyTransServiceFactory.cs
 MyTransServiceInitialiser.cs MyTransServiceFactoryImpl.cs
 MyTransServiceImpl.cs GreetingPortType.cs
 /r:<OgsiContainer>\bin\Ogsi.Container.dll
 /r:<installDir>/MS.NETGrid/Ogsi.Common/bin/Ogsi.Common.dll
 /r:System.Web.Services.dll
```

6.6 Write `.asmx` files for the transient service and factory service

To utilise ASP.NET to deploy our Grid service we must create a `.asmx` file for the transient service and the factory service. As in the case of the persistent service this simply maps the service name to the proxy class.

The `.asmx` file for the transient service is (save in file `MyTransService.asmx`):

```
<%@ WebService Class="MyGridServices.MyTransService"%>
```

The `.asmx` file for the factory service is (save in file `MyTransServiceFactory.asmx`):

```
<%@ WebService Class="MyGridServices.MyTransServiceFactory"%>
```

6.7 Configure the OGSI container

For each Grid service hosted by the OGSI container an entry must be added to the container's `Web.config` file (this file is located at *<OGSIContainer>*`\Web.config`).

For the transient service the following entry must be added to the `<gridServiceDeployment>` element of the `Web.config` file:

```
<gridServiceDeploymentDescriptor
asmxProxyFileName="MyTransService.asmx"
      serviceClass="MyGridServices.MyTransServiceImpl"
```

```
      assembly="MyTransServices"
      persistence="transient"
      >
</gridServiceDeploymentDescriptor>
```

This contents of this element are similar to that shown in section 4.6 for the simple persistent service. Notice the value of the "persistence" attribute is now "transient".

For the factory service the following element must be added to the `<gridServiceDeployment>` element of the `Web.config` file:

```
<gridServiceDeploymentDescriptor
      asmxProxyFileName="MyTransServiceFactory.asmx"
      serviceClass="MyGridServices.MyTransServiceFactoryImpl"
      assembly="MyTransServices"
      persistence="persistent"
      >
  <!-- The type of the service object to create -->
  <serviceParameter
           name="creationType"
          value="MyGridServices.MyTransServiceImpl"/>
  <!-- The type of the initialiser, with its assembly -->
  <serviceParameter
         name="initialiserType"
         value="MyGridServices.MyTransServiceInitialiser,
                MyTransServices"/>
</gridServiceDeploymentDescriptor>
```

The majority of this element is very similar to that for the transient service. The `<serviceParameter>` elements are new. ServiceParameter elements provide a way of passing services customisable information when they are created. This element specifies a parameter that is passed to the service. In this case one parameter name is "`creationType`". The parameter value is the full name of the type created by this factory service. The second serviceParameter, called `initialiserType`, has as its value the name of the initialiser that the factory service uses to initialise services after they are created, along with its assembly name.

The final piece of configuration that must be carried out is to set the domain name that is used when creating transient services. This is simply the domain name of the machine running IIS. Edit the following line in the `Web.config` file (found in the `configuration/gridContainer.config/containerProperties` element) and set the value field to the appropriate domain name (e.g. "`www.epcc.ed.ac.uk`"). This domain information is used when generating URLs and handles for grid services.

```
            <add key="domain" value="localhost"/>
```

6.8 Install the services

To install the service:

- Copy the `MyTransServices.dll` assembly to *`<OGSIContainer>`*`\bin`.
- Copy the `MyTransService.asmx` file to *`<OGSIContainer>`*`\services\transient`

- Copy the `MyTransServiceFactory.asmx` file to *<OGSIContainer>*`\services\persistent`.
- Restart IIS

7 Writing a client of a transient service

Writing a client to talk to a transient Grid Service is virtually identical to writing a client to talk to a Web Service. Firstly we must create a proxy classes for the services and then use them in our own code.

To create proxy classes create a new directory and within that directory issue the following command line commands (each should be entered on a single line):

```
WSDL /namespace:MyTransServiceNS
 http://localhost/Ogsi.Container/services/transient/MyTransService.asmx
```

```
WSDL /namespace:MyTransServiceFactoryNS
 http://localhost/Ogsi.Container/services/persistent/MyTransServiceFactory.asmx
```

This will create proxy classes called `MyTransService.cs` and `MyTransServiceFactory.cs` each within their own namespace to avoid name clashes with common types.

We can now write a client that uses the factory and the service instances it creates. The code for the client is (save this in a file called `MyTransClient.cs`):

```
using System;
using System.Xml;

using MyTransServiceNS;
using MyTransServiceFactoryNS;

class MyTransClient
{

   static void Main()
   {
      // Create a factory service object
      MyTransServiceFactory factoryService =
                              new MyTransServiceFactory();

      // Construct XML creation element this is used to create
      // the service instance
      XmlDocument doc = new XmlDocument();
      XmlElement xmlParams = doc.CreateElement("initialValue");
      XmlNode    val       = doc.CreateTextNode("Ciao");
      xmlParams.AppendChild(val);

      // Construct a creation object
      MyTransServiceFactoryNS.CreationType creation;
      creation = new CreationType();
      creation.terminationTimeSpecified = false;
      creation.serviceParameters = xmlParams;

      // Create a service using the factory
      MyTransServiceFactoryNS.LocatorType locator;
      locator = factoryService.createService( creation );

      Console.WriteLine( "Have a service the locator is:\n"
        + locator.handle[0] );
```

```
      // Create service object
      MyTransService service = new MyTransService();

      // Tell this service object the location of the service
      // instance
      service.Url = locator.handle[0];

      // Use the service object
      Console.WriteLine( service.hello( "Mike"   ) );
      Console.WriteLine( service.hello( "Daragh" ) );
      Console.WriteLine( service.hello( "Ally"   ) );

      // Destroy the transient service when finished
      Service.destroy();
   }
}
```

There are several points to note about the code:

- We construct an XML element called `xmlParams` that contains the data required by the factory creator class. The schema of this element is specific to this particular factory, other factories can use different schema for their creation service parameters.
- The creation parameters XML element is bundled inside a `CreationType` object that is passed as input to the factory's `createService()` method.
- The `Url` property of all transient services proxies must be set before any methods of the proxy are called. The `Url` property must be set to the handle returned by the factory via the `LocatorType` object.
- The transient service can be destroyed when the client is finished with it by calling the `destroy()` method.

Compile this file along with `MyTransService.cs` and `MyTransServiceFactory.cs` as shown in the following command line:

```
csc /out:MyTransClient.exe MyTransClient.cs MyTransService.cs MyTransServiceFactory.cs
```

This will produce an executable called `MyTransClient.exe` that will use the factory to create a new instance of the transient Grid service and then call that service's `hello()` method several times to producing the output shown below:

```
Have a service the locator is:
http://localhost/Ogsi.Container/services/transient/MyTransService.asmx?instanceId=instance-
141776660--214136689-1040913524821511526894621964229651231731 37
Ciao Mike 1
Ciao Daragh 2
Ciao Ally 3
```

8 Service data

The concept of service data is an important part of OGSI. The `GridServiceSkeleton` base class provides a collection for service data via the `InstanceServiceData` property. Through this collection the service developer can manipulate the service data in many ways. This section describes how the developer can manipulate the service data.

8.1 Default service data elements

All services will have several service data elements by default. These are defined by the OGSI specification and are implemented in the `GridServiceSkeleton` base class. The default service data elements are:

- interface
- factoryLocator
- gridServiceHandle
- gridServiceReference
- findServiceDataExtensibility
- setServiceDataExtensibility
- terminationTime
- serviceDataName

8.2 Service data set

The `GridServiceSkeleton` base class provides a protected member variable through which service implementations can access the service data collection. This member variable is called `serviceDataSet_` and is of type `ServiceDataSet`. The service data set is a container for service data elements. Each service data element is referenced by an XML qualified name (consisting of a local name and a namespace). The `ServiceDataSet` class provides methods to create new service data elements, add service data elements to the collection, access service data elements from the collection as well as several others. See the class documentation for full details of the class.

New service data elements can be added to the collection with code similar to the following:

```
// Construct a qualified name for the service data element
System.Xml.XmlQualifiedName sdeName =
  new System.Xml.XmlQualifiedName( "mySdeName",
                                   "MySdeNamespace" );

// Create a new service data element
ServiceData sde;
sde = serviceDataSet_.Create( sdeName );

// . . . code to populate the ServiceData object goes here
```

```
// Add service data element to the service data set
serviceDataSet_.Add( sde );
```

The indexer property of the `ServiceDataSet` can be used to access the `ServiceData` objects by the service data element's qualified name as show here:

```
// Construct the qualified name for the service data element
System.Xml.QualifiedName terminationTimeQName;
terminationTimeQName =
  new System.Xml.QualifiedName("terminationTime"
                               CoreConstants.OGSI_NAMESPACE_URI);

// Obtain the ServiceData object using the indexer
ServiceData sd = serviceDataSet_[ terminationTimeQName ];

// can now use sd to access and alter the termination time
// service data element
```

8.3 The `ServiceData` class

The `ServiceData` object is used to store the properties and values of the service data elements. For full details of the `ServiceData` class see the class documentation – this section provides an introduction to some of the mean features of the class.

8.3.1 Service data element properties

There are a number of standard OGSI properties that are associated with a service data element. These are listed below along with their default values and corresponding property of the `ServiceData` class.

OGSI property	Default value	`ServiceData` property
Minimum occurs	0	`minOccurs`
Maximum occurs	1	`maxOccurs`
Mutability	static	`mutability`
Modifiable	false	`modifiable`
Nillable	false	`nillable`

8.3.2 Service data values

Service data elements can have zero or more values. It is important to initialise a `ServiceData` object with details of the values that can be stored within it. For example, the following code configures a `ServiceData` object to store values of type `MyClass`:

```
// Assume there is a ServiceData object called sd

// Specify that the ServiceData objects will store values of
// type MyClass
sd.ValuesType = typeof(MyClass);
```

Values can be added to the `ServiceData` using the `AddValue()` method. For example:

```
MyClass myObject = new MyClass();
sd.AddValue( myObject );
```

There are several other methods and properties defined on the `ServiceData` class to manipulate the values of a service data element. See the class documentation for more details.

8.3.3 Using a callback to determine service data values dynamically

In some instances it is preferably to calculate the service data values dynamically as and when they are needed rather than storing a collection of values in the `ServiceData` object. This can be easily done by providing a callback class that the `ServiceData` class will use to determine the service data values whenever they are requested.

The callback class must implement the `IServiceDataValuesCallback` interface. This interface specifies a single read-only property called `ServiceDataValues` that returns an array of objects that are the service data values. For example:

```
class MyCallbackClass : IServiceDataValuesCallback
{
  public object[] ServiceDataValues
  {
    get
    {
       // code to create an object array of values goes here
       return result;
    }
  }
}
```

To specify that the callback mechanism should be used to determine the service data values the `Callback` property of the `ServiceData` class must be set to the instance of the callback class. For example:

```
MyCallbackClass callbackInstance = new MyCallbackClass();
sd.Callback =  callbackInstance;
```

It is important to note that this mechanism for dynamic service data value creation can only be used when the service data element is non-modifiable.

8.3.4 Lifetime attributes

Service data values can have lifetime attributes associated with them. To add a service data value with specific lifetime values to a `ServiceData` object the overloaded `AddValue()` method can be used. There is a version of the method that takes lifetime values as well as the value object itself. For example:

```
// Create value object
MyClass myObject = new MyClass();

// Create lifetime objects
System.DateTime goodFrom = System.DateTime.Now;
System.DateTime goodUntil = Ogsi.Core.Util.GridDateTime.INFINITY;
System.DateTime availableUntil =
                              Ogsi.Core.Util.GridDateTime.INFINITY;
```

```
sd.AddValue( myObject,
             goodFrom,
             goodUntil,
             availableUntil );
```

If no lifetime values are specified the `goodFrom` attribute defaults to the current time and the `goodUntil` and `availableUntil` attributes default to `INFINITY`.

It is possible to obtain the lifetime attributes associated with the values of a `ServiceData` object using the `GetLifetimes()` method. This method returns an array of `ServiceDataAttributes` objects each corresponding to a service data value. The `ServiceDataAttributes` class has properties to get and set the lifetime attributes and also get and set the value to which the lifetime attributes are associated.

See the class documentation for classes `ServiceData` and `ServiceDataAttributes` for more details.

8.3.5 Service data evaluators

The grid service `findServiceData` and `setServiceData` operations are implemented by the `GridServiceSkelton` base class. By default the `findServiceData` operation supports the `queryByServiceDataNames` query expression and the setServiceData operation supports the setByServiceDataNames and deleteByServiceDataNames query expressions.

It is possible to add new query expressions to these operations. To add support for a new query expression you must implement a new query evaluator to evaluate the query. Query evaluators must implement the `IExpressionEvaluator` interface. The evaluator must then be added to the query engine. The `GridServiceSkeleton` base class has a protected member variable `queryEngine_` of type `Ogsi.ServiceData.Query.QueryEngine`. This class has a method `RegisterEvaluator()` that can be used to add a query evaluator to the query engine. For examples of query evaluators see the implementations of `QueryByServiceDataNamesEvaluator` and `SetByServiceDataNamesEvaluator` both in the `Ogsi.ServiceData.Query` namespace.

9 PortTypes

A large portion of [OGSI-Spec] addresses the issue of services providing multiple portTypes, and allowing services to inherit from multiple portTypes. In our approach, the `WebMethodAttribute`-marked methods in the proxy classes essentially represent the most-derived portType for a service, and it is with this that the clients communicate. At present, it is only possible to present this most-derived portType to the client. It is possible to alter the properties and namespaces of the exposed operations by using the `WebServiceAttribute`, `SoapDocumentMethodAttribute` and `SoapRpcMethodAttribute` classes, and others, provided by the .NET framework. See the .NET user documentation for details.

We address the issue of portType re-use in the following manner. When developing a service implementation, service methods may be placed on the class you derive from `GridServiceSkeleton`, like so:

```
public class MyService : PersistentGridServiceSkeleton
{
      public string myMethod()
      {
            this.InstanceServiceDataSet[someQName] = 10;
            return "hello";
      }
}
```

With this method, access to the service's `ServiceDataSet` is via the `InstanceServiceData` property inherited from `GridServiceSkeleton`. Adding methods in this manner essentially means adding methods to a base service portType.

In other circumstances, it may be desirable to use functionality developed elsewhere to provide operations, or to logically group related functionality into single portTypes, which you then wish to aggregate on your service. We provide means of doing this using the `Ogsi.Core.OgsiPortTypeAttribute` and `Ogsi.Core.OperationProviderBase` classes and the `Ogsi.Core.IOperationProvider` interface.

9.1 OgsiPortTypeAttribute

We use as an example a class `SomeUsefulClass`, with a method `SomeUsefulMethod`. It is desired to expose this method as an operation on your service. The simplest way to do this is as follows:

```
[OgsiPortType( typeof( MyAssembly.SomeUsefulClass ),
               "http://mydomain.com/myNameSpace",
               "UsefulPortType" );
public class MyService : PersistentGridServiceSkeleton
{
}
```

The first parameter in the attribute is the type of your provider class. The second parameter is the URI of the namespace for this new portType. The third parameter is the name of the portType. The latter two parameters do not have an effect at the moment but are included for possible future work.

When the `MyService` class is instantiated by the container, instances of every class defined by its `OgsiPortTypeAttributes` are created and associated with the service instance.

When developing proxy methods for provider-based methods, the `CallMethodOnPortType` method of `GridServiceInstanceAspProxy` is used:

```
[WebMethod]
public int SomeOtherUsefulMethod()
{
  return CallMethodOnPortType(
    "MyAssembly.SomeUsefulClass",
    "SomeUsefulMethod",
    (object[]) null
  );
}
```

The provider classes can be given access to the `ServiceDataSet` of the service in the following manner. If the provider classes are found to implement `IPortTypeProvider`, the `ServiceInstance` property defined by this interface is initialised with a reference to the instance of `MyService` that it is associated during initialisation.

We provide a basic implementation of `IPortTypeProvider` called `Ogsi.Core.PortTypeBase` for convenience.

The provider class may now access the service's `ServiceDataSet` by obtaining a reference from the reference it maintains to the service class:

```
public class SomeOtherUsefulClass : OperationProviderBase
{
  public int SomeOtherUsefulMethod()
  {
    int i;
    // process
    ...
    this.ServiceInstance.InstanceServiceData[someQName] = i;
    return i;
  }
}
```

10 Example services

As an example of using portTypes, serviceData and persistent and transient services, we provide a pair of demonstration services. These are located in the `<install_dir>/MS.NETGrid/DemoServices` directory.

10.1 Basic Grid Service

The `basicService` subdirectory contains a persistent Grid Service that implements the GridService portType, a transient version of this service and a factory for creating instances of the transient service.The code for the service implementation classes, the proxy classes and the .asmx file are all found in this directory. These services do not expose any additional functionality, but do demonstrate the programming model that this document describes.

10.2 Counter Service

The counter subdirectory contains a more interesting example. The transient service, located in `CounterServiceSimple.cs`, exposes all the functionality of the GridService portType, inherited from `GridServiceSkeleton`. In addition, Increment and Decrement methods are provided. A serviceData element called count is defined with its own namespace. The `Increment` and `Decrement` methods are used to alter the value of this serviceData element. In addition, the value of count may be set using the `setServiceData` operation provided by the GridService portType.

The deployment for the transient counter service and the persistent counter factory service may be examined in `<install_dir>/MS.NETGrid/Ogsi.Container/Web.config`.

The counter factory demonstrates the use of previously existing portTypes via the `OgsiPortTypeAttribute` attribute. It also demonstrates the use of state in Grid services via serviceData. The `CounterFactoryCreator` class demonstrates the use of the `IFactoryProvider` interface. The `CreateServiceObject`() method takes an XML element with that specifies the initial value of the counter.

10.3 OGSA-DAI Demonstrator Service

For more information about the architecture of OGSA-DAI, please see http://www.ogsadai.org.uk. OGSA-DAI is an attempt to establish standard mechanisms for accessing and manipulating data, both structured and unstructured, using Grid services.

This section contains instructions on how to install the OGSA-DAI Grid Data Service within the MS.NETGrid OGSI container. At the time of release, the service is designed to work with, and has been tested using, a standard installation of SQL server.

The relevant source and binary modules are found in `<installDir>\MS.NETGrid\DemoServices\OgsaDaiDemo`. There should be four directories in this directory, namely:

```
Client
Service
Utils
Docs
```

The code for the Grid Data Service is contained in the `Service` directory.

To install the Service, carry out the following steps:

1. Copy the file `OgsaDaiDemo\Service\bin\GsdDemoService.dll` to the directory *<OGSIContainer>*`\bin`.

2. The service uses a utility library that is located at `OgsaDaiDemo\Utils\bin\GdsUtils.dll`. Copy this file to *<OGSIContainer>*`\bin`, i.e. the same directory as above.

3. Copy the file `OgsaDaiDemo\Service\GridDataService.asmx` to the folder *<OGSIContainer>*`\services\persistent`.

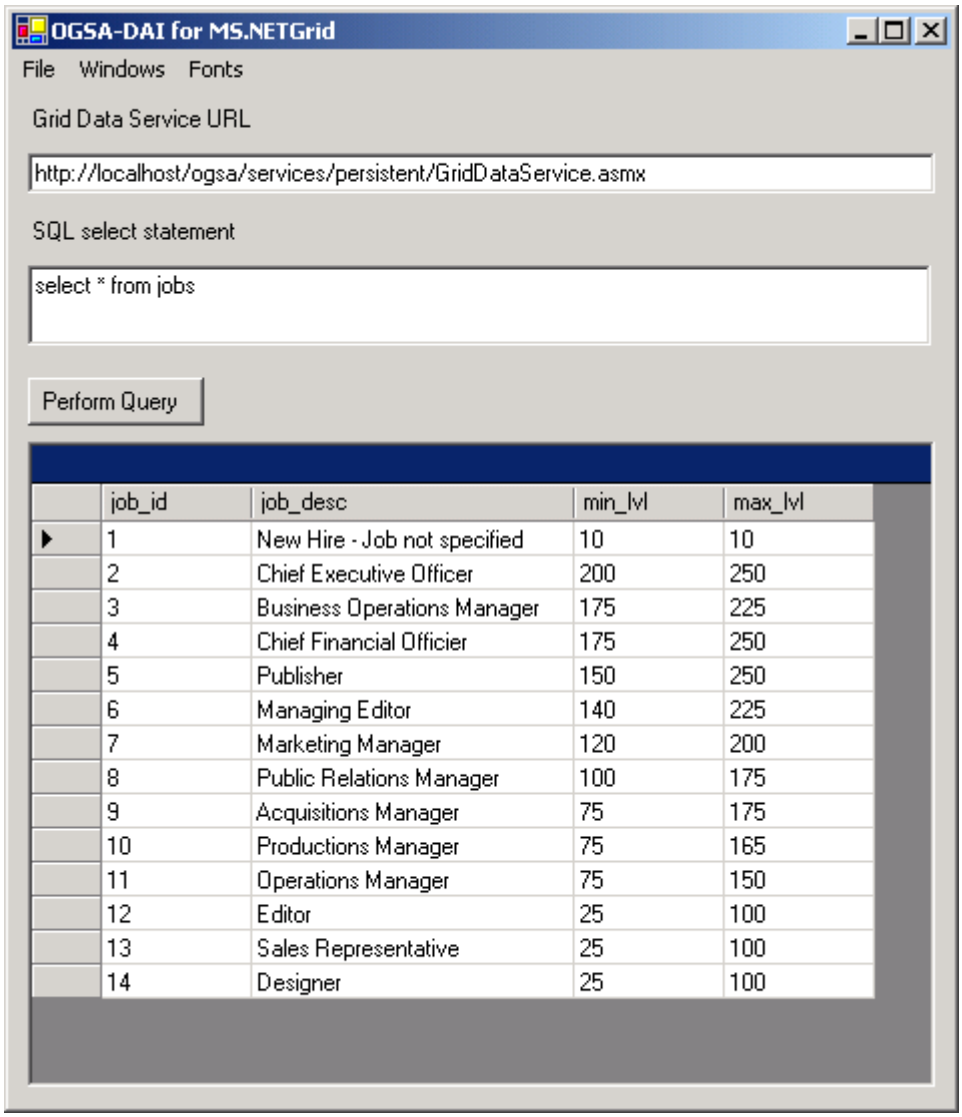
4. Copy the contents of `OgsaDaiDemo\Service\applicationFiles\GridDataService` to *<OGSIContainer>*`\applicationFiles\GridDataService`. You will need to create the `GridDataService` directory in `applicationFiles`.

5. Add a child element to the configuration\gridContainer.config\gridServiceDeployment element. The element should be called gridServiceDeploymentDescriptor and should look like the following:

```
<gridServiceDeploymentDescriptor
  asmxProxyFileName="GridDataService.asmx"
  serviceClass="Ogsi.Gds.Service.GridDataServiceImpl"
  assembly="GdsDemoService"
  persistence="persistent"
>
  <serviceParameter
     name="connectionString"
     value="data source=MACHINE_NAME;initial
catalog=pubs;persist security info=False;user
id=ogsadai;pwd=ogsadai;workstation id=MACHINE_NAME;packet
size=4096" />
  <serviceParameter
     name="performDocumentSchemaLocation"

value="file://C:\MSNETGRID_INSTALL_DIR\MS.NETGrid\Ogsi.Contain
er\applicationFiles\GridDataService\perform_document.xsd" />
 <serviceParameter
     name="sqlQueryStatementActivitySchemaLocation"

value="http://localhost/ogsa/applicationFiles/GridDataService/
sql_query_statement.xsd" />

</gridServiceDeploymentDescriptor>
```

The connection string should be set appropriately to point to the relevant database server and database. The service should now be deployed and running in the container. You can check that this is so by running the client, found in the `bin` directory of the `Client` directory.

Try running arbitrary SQL queries. The results should be displayed in a datagrid on the main form of the client. The correctly functioning Grid Data service should look like the following. If an exception occurs, it is most likely to do with incorrect configuration information.



job_id	job_desc	min_lvl	max_lvl
1	New Hire - Job not specified	10	10
2	Chief Executive Officer	200	250
3	Business Operations Manager	175	225
4	Chief Financial Officier	175	250
5	Publisher	150	250
6	Managing Editor	140	225
7	Marketing Manager	120	200
8	Public Relations Manager	100	175
9	Acquisitions Manager	75	175
10	Productions Manager	75	165
11	Operations Manager	75	150
12	Editor	25	100
13	Sales Representative	25	100
14	Designer	25	100

Glossary

For convenience of reference we include the following glossary.

.NET – The Microsoft .NET platform, including the .NET SDK (Software Development Kit), associated development tools such as Visual Studio .NET [http://www.microsoft.com/net] and the Common Language Runtime.

Common Language Runtime – a runtime environment that executes Microsoft Intermediate Language (MSIL).

MSIL – The .NET bytecode, a low-level assembler-like language executed by the Common Language Runtime.

Managed code – Code that runs on the .NET CLR.

IIS – Microsoft Internet Information Server, a web server offering communication over a variety of internet protocols.

ISAPI – An Application Programming Interface (API) that allows applications to use the network services provided by IIS. Commonly used to provide HTTP filters that respond to all HTTP requests on the server.

AppDomain – .NET allows the partitioning of a single Operating System process into a number of AppDomains, which are essentially memory safe areas within the process. If the code in an AppDomain crashes, other AppDomains within the process are unaffected. This concept has some performance advantages as using AppDomains avoids the overheads associated with starting a new process.

.NET Remoting – The mechanism by which .NET allows Remote Procedure Call (RPC) between AppDomains and between remote machines.

ASP – Active Server Pages, a Microsoft technology that works in conjunction with IIS to host Web Applications.

ASP.NET – The .NET version of ASP, with extensions for Web Services and Enterprise Applications.

C# - Pronounced C-Sharp, a new Microsoft programming language that takes advantage of .NET platform features.

Assembly – A collection of managed code, which may form an executable or a library, and may be distributed across a number of physical files. Assemblies facilitate the separation of logical and physical resources.

OGSA – The Open Grid Services Architecture, which is a set of proposed architectures and protocols for Grid computing [Anatomy].

OGSI – The Open Grid Services Infrastructure, a specification of behaviours and properties of Grid Services [Physiology, OGSI-Spec].

OGSA-DAI – OGSA Data Access and Integration, Grid Services framework for seamless access and integration of data using the OGSA paradigm [OGSA-DAI].

Globus – The Globus toolkit. A standard toolkit of Grid software. Version 3 will contain an implementation of OGSI in Java [http://www.globus.org].

WSDL – The Web Services Description Language [http://www.w3.org/TR/wsdl].

Web Services – Refers to network-enabled software capabilities accessed using common Internet protocols such as HTTP and SOAP and described using WSDL.

Grid Services – Web Services conforming to the OGSI specification.

Grid Service Instance – A network accessible instance of an OGSI-compliant service.

PortType – A set of operations supported by a Web or Grid Service.

ServiceData – An XML-based mechanism that allows a client to query the state of a service instance in a flexible and extensible manner.

ServiceDataElement – A particular element of serviceData, identified by a name and a value

Tomcat – A container for Java-based Web Applications [http://jakarta.apache.org].

AXIS – A Web Application running under Tomcat, which provides Web Services functionality. [http://xml.apache.org].

Programming Model – A set of procedures and APIs used to develop an application in a given domain

References

Documents referenced in the text, and other related documents, include the following.

[MS.NETGridOgsiDesignOverview] MS.NETGrid OGSI Implementation Design Overview (MS.NETGridOgsiDesignOverview-v1.0), *D.Byrne*, EPCC, June 16th 2003.

[MS.NETGrid-Proj-Def] Project Definition for a Collaboration Between Microsoft and EPCC (MS.NetGrid-ProjDef-V2.0), *M. Jackson*, EPCC, April 25th 2003.

[OGSI-Spec] Open Grid Services Infrastructure (Draft 29), *S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, P. Vanderbilt,* April 5th 2003. http://www.gridforum.org/ogsi-wg.

[Virginia-Impl] – OGSI.NET: An OGSI-compliant Hosting Container for the .NET Framework, Grid Computing Group, University of Virginia. WWW site: http://www.cs.virginia.edu/~humphrey/GCG/ogsi.net.html.

[GTk3] – Globus Toolkit version 3 and OGSI, Available at http://www.globus.org/ogsa

[Physiology] – The Physiology of the Grid (Draft 2.9), *I. Foster, C. Kesselman, J.Nick, S. Tuecke,* Available at http://www.gridforum.org.org/ogsa-wg

[Anatomy] – The Anatomy of the Grid, *I. Foster, C. Kesselman, S. Tuecke,* Available at http://www.gridforum.org/ogsa-wg

[OGSA-DAI] – Open Grid Service Architecture, Database Access and Integration http://www.ogsadai.org.uk

[WSDL-Spec] Web Services Description Language 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[AXIS] – The Apache Axis SOAP Engine, http://ws.apache.org/axis/