Building a Scalable, Reliable OGSI Container

Project Title:	MS.NETGrid
Document Title:	*Building a Scalable, Reliable OGSI Container*
Document Identifier:	MS.NetGrid-ScalableOGSIDesign.doc
Distribution Classification:	Commercial in Confidence
Authorship:	Daragh Byrne
Document History:	

Personnel	Date	Summary	Version
DB	22nd February 2004	EPCC Approved	1.0

Approval List: *EPCC: Project Leader, Technical Staff, Technical Reviewers x 2, Coach*

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

Approval List: *Microsoft Research: Managing Director, University Relations x 2*

Andrew Herbert, Fabien Petitcolas, Van Eden, Dan Fay

1 Introduction

The Open Grid Services Infrastructure (OGSI) [OGSI-Spec] represents an attempt to build upon Web Services standards in order to make them meet the requirements found in Grid computing environments. In OGSI, a resource on the Grid is represented conceptually by an "instance" of a "Grid Service". Instances of Grid Services reside in software environments known as "containers". Instances may be activated by clients, or created and managed by a container.

OGSI specifies the following, which are not addressed by standard Web Services specifications:

- Interfaces that allow the dynamic creation and destruction of stateful Grid Service instances by client applications
- Lifetime and expiration behaviours for Grid Service instances – a client may not need indefinite access to a service instance and these are provided to enable the freeing of resources on the server
- A standard means of accessing and manipulating the state associated with a service instance
- A means of notifying interested parties to changes in the state of a Grid Service instance's state
- A mechanism that allows reliable identification and virtualisation of physical location for service instances
- A means of grouping information about services together to create registry and discovery services.

1.1 Reliability

The meaning of reliability in this document is "every service instance resident in a container when the container fails or is shut down should be able to have its state restored on when the container is restarted". Essentially, we are referring to storing, or "persisting", the state of every service instance in the container at suitable times, or at container shutdown, and reloading that state upon container restart.

1.2 Scalability

There are a number of senses in which the terms "scalable" and "scalability" are used in this document. A scalable container should be able to host a large number of Grid Service instances. A scalable container should also be able to respond well to a large number of client requests to a particular service instance. The architecture of a scalable container should allow for "scaling up" – running the container on a single machine of better specification – and "scaling out" – being able to add new machines to a cluster on which the container is running, in the manner of a Web farm. Both of these forms of scalability are common in enterprise and scientific computing.

As we shall see, the stateful nature of the OGSI architecture presents a number of interesting problems in the design of a scalable container.

1.3 Current State of Play of MS.NETGrid-OGSI Software

Our container takes the form of a Microsoft ASP.NET Web application that is designed to run under Microsoft IIS (Internet Information Server). In MS.NETGrid-OGSI version 1.2, services are divided into container-managed ("persistent service instances"), which are activated by the container upon start-up and live, effectively, for the life of the container, and client activated ("transient service instances"), which are activated by client programs using "factory" service instances (services that implement the OGSI Factory portType, usually themselves persistent service instances).

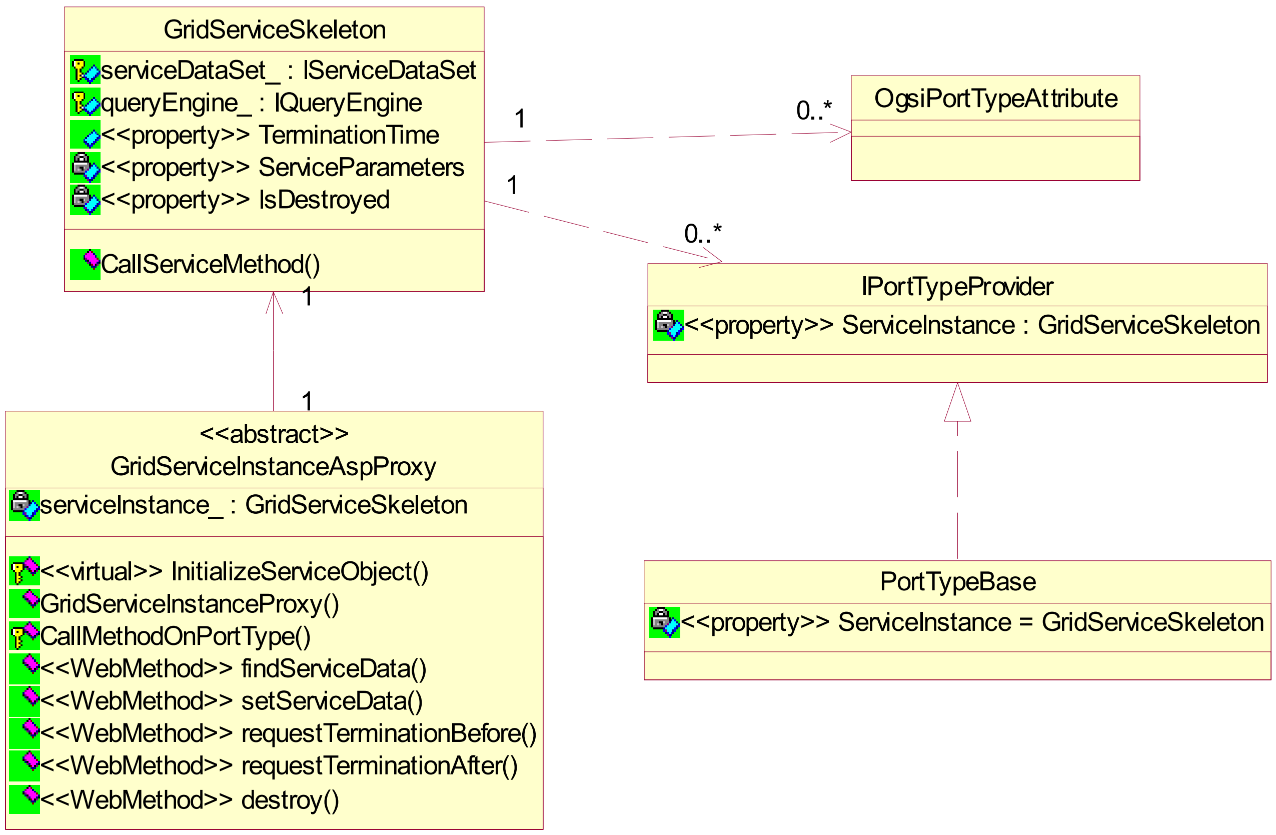
In the MS.NETGrid software a single service instance is represented by a graph of connected objects. The principal object associated with a service instance is an instance of a class that inherits from the `Ogsi.Core.GridServiceSkeleton` class. The following classes are associated with this "skeleton" by composition:

- A collection of classes that provide the functionality of the portTypes the service offered – these are known as portType providers and usually implement the `IPortTypeProvider` interface
- A single instance of `ServiceDataSet`, which is a container for all the service data elements associated with the service
- Other classes provided by the applications programmer.

References to these Grid Service instances are stored in a table. Each service instance has a unique name within the container.

The use of a Web Service proxy allows the skeleton object to be used by clients. The proxy is an extended version of an ASP.NET Web Service that looks up the appropriate skeleton on a per-request basis and passes invocations off to it.

The following diagram illustrates the structure of a Grid Service instance in MS.NETGrid-OGSI:


GridServiceSkeleton
serviceDataSet_ : IServiceDataSet
queryEngine_ : IQueryEngine
<<property>> TerminationTime
<<property>> ServiceParameters
<<property>> IsDestroyed
CallServiceMethod()
1
0..*
OgsiPortTypeAttribute
1
0..*
IPortTypeProvider
<<property>> ServiceInstance : GridServiceSkeleton
1
1
<<abstract>>
GridServiceInstanceAspProxy
serviceInstance_ : GridServiceSkeleton
<<virtual>> InitializeServiceObject()
GridServiceInstanceProxy()
CallMethodOnPortType()
<<WebMethod>> findServiceData()
<<WebMethod>> setServiceData()
<<WebMethod>> requestTerminationBefore()
<<WebMethod>> requestTerminationAfter()
<<WebMethod>> destroy()
PortTypeBase
<<property>> ServiceInstance = GridServiceSkeleton

MS.NETGrid-OGSI uses a configuration file to store information about services. Specifically, the configuration file contains information about which persistent services should be created by the container on start-up, and information relating skeleton types with proxy types and service endpoints.

Further information can be found by reference to the source code, the user manual and the design overview, which are all available with the MS.NETGrid-OGSI software download.

2 Reliability Use Cases

The following use cases describe the functionality that our container should provide with regard to reliability as defined previously.

UC-RS0 – Container-managed service instances should be initialised in the container on initial container startup

When the container is started for the first time, all container-managed services in the configuration should be initialised and stored in suitable persistent media. They should be initialised "from scratch" – i.e. freshly created.

UC-RS1 – Container-managed service instances in the container should be restored after a container crash

UC-RS2 - Container-managed service instances in the container should be restored on container manual restart

When the container is restarted, whether manually or as a result of a fault, the container-managed services that it initially created should be recreated in their last known good state.

UC-RS3 – Factory-created service instances in the container should be restored after a container crash, within their lifetime constraints

UC-RS4 – Factory-created service instances in the container should be restored on manual restart, within their lifetime constraints

All factory-created service instances should be restored in a similar manner to container-managed instances. However, they should be checked upon restart to make sure that their expiration time has not passed by, in which case they should be disposed of in the normal manner.

UC-RS4 – Administrator may wish to reset or permanently delete a container-managed service instance or factory-created service instance from the container

In the case, for example, that a persistent service instance needs to be reset (for example it has spawned an infinite loop or has some other bug) we need a means for a container administrator to kill the service instance and restart it again later.

3 Reliability Outline Design

3.1 Scope

Within the current MS.NETGrid project there is scope to implement UC-RS2 and UC-RS4 above. We outline a design with these use cases in mind.

3.2 Refactoring of Current Deployment Model

In order to correct a design flaw of the current MS.NETGrid software and thus make the reliability code easier to implement, we will carry out some refactoring. The only difference between "persistent" and "transient" services (as understood in the meaning defined by the design overview document) is that persistent services will live as long as the container i.e. a persistent service instance will respond to all operation invocations as long as the container is running. Transient service instances stop responding to operation invocations after their lifetime has expired.

Currently the divide between transient and persistent services is maintained on three levels:

- A different base class is used for persistent and transient services
- A different indexing scheme is used within the container registry of services to name and access persistent and transient services
- A different proxy is used for persistent and transient services.

This divide is quite artificial, and may in fact impede user's understanding of the MS.NETGrid software. Also, different treatment of each service type will add to the workload when developing code for the reliability features mentioned in section 3.1 above that we are to implement. The re-factorings proposed below will minimise the need for code duplication in the reliability features implementation and thus save time spent developing and testing.

The first re-factoring concerns the service deployment model. In the MS.NETGrid software, a service is deployed in the container by writing a Grid Service Deployment Descriptor. This is an XML element that is placed in the `Web.config` file of the MS.NETGrid-OGSI Web application. The deployment descriptor is placed as a child element of the gridContainer.config element, as follows:

```
<gridContainer.config>
   <gridServiceDeploymentDescriptor
      asmxProxyFileName=”MyService.asmx”
      serviceClass="ServiceClass"
      assembly="SomeAssembly"
      persistence="persistent" />
      <serviceParameter name=”SomeParameter” value=”someString”/>
   </gridServiceDeploymentDescriptor>
</gridContainer.config
```

The above deployment descriptor describes a "persistent" Grid Service (one whose lifetime is managed by the container), along with information about the service skeleton class, the assembly the service class is to be found in, and a number of service specific parameters. Transient services are also described in this manner. The current deployment and naming scheme presents the following problems:

- A persistent service can only have one instance deployed within the container. This is because service deployment information is looked up based on the serviceClass value

- Arbitrary names cannot be used for service instances.

We propose a deployment scheme that looks like the following:

```
<gridContainer.config>
   <containerProperties>
   …
   </containerProperties>
   <serviceTypes>
      <serviceType
         typeId=”MyServiceType”
         skeletonClass=”SkeletonClassName, AssemblyName”
         proxyFilePath=”someDir/ProxyFile.asmx”>
         <serviceParameter name=”name” value=”value”/>
      </serviceType>
   </serviceTypes>
   …
   <containerManagedServices>
      <!-- note that the type attribute value references the above typeId attribute value -->
      <service type=”MyServiceTypeName” identifier=”someStringName”>
          <serviceParameter name=”name” value=”value” />
      </service>
      ….
   </containerManagedServices>
</gridContainer.config>
```

This refactoring lends clarity to the deployment model, and allows the service author to use the same code for both container-managed and factory-created services. It removes the divide between “persistent” and “transient” services within the container and therefore makes the coding of the persistence logic simpler.

The second re-factoring is a re-factoring of terminology. We shall cease to refer to persistent and transient services and instead refer to “container-managed” and “client managed” services.

The third re-factoring is that of the registry of active services. At the moment, the `OgsiContainer` class and associated registry classes maintain separate lists of container-managed and client managed service instances. Client managed service instances are registered by a randomly generated string that functions as their instance identifier and form part of their handle. Container-managed service instances are indexed by a string containing their path on the server. It is recommended that both container-managed and client managed services adopt the naming convention currently used by client managed services. Both will then have handles of the form:

http://servername/ogsa/services/someProxy.asmx?instanceId=someString

This will have the added advantage of allowing multiple deployments of the same type of container-managed service instances at the same service endpoint (i.e. using the same proxy file).

The fourth re-factoring is the removal of the `PersistentGridServiceSkeleton`, PersistentGridServiceInstanceAspProxy and TransientGridServiceInstanceAspProxy classes. We will also implement the (currently abstract) InitialiseServiceObject method of GridServiceInstanceAspProxy to use the new service naming and indexing scheme. GridServiceInstanceAspProxy will also be renamed to GridServiceProxyBase. GridServiceSkeleton will be given a Boolean property called IsContainerManaged to indicate whether the service instance is container-managed. GridServiceProxyBase will use this property in its CallMethodOnPortType method when deciding whether to accept or reject operation calls (at the moment it makes the decision depending on the type of the service skeleton class). This will allow use to use the same proxy class for both factory-created and container-managed services.

The fifth re-factoring is that of the ServiceActivator class. At the moment this class has some code that operates differently depending on whether an instance is container-managed or factory-created. This refactoring will remove this duplicate code. ServiceActivator also needs to set the IsContainerManaged property on container-managed services to true.

The sixth re-factoring is deprecating the PostCreate method of GridServiceSkeleton and to refactor the functionality of GridServiceSkeleton that implements the GridService portType out to a separate provider class that can be used in the normal manner via OgsiPortTypeAttribute, which can then use the same serialisation logic as the other portTypes.

3.3 Saving and Loading Service State

In order to meet the needs of the scoped requirements, i.e. services within the container survive a scheduled container restart, we need to implement the following gross algorithms

- In the Application_OnEnd ASP.NET event handler in Global.asax.cs, we issue the command to serialise the state of all services in the container to persistent media, and save a value in a configuration file so that the application knows to restart services on the next startup.
- A check in the Application_OnStart event handler that investigates whether this is a container restart or not. This could involve looking up a value in a configuration file. If it is a restart, the container proceeds to re-activate service instances from persisted state. If it is not a restart, the container uses configuration information to initialise container-managed services in the usual manner.

3.4 Serialisation of Service State

The .NET framework provides extensive capabilities for the serialisation and deserialisation of object graphs. The following pseudocode illustrates the ease of this process:

```
MemoryStream objects = new MemoryStream();
// Create a binary formatter object and use it to serialize
// some instance from the container to a stream. The stream
// can then be persisted.
BinaryFormatter formatter = new BinaryFormatter();
GridServiceSkeleton someInstance =
    LookUpSkeletonInContainer(skeletonIdentifier);
formatter.Serialize( objects, someInstance);
```

The stream can then be saved directly as a byte stream to a BLOB field in the database, or to a file. De-serialisation (which would be performed when the container starts up) follows a similar scheme.

Custom serialization of objects may be carried out in one of two manners. In the first approach, the class to be serialized is marked with the `[Serializable]` attribute. All appropriate serializable fields are automatically serialised to the output binary stream. The second method involves implementing the `ISerializable` interface. This gives the developer more fine grained control over what exactly is serialised, and is our preferred option.

The simplest option is to save the state of the object graph to the database every time the service instance is accessed. This guarantees that the last known good state will be loaded. A problem with this approach is that serialisation is compute intensive and may slow the server down.

A second option is to persist the object graph to the database only when it is known that the state has changed. However this is likely to result in burdens on the service developer, as only they know for certain where the state has been changed, and we wish to make this feature as invisible as possible.

A third option is have a background thread that serialises the object graph after a specified interval (e.g. five seconds). This reduces the compute time associated with serialisation, but does mean that the last stored state may not match the final state of the object, if it changes within this time period. It is not clear if this does in fact offer reliability benefits, since the state obtained upon restart may be out of date due to manipulation since the last serialisation.

A fourth option is to store the state only when the application ends. ASP.NET allows you to implement a method called `Application_OnEnd` that is triggered precisely once. However, it is unclear whether this method is called in the case of a fatal crash, in which case any serialisation would not be performed. It is definitely triggered during a manual shutdown, which is the aim of this iteration of development.

A fifth option is a combination of the last two – infrequently store the object graph, which will offer some sort of state recovery in the event of fatal crashes, and store the object graph during `Application_OnEnd`.

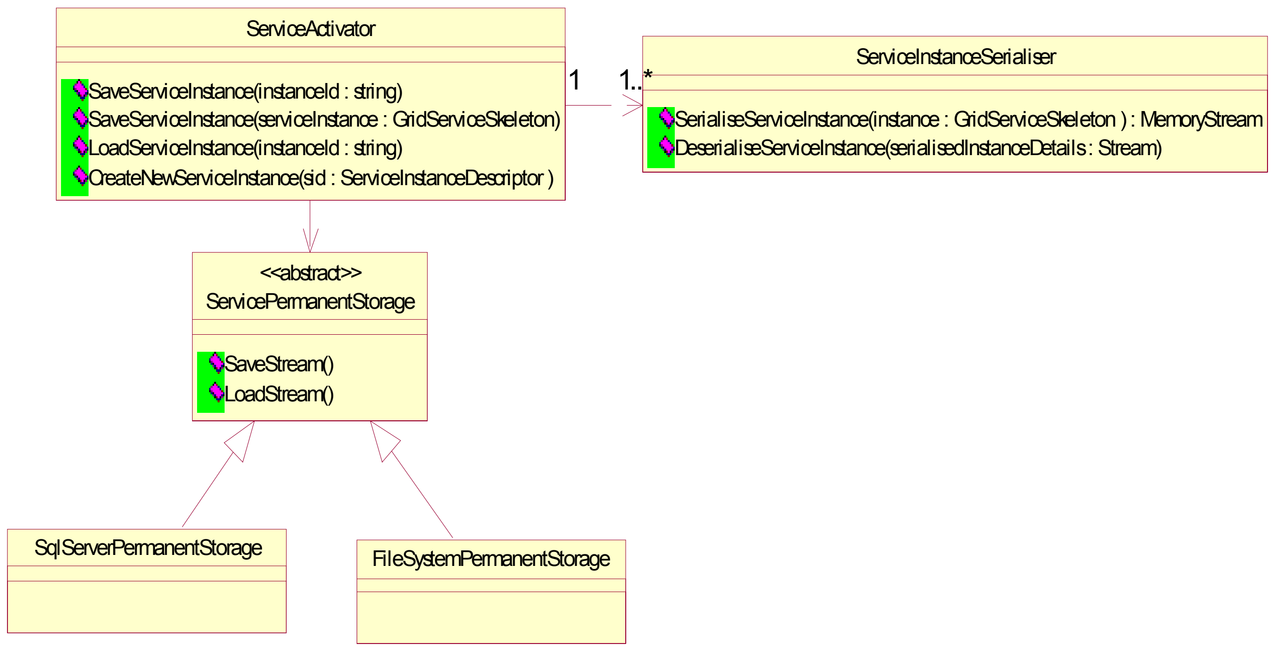
The solution we will implement is option 4.

3.4.1 Problems with the serialisation approach and solutions

We do not propose to solve all serialization problems for developers. As part of the persistence framework we provide, we mandate that developers must implement their own serialization logic.

3.4.2 Object Persistence Framework

The following UML diagram illustrates the proposed set of classes to deal with object persistence:


ServiceActivator
SaveServiceInstance(instanceId : string)
SaveServiceInstance(serviceInstance : GridServiceSkeleton)
LoadServiceInstance(instanceId : string)
CreateNewServiceInstance(sid : ServiceInstanceDescriptor)
1
1..*
ServiceInstanceSerialiser
SerialiseServiceInstance(instance : GridServiceSkeleton) : MemoryStream
DeserialiseServiceInstance(serialisedInstanceDetails : Stream)
<<abstract>>
ServicePermanentStorage
SaveStream()
LoadStream()
SqlServerPermanentStorage
FileSystemPermanentStorage

The ServiceActivator class is already present within the MS.NETGrid software and is responsible for the creation of service instances. In our new model, calls to `ServiceActivator` result in the de-serialization of objects resident in the container and persisting to service storage, using the `ServiceInstanceSerialiser` object and and instance of the `ServicePermanentStorage` object. We propose that at least `FileSystemPermanentStorage` be implemented.

4 Scalability Use Cases

UC-RS4 – Client programs create many instances of a service type, possibly exceeding the memory of the server

In the MS.NETGrid software as it stands, all state is stored in memory. It is possible we could use the code developed for the reliability features to "swap out" service instances and their associated state as the strain on memory gets bigger. It would then be possible to load the state back in on a per request basis. This seems to be a recommended strategy for stateful Web applications according to best practices in the community.

UC-RS5 – A particular service instance is being hit by many clients simultaneously

At the moment requests are queued via thread locking on the skeleton object. The performance impact of this is unknown but should be investigated. It is possible that a finer locking strategy may have a positive impact on the performance of the container at a high level.

UC-RS6 – Extra servers are added to the Web Farm hosting the application

MS.NETGrid is designed to work on a single machine, in which all service instances live in the same .NET process (and indeed Application Domain (.NET Term)). Web farms are a common solution to scalability scenarios in industrial high-load web applications. MS.NETGrid-OGSI should deal with the case that we want to add resources to the application in such a manner.

5 Scalability Outline Design

Unfortunately there is no scope in the current phase of development for implementing any of the scalability features. However, it would be possible to leverage much of the code developed for the reliability features in this endeavour in the future.

5.1 Swap out Single Box strategy (UC-RS4)

In the single box scenario, it would be possible to mark service skeleton objects with the last time they were used for handling an operation call. It would then be possible to start a background thread that monitored memory usage and periodically swapped out service instances (using our reliability serialisation code) using some generational algorithm.

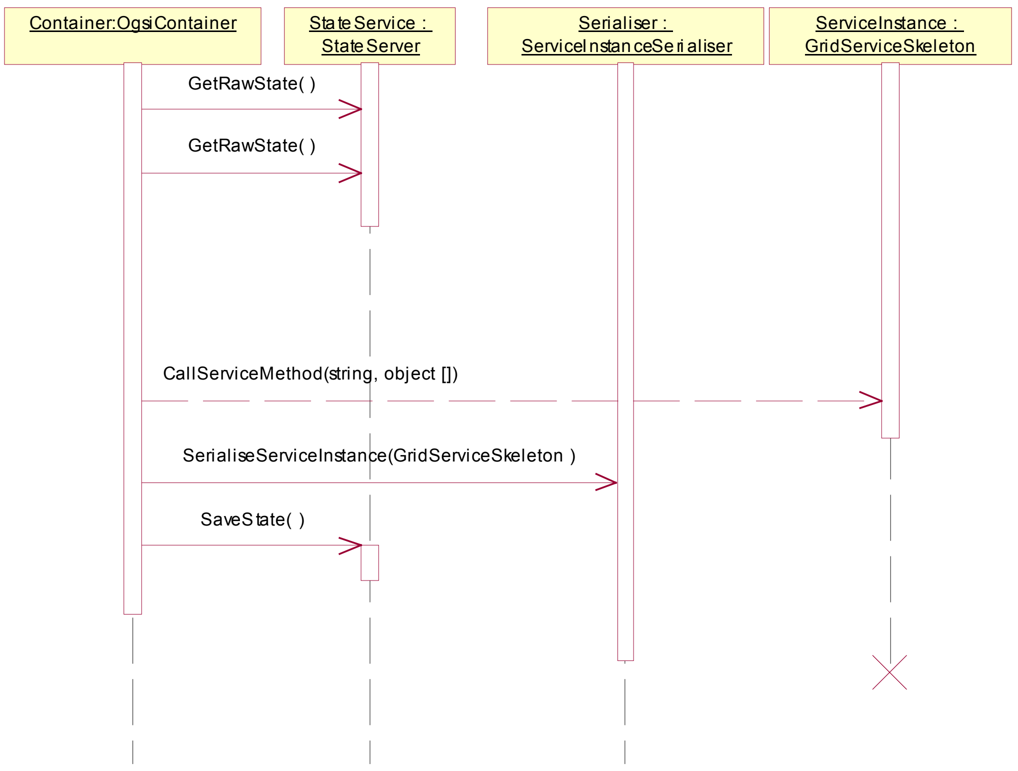
A possible problem here is meeting the notification requirements of OGSI. If state (particularly service data) is stored in the database rather than in memory then how does MS.NETGrid-OGSI know it has changed? This is an issue that may be addressed by a finer grained serialisation strategy. For example, notification subscriptions could be stored independently of other services in a database, and a separate process (perhaps a Windows Service) could continuously monitor the database and execute notification logic when necessary.

5.2 Scale-Out Web Farm Strategy

In this circumstance, we envisage a separate copy of the MS.NETGrid-OGSI application running on every server in the Web farm. Some load balancing mechanism ensures requests are distributed between machines.

In this scenario, state is loaded from the back end database for each request, and the altered skeleton object is saved back to the database after the request has been carried out.

We propose a state service that interacts with the database directly. Serialisation and deserialisation are handled on the outer nodes. The architecture looks like:


Container:OgsiContainer
StateService : StateServer
Serialiser : ServiceInstanceSerialiser
ServiceInstance : GridServiceSkeleton
GetRawState( )
GetRawState( )
CallServiceMethod(string, object [])
SerialiseServiceInstance(GridServiceSkeleton)
SaveState( )

The reliability code we have written can be used to serialise and deserialise the state, as well as save the raw bytes to permanent storage. Locking will have to be implemented by the state service, which keeps a record of the services “checked out”. We can remove the single point of failure of the state service by running multiple instances of it.

6 References

Documents referenced in the text include the following.

[OGSI-Spec] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, Grid Service Specification / OGSI Specification (Draft 29). Open Grid Service Infrastructure WG, Global Grid Forum, April 5th 2003. See http://www.gridforum.org/ogsi-wg.

[OGSI.NET] OGSI.NET Project, Grid Computing Group, University of Virginia. Project WWW site: http://www.cs.virginia.edu/~humphrey/GCG/ogsi.net.html.

[OGSI.NET.Design] G. Wasson, N. Beekwilder and M. Humphrey. OSGI.NET: An OGSI-compliant Hosting Container for the .NET Framework (Draft), Grid Computing Group, Computer Science Department, University of Virginia, April 5th, 2003.

[GTk3A] – Globus Toolkit version 3, Alpha Version, Available at http://www.globus.org/OGSI.