Java OGSI Hosting Environment Design – A Portable Grid Service Container Framework

Thomas Sandholm, Steve Tuecke, Jarek Gawor, Rob Seed
sandholm@mcs.anl.gov, tuecke@mcs.anl.gov, gawor@mcs.anl.gov, seed@mcs.anl.gov
Argonne National Laboratory, IL

Tom Maguire, John Rofrano, Scott Sylvester, Mike Williams
tmaguire@us.ibm.com, rofrano@us.ibm.com, sylvests@us.ibm.com, mdw@us.ibm.com
IBM Poughkeepsie, NY

Abstract

The Open Grid Service Infrastructure (OGSI) specification defines a set of WSDL interfaces to be implemented by various hosting environments. In this document we define one such hosting environment written in Java. We focus on defining the server-side programming model in order to allow Grid services written in one environment to be easily deployed in others. In addition to facilitating service implementation portability, we also define a container framework that is responsible for implementing the low-level infrastructure of OGSI, so that service providers can focus on the application logic. A key design point of the framework is its flexibility to allow for custom dispatchers to be written for a wide range of back-end hosting environments, such as EJB containers, Servlet containers, and CORBA servers. We define a set of interfaces and classes that must be supported to comply with the container framework, and we provide two use cases of applications of this framework; a lightweight Java implementation, and a more enterprise-oriented EJB Entity Bean implementation of a Grid service.

Contents

1 Introduction

The purpose of this document is twofold; 1) to share the experience gained from developing a hosting environment for Open Grid Service Infrastructure (OGSI) based services in Java[1]; and 2) to provide a set of interfaces that can be used to define the interaction between a Grid service implementation, and its container implementing the OGSI required behavior [2][3]. We hope that this effort will lead to an exchange of ideas among different OGSI implementations, such as [1], and [4], in order to achieve Grid service implementation portability in Java. Hence, we focus on the service-provider programming model, and the responsibilities of an OGSI container implementation. The OGSI container could be seen as a request dispatcher between the marshalling engine and the service implementations. The service implementations could be either OGSI container provided implementations of interfaces defined in the Grid Service specification [3], or implementations of service provider defined interfaces. Since the OGSI container is envisaged to integrate a wide range of heterogeneous service implementations, it was designed to support customization of the request flow. First we describe the relationship to other specifications, and in what areas this document extends theses specifications. Then we discuss the proposed server-side programming model, and present the OGSI container interfaces and classes in some more detail. Finally, we provide two use cases of applications of the framework. The first use case exemplifies how to develop a Grid service in a lightweight Java environment, whereas the second use case shows an enterprise Java implementation of a Grid service using the framework.

2 Related Specifications

Neither the WSDL [5] nor the SOAP [6] specification defines how to process requests on the server-side. This was left out in order to allow for implementations in many different hosting environments using various programming models to be utilized. Since both of these specifications are widely applied in today's Web services implementations and specifications, they will however indirectly have an impact on the server-side programming model defined here. JAX-RPC [7] defines a server-side programming model in a Servlet container [8]. It is based on a stateless object-pooling based lifecycle model, which does not map very well onto the soft state model of a Grid service. JAX-RPC, however, defines XML Schema [9] and WSDL to Java mappings, which we adopt in our framework. The Web Services for J2EE specification [10] extends the JAX-RPC server side-programming model to facilitate Web service implementation hosting in EJB [11] containers as stateless Session Beans. From a service provider's point of view it therefore does not provide any new interfaces or APIs apart from those already defined in the EJB specification. When integrating EJBs into our framework we take a very similar approach. The main difference in our approach is that we allow stateful Session Beans, as well as Entity Beans to be exposed as Grid services.

3 Scope

We are not defining the client-side programming model in this document, although it is equally important to assure portability among OGSI clients. However, JAX-RPC already defines a rich client-side programming model that can be used to communicate with Grid

services as well. In the future, though, we expect there to be extensions to JAX-RPC in order to make it easier for clients to use OGSI compliant Grid services. We want Grid services to be deployed in a wide range of operating environments, and hence do not define or mandate any deployment, packaging, or configuration in this document. The implementations of an OGSI hosting environment are expected to leverage the features of their respective target environment (e.g the J2EE Servlet or EJB models) in this area. Further, the security model is deferred to other documents. The notification model (JAXM [12], JMS [13], Message Driven Beans [11]) will also be dealt with elsewhere.

4 Goal

In the currently available set of specifications there is a gap in terms of expected behavior between receiving a Web service request over for instance SOAP, and dispatching it into a stateful service instance in a scalable manner. Our hope is to came up with a portable dispatching framework that can be used both to write reusable Java Grid services, as well as plugging in dispatchers to other component models hosting for instance EJB, or CORBA objects [14]. The dispatcher framework will furthermore implement the OGSI behavior defined in [3] and allowing it to be fully transparent to the service provider's implementation. In some cases, the service implementation or the custom dispatchers may, however, want to explicitly interact with their container, for instance when managing Service Data Elements (SDEs).

Note that even though JAX-RPC defines mappings between WSDL, and XML Schema constructs; and Java interfaces, and types, our dispatcher framework does not make any assumptions about how the XML types are deserialized into Java types; this is up to the serialization (JAX-RPC) engine implementation being used.

This framework is intended to be compatible both with Java 2 Standard Edition (J2SE) [15] as well as Java 2 Enterprise Edition (J2EE) [13]. Future versions of this document may also address Java 2 Micro Edition (J2ME) [16].

5 Grid Service Port Types as Java Interfaces

The most fundamental prerequisite for portability among Grid service implementations is that they all implement the same core interfaces. By core interfaces we refer to the WSDL interfaces defined in the Grid service specification [2]. We are therefore taking the approach of leveraging the WSDL to Java mappings defined in JAX-RPC. JAX-RPC refers to these interfaces as *Service Endpoint Interfaces*. Grid services exposing any of the Port Types defined in [3] must implement their respective Service Endpoint Interfaces to be compliant with our framework.

6 Implementing Port Type Interfaces

Our design goal has been to allow implementations of the core OGSI interfaces to be provided on behalf of service implementers. Implementations of this design could hence be seen as hosting environments for Grid services. Further, the framework should allow service providers to easily add in implementations of new Port Types through configuration or tool driven code generation. The overall design of the basic components

of the OGSI container architecture is depicted in Figure 1. The `ServiceSkeleton` class is the base class for all Grid services and can hence be compared to a CORBA or Java Object. As the base for all Grid services, this class naturally implements the `GridServicePortType` interface. It also contains a `ServiceDataContainer` allowing instance state and meta data to be published and queried. A `FactoryServiceSkeleton` gives service implementers an easy way to implement OGSI compliant factories. All optional core OGSI Port Types are provided as `DelegationSkeleton`s meaning that they can be contained by and delegated to by a `ServiceSkeleton` instance.

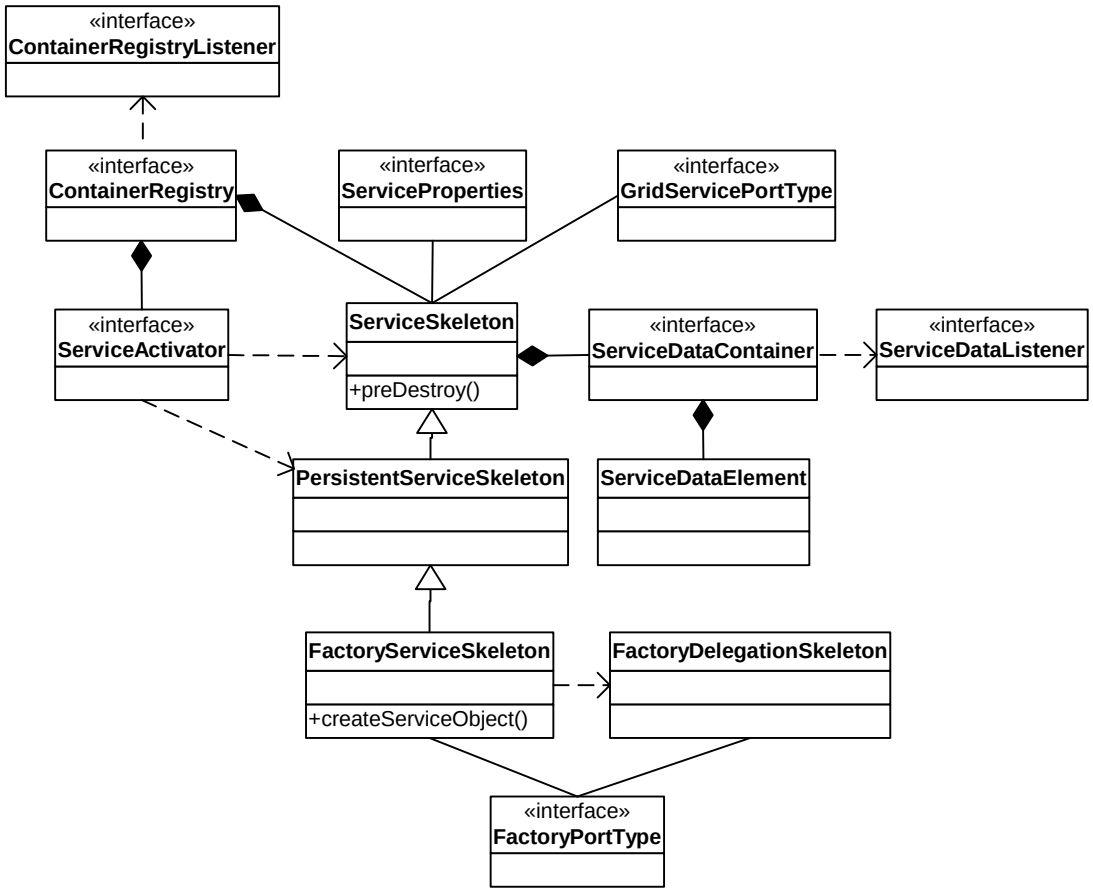

Figure 1: UML Class Diagram of OGSI Container Architecture

So, implementing a service using this container framework typically involves providing a service implementation that inherits from the `ServiceSkeleton` class and delegates to any number of optional, either core or user defined Port Type implementations. Optionally the service implementer may want to provide a Grid service factory by extending the `FactoryServiceSkeleton` class in order to allow dynamic creation of stateful `ServiceSkeleton` instances for its service. If no factory exists for a service it is said to be a persistent service, and must then be made available by the container during initialization, or by using a lazy-loading scheme. Persistent services

should extend the `PersistentServiceSkeleton` class, which is an extension of `ServiceSkeleton`.

7 Registering Service Instances

In order to allow the container to look up contained services when requests are to be dispatched to them, they have to be registered with the container. The `ContainerRegistry` maps an incoming WSDL service endpoint to the service instance using the service path part of the endpoint. Thus, when registering a service, the path has to be conveyed to the container along with a `ServiceSkeleton` or `ServiceActivator` instance. By registering a `ServiceActivator` instance, `ServiceSkeleton`s can be loaded into memory first when being used. In order to passivate stale instances to optimize memory usage, a JAX-RPC Handler can be used that for instance passivates instances on a Least Recently Used (LRU) basis. Either a `ServiceSkeleton` or a `ServiceActivator` has to be registered for a service endpoint in order for the container to dispatch the request successfully. If a `ServiceSkeleton` instance is found in the registry it will pass the incoming request directly to the instance. If however a `ServiceActivator` instance is encountered, the activator is first given a chance to activate the instance before the requested operation is invoked. The activation hooks combined with the factory pattern of OGSI allow you to write custom dispatchers delegating to remote hosting environments with their own lifecycle management.

8 External Sate Management

A Grid service exposes its remotely introspectable state trough Service Data Elements (SDEs). The Grid service specification defines what SDEs the core PortTypes must expose, but service implementers should also be able to add their own information to the set of SDEs contained by a `ServiceSkeleton`. Since this SDE container should be seen as a logical set of XML instances complying to a well defined XML Schema model, we want to allow many different implementations of this container, some of which may be provided by the OGSI container and dispatcher framework provider transparently to the application. But we also want to allow custom SDE containers to be written that are hosted outside of the OGSI. This can be achieved by providing an implementatin of the `ServiceDataContainer` interface, and then associating it with a `ServiceSkeleton` instance.

9 OGSI Container Interfaces and Classes

In this section we describe a set of Java interfaces and classes that define the interaction between an implementation of the OGSI container, and the service implementations and custom dispatchers.

9.1 ServiceSkeleton

```
package org.gridforum.ogsi.provider;

import org.gridforum.ogsi.GridServiceException;
```

```
import org.gridforum.ogsi.servicedata.ServiceDataContainer;
import javax.xml.rpc.handler.MessageContext;

public abstract class ServiceSkeleton
    implements GridServicePortType, ServiceProperties {
  public ServiceDataContainer getServiceDataContainer();
  public void setServiceDataContainer(ServiceDataContainer container);
  public void postCreate(MessageContext context) throws GridServiceException;
  public void postActivate(MessageContext context) throws GridServiceException;
  public void prePassivate() throws GridServiceException;
  public void preDestroy() throws GridServiceException;
}
```

The `ServiceSkeleton` is the base class for all Grid services. It provides an implementation of the `GridServicePortType` (Service Endpoint Interface) generated from the OGSI WSDL Port Type. Further it provides getters and setters to plug in customized implementations of a `ServiceDataContainer`. It is however expected that the OGSI container provides a default `ServiceDataContainer` implementation. All of its non-abstract subclasses must have a default constructor, in order to allow the framework to create instances on demand. In the simplest scenario Grid service providers can let their implementation inherit from the `ServiceSkeleton` directly, but in many cases the extended service skeleton would be a generated dispatcher delegating out to the actual service implementation and optional implementations of the OGSI WSDL Port Types. The postCreate, postActivate, and postPassivate methods can be used to bootstrap, and swap out state information like service data. The creation and activation callbacks are triggered by an incoming request message and these operation take the JAX-RPC defined `MessageContext` that triggered its invocation as input.

9.2 PersistentServiceSkeleton

```
package org.gridforum.ogsi.provider;

import org.gridforum.ogsi.GridServiceException;
import javax.xml.rpc.handler.MessageContext;

public abstract class PersistentServiceSkeleton
    extends GridServiceSkeleton {
  public void postPeristentCreate(MessageContext context)
        throws GridServiceException;
  public void postPersitentActivate(MessageContext context)
        throws GridServiceException;
  public void prePersistentPassivate() throws GridServiceException;
  public void prePersistentDestroy() throws GridServiceException;
}
```

All Grid services that are not to be created by factories should inherit from the `PersistentServiceSkeleton` class, and thus indicating to the OGSI container that it has to make these services available when requests targeting them are encountered.

9.3 Delegation Skeletons

```
package org.gridforum.ogsi.provider;

public class <PorType>DelegationSkeleton
    implements <PortType>PortType {
}
```

Delegation skeletons should be provided for all OGSI WSDL Port Types except the `GridServicePortType`, which is implemented in the `ServiceSkeleton` class. The `ServiceSkeleton` will delegate the requests to these skeletons, which implement the respective Service Endpoint Interfaces for their Port Types. These Port Type implementations can be seen as the default implementations provided by the OGSI container. Service providers may however override these implementations.

9.4 FactoryServiceSkeleton

```
package org.gridforum.ogsi.provider;

import org.gridforum.ogsi.GridServiceException;

public abstract class FactoryServiceSkeleton
    extends PersistentServiceSkeleton, implements FactoryPortType {
  public ServiceSkeleton createServiceObject(Object input)
        throws GridServiceException;
}
```

This class is a convenience class for implementing factories. It allows the OGSI container to provide most of the underlying implementation of the factory on behalf of the service provider. The only method the service provider has to implement is createServiceObject returning a `ServiceSkeleton` that implements the service provider PortType(s).

9.5 ServiceActivator

```
package org.gridforum.ogsi.provider;
import org.gridforum.ogsi.GridServiceException;
import javax.xml.rpc.handler.MessageContext;

public interface ServiceActivator {
  public ServiceSkeleton activate(MessageContext context)
        throws GridServiceException;
  public PersistentServiceSkeleton activatePersistent(
                  MessageContext context) throws GridServiceException;
}
```

The OGSI container will invoke the `ServiceActivator` if a service is registered but not active yet (i.e. no `ServiceSkeleton` instance exists). A service activator is expected to be able to recover the state of a `ServiceSkeleton` instance, in order to make server lifecycles transparent to clients. The activator would commonly also work in conjunction with a passivator that could be implemented using the JAX-RPC Handler framework to provide a LRU cache of skeletons.

9.6 ServiceProperties

```
package org.gridforum.ogsi.provider;

public interface ServiceProperties {
  public Object getProperty(String key);
  public void setProperty(String key, Object property);
}
```

This interface is used to set and get properties on service instances, and thus represents the local state of the instance. This interface is mainly intended to be used to

communicate internal state between delegation skeletons, and thus promoting their decoupling.

9.7 ContainerRegistry

```
package org.gridforum.ogsi.registry;

import org.gridforum.ogsi.provider.ServiceSkeleton;
import org.gridforum.ogsi.provider.ServiceActivator;
import org.gridforum.ogsi.GridServiceException;

public interface ContainerRegistry {
  public void registerService(ServiceSkeleton service,
                              String servicePath) throws GridServiceException;
  public void registerActivator(ServiceActivator activator,
                             String activatorPath) throws GridServiceException;
  public void unregisterActivator(String activatorPath)
        throws GridServiceException;
  public ServiceSkeleton unregisterService(String servicePath)
                             throws GridServiceException;
  public void passivateService(String servicePath) throws GridServiceException;
  public ServiceSkeleton lookup(String servicePath)
        throws GridServiceException;
  public Collection getServices(String servicePath)
        throws GridServiceException;
  public void addListener(String servicePath,
              ContainerRegistryListener listener) throws GridServiceException;
  public void removeListener(String servicePath,
              ContainerRegistryListener listener) throws GridServiceException;
}
```

The `ContainerRegistry` is a local registry of services currently registered in the container. The OGSI container cannot dispatch the request if the service is not registered in this container. A lookup of a service may result in activation if the service has been associated with an activator. The services are ordered in a hierarchical structure determined by the servicePath. An activator registered at a certain path will be able to activate all services below itself in the hierarchy. Factories are expected to register the services they created in sub trees as well, thus allowing per factory activators to be registered.

9.8 ContainerRegistryListener

```
package org.gridforum.ogsi.registry;

public interface ContainerRegistryListener {
  public void registryChanged(String registryPath);
}
```

If a registry entry changes under the given path a notification is sent out to all registered listeners on that path.

9.9 ServiceDataContainer

```
package org.gridforum.ogsi.servicedata;
public interface ServiceDataContainer {
  public ServiceDataElement createServiceData()throws GridServiceException;
  public void addServiceData(ServiceDataElement data)
        throws GridServiceException;
  public Object findServiceData(Object queryExpression)
```

```
        throws GridServiceException;
  public ServiceDataElement getServiceData(String name)
        throws GridServiceException;
  public ServiceDataElement removeServiceData(String name)
        throws GridServiceException;
  public void addListener(String name,
                ServiceDataListener listener) throws GridServiceException;
  public void removeListener(String name,
                ServiceDataListener listener) throws GridServiceException;

}
```

A service data container must be associated with all Grid services. It allows Service Data Elements (SDEs) to be created, published, and discovered. The SDEs are generated from schemas that comply with the XML Schema definition of an SDE in the Grid Service specification [3] according the mapping defined by JAX-RPC. The GridServiceSkeleton will call the findServiceData method when a remote findServiceData request is received.

9.10 ServiceDataListener

```
package org.gridforum.ogsi.servicedata;

public interface ServiceDataListener {
  public void serviceDataChanged(String serviceDataName);
}
```

If a service data value changes a notification is sent out to all registered listeners on that name.

10 Use Case: Light-weight Java Grid service Implementation

In the design shown in Figure 1, there are two implementation approaches that a service can choose from; including direct sub-classing of the `ServiceSkeleton`, and `FactoryServiceSkeleton` classes; and delegation. These implementation choices will allow us to meet the requirements to support a service implementation as both a Java Object, and an EJB (exemplified in the next section), depending on the specific QoS required by the service. Even in a Java Object scenario, the delegation model may be useful if, for instance, an existing application needs to be exposed as a Grid service.

The following examples implement a counter Grid service. The counter Grid service is a very simple stateful service that supports the operations add(), subtract() and getValue().

10.1 Java Subclass Implementation of Counter Grid service

The subclass implementation of the Counter grid service includes a Java implementation of the counter that is derived from the `ServiceSkeleton` and a Java implementation of the counter factory that is derived form the `FactoryServiceSkeleton`.

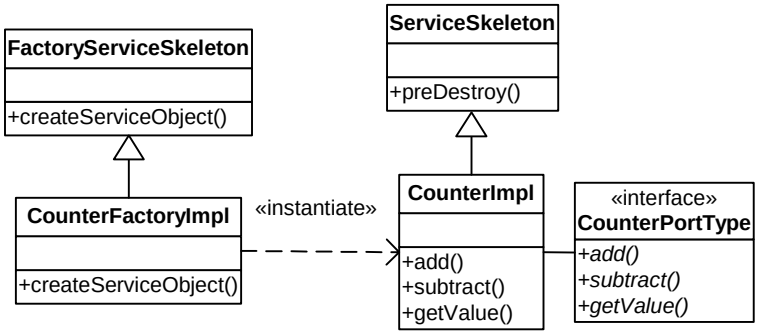

Figure 2: Subclass implementation of Counter Grid service

10.1.1 Tool Generated Code

The code generated in the Java subclass implementation consists of the following:

1. The JAX RPC client proxies and associated Java interfaces. In the above example, the `CounterPortType` is one of these artifacts, which is also used by the service implementation. Note, the other client side artifacts are not shown on this model. The JAX-RPC provider tools typically generate this code.
2. The factory implementation, which in this example is represented by `CounterFactoryImpl`.
3. A service implementation shell or skeleton that will need to be augmented with the applicable business logic. `CounterImpl` represents this in the above example.

10.1.2 CounterFactoryImpl.java Source Code

```
package org.globus.ogsa.impl.samples.counter.basic;

import org.gridforum.ogsi.provider.FactoryServiceSkeleton;
import org.gridforum.ogsi.provider.ServiceSkeleton;
import org.gridforum.ogsi.GridServiceException;

public class CounterFactoryImpl extends FactoryServiceSkeleton {
    public ServiceSkeleton createServiceObject(Object input)
            throws GridServiceException {
        return new CounterImpl();
    }
}
```

10.1.3 CounterImpl.java Source Code

Note, the code generated by the tooling would include the signature for CounterImpl, however would not include the business logic associated with this signature. In this example, the Grid Service developer would be responsible for the implementation of the add(), subtract(), and getValue() methods which will be exposed as operations on the service.

```
package org.globus.ogsa.impl.samples.counter.basic;

import org.globus.ogsa.samples.CounterPortType;
import org.gridforum.ogsi.provider.ServiceSkeleton;
import java.rmi.RemoteException;

public class CounterImpl extends ServiceSkeleton
                         implements CounterPortType {
    private int val = 0;
    public int add(int val) throws RemoteException {
        this.val = this.val + val; return this.val;
    }
    public int subtract(int val) throws RemoteException {
        this.val = this.val - val; return this.val;
    }
    public int getValue() throws RemoteException {
        return this.val;
    }
}
```

10.2 Java Delegation Implementation of Counter Grid service

In the Java delegation implementation, a service skeleton subclasses `ServiceSkeleton` and delegates the requests to the counter implementation.

10.2.1 Tool Generated Code

It is assumed that the Java delegation implementation will be the implementation approach used by the tooling when exposing, for example, an existing Java class implementation that cannot be modified, as a Grid service.

The code generated in the Java delegation implementation consists of the following:

- The JAX RPC client proxies and associated Java interfaces
- The factory implementation
- A service skeleton, which will contain and exploit the business logic of the Java class via delegation. In the above example, this is represented by `CounterSkeleton.`

There should be no additional Java code that needs to be written by the service developer using this approach.

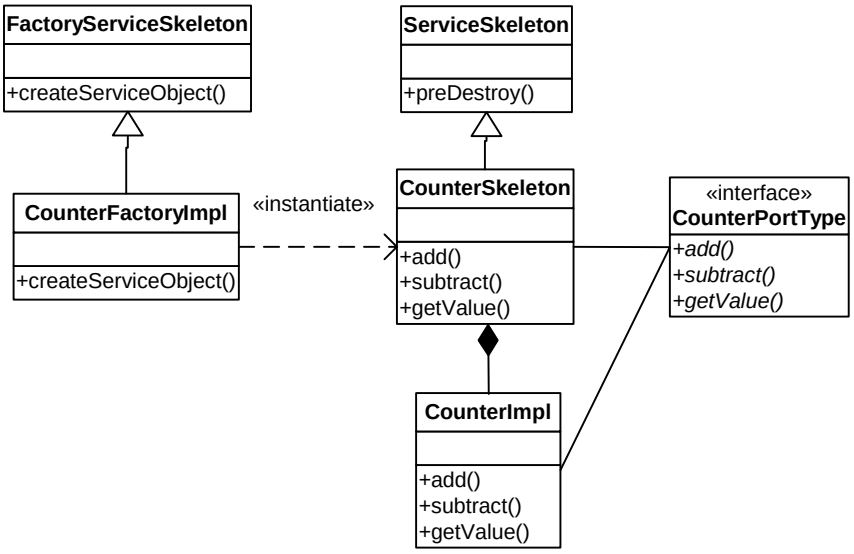

Figure 3: Delegation implementation of Counter Grid service

10.2.2 CounterFactoryImpl.java Source Code

```
package org.globus.ogsa.impl.samples.counter.delegation;

import org.gridforum.ogsi.provider.FactoryServiceSkeleton;
import org.gridforum.ogsi.provider.ServiceSkeleton;
import org.gridforum.ogsi.GridServiceException;

public class CounterFactoryImpl extends FactoryServiceSkeleton {
    public ServiceSkeleton createServiceObject (Object impl)
            throws GridServiceException {
        return new CounterSkeleton(new CounterImpl());
    }
}
```

10.2.3 CounterSkeleton.java Source Code

```
package org.globus.ogsa.impl.samples.counter.delegation;

import org.gridforum.ogsi.provider.ServiceSkeleton;
import org.globus.ogsa.samples.CounterPortType;
import java.rmi.RemoteException;

public class CounterSkeleton extends ServiceSkeleton
                    implements CounterPortType {
    private CounterImpl impl;
    public CounterSkeleton(CounterImpl impl) {
        this.impl = impl;
    }
    public int add(int val) throws RemoteException {
```

```
        return this.impl.add(val);
    }
    public int subtract(int val) throws RemoteException {
        return this.impl.subtract(val);
    }
    public int getValue() throws RemoteException {
        return this.impl.getValue();
    }
}
```

11 Use Case: Custom EJB Dispatcher

J2EE technology provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. J2EE simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically, without complex programming. J2EE has two containers that isolate applications from underlying platforms. These are the EJB container and the Web container. Grid service implementations can be rendered in either container with varying qualities of service. The EJB container manages things like security, transaction, concurrency, and persistence so that application developers can focus on business behavior instead of infrastructure behavior. If this level of infrastructure support is not required, grid services can also be implemented as Java Objects, as exemplified in the previous section. Either way, implementing Grid services within the J2EE component model leverages this base and all it has to offer software developers.

Basing grid services on the J2EE model is also important because it reuses a familiar programming model instead of introducing a new one. Developers familiar with J2EE and EJBs can quickly get started creating Grid services.

A full explanation of the EJB model is beyond the scope of this document. The important concepts being leveraged here are primary classes that comprise an EJB. These are the Home interface, the Remote interface and the Enterprise bean itself. Figure 4 shows the Counter example implemented as an EJB.

11.1 Grid Service as an EntityBean

When the `Counter` class is implemented as an EJB, there are actually four classes that are needed. This is because `Counter` has persistent state and should be implemented as an Entity Bean. For services without persistent state, or states that doesn't need to be preserved between server restarts, Session Beans could be used. The `Counter` class signature becomes the remote interface and retains the original name of the class for clients to refer to. It only has the methods defined in the public interface of the original class because it is a stub that will delegate back to the actual enterprise bean.

`CounterBean` is the persistent enterprise bean class, which contains the value attribute and the implementations for the accessor methods and business methods. It contains all the logic from the original class. It also has several methods that are required by the EJB specification for managing the lifecycle of the bean.

All EJBs are created via a factory. The factory is called the `HomeInterface` and is implemented by the `CounterHome` class in this example. Finally there is a `PrimaryKey` class, which is used in `findByPrimaryKey` operations to uniquely identify the entity EJB. This class is only generated for Entity beans. The only class that was defined by the developer was the original `Counter` class. The four classes that are presented here were generated by EJB tooling. This is the same strategy used for developing Grid services.

OGSA Grid service developers can develop their business functions just like any other Java class or EJB. The tools that will be provided will generate code that leverages this design and wraps the Home interface, Remote interface, and actual Enterprise bean, respectively and add the behavior required of a Grid Service. It is important to note that if you develop your service as a Java object and later want to change it to an EJB implementation, you must adhere to the J2EE specification and not do anything that would violate the EJB container contract (e.g., like spawning your own threads or accessing resources that are not container managed without using the Java Connector Architecture (JCA)).

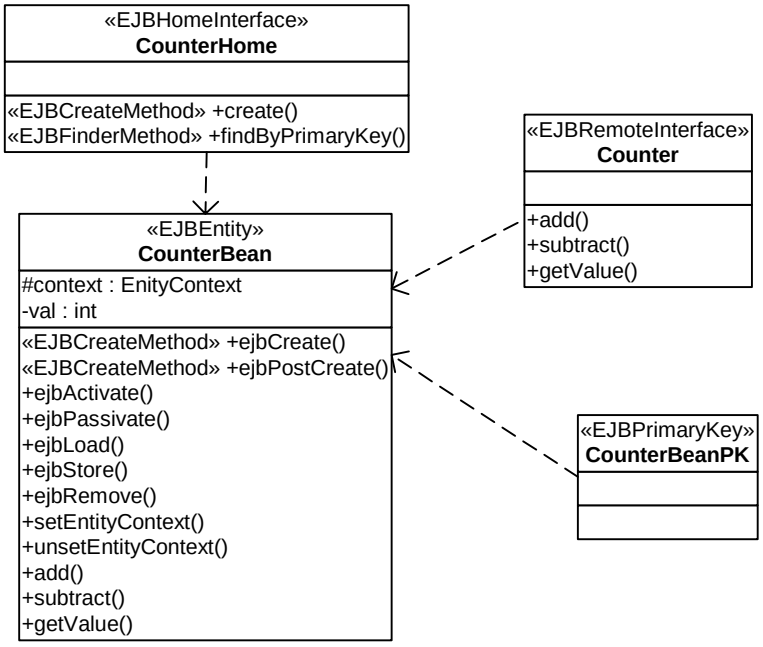

Figure 4: EJB Counter Components

The technique used to support a Grid service EJB implementation is very similar to the Java delegation implementation model described in section 10.2. In this case, the service

skeleton (`CounterSkeleton`) in our example is what integrates the EJB into the OGSI container framework.

11.1.1 Tool Generated Code

The tooling support for EJBs provides the ability to expose an existing EJB as a Grid service.

The code generated in the EJB implementation of a Grid service consist of the following:

- The JAX RPC client proxies and associated Java interfaces.
- The factory implementation, which in this example is represented as `CounterFactoryEjbImpl`.
- A service EJB skeleton, which will contain a remote reference to the service EJB that is represented as `CounterEjbSkeleton` in this example.

Exposing an existing EJB will have the same characteristics, as the use of an existing Java class in that no additional code will need to be developed by the service implementer.

11.1.2 CounterFactoryEjbImpl.java Source Code

```
package com.ibm.ogsa.impl.demo.counter.ejb;

import org.gridforum.ogsi.provider.FactoryServiceSkeleton;
import org.gridforum.ogsi.provider.ServiceSkeleton;
import org.gridforum.ogsi.GridServiceException;

import com.ibm.ogsa.sample.CounterHome;
import com.ibm.ogsa.sample.Counter;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class CounterFactoryEjbImpl extends FactoryServiceSkeleton {
    public ServiceSkeleton createServiceObject (Object input)
          throws GridServiceException {
       CounterEjbSkeleton aCounterSkeleton = null;
       try {
             Context jndiContext = new InitialContext();
             Object ref =
                 jndiContext.lookup("ejb/com/ibm/ogsa/sample/CounterHome");
             CounterHome home =
                 (CounterHome)PortableRemoteObject.narrow(ref,
                                                   CounterHome.class);
             Counter anEJBCounter = home.create();
```

```
            aCounterSkeleton = new CounterEjbSkeleton(anEJBCounter);
        } catch (Exception e) {
            throw new GridServiceException(e);
        }
        return aCounterSkeleton;
    }
}
```

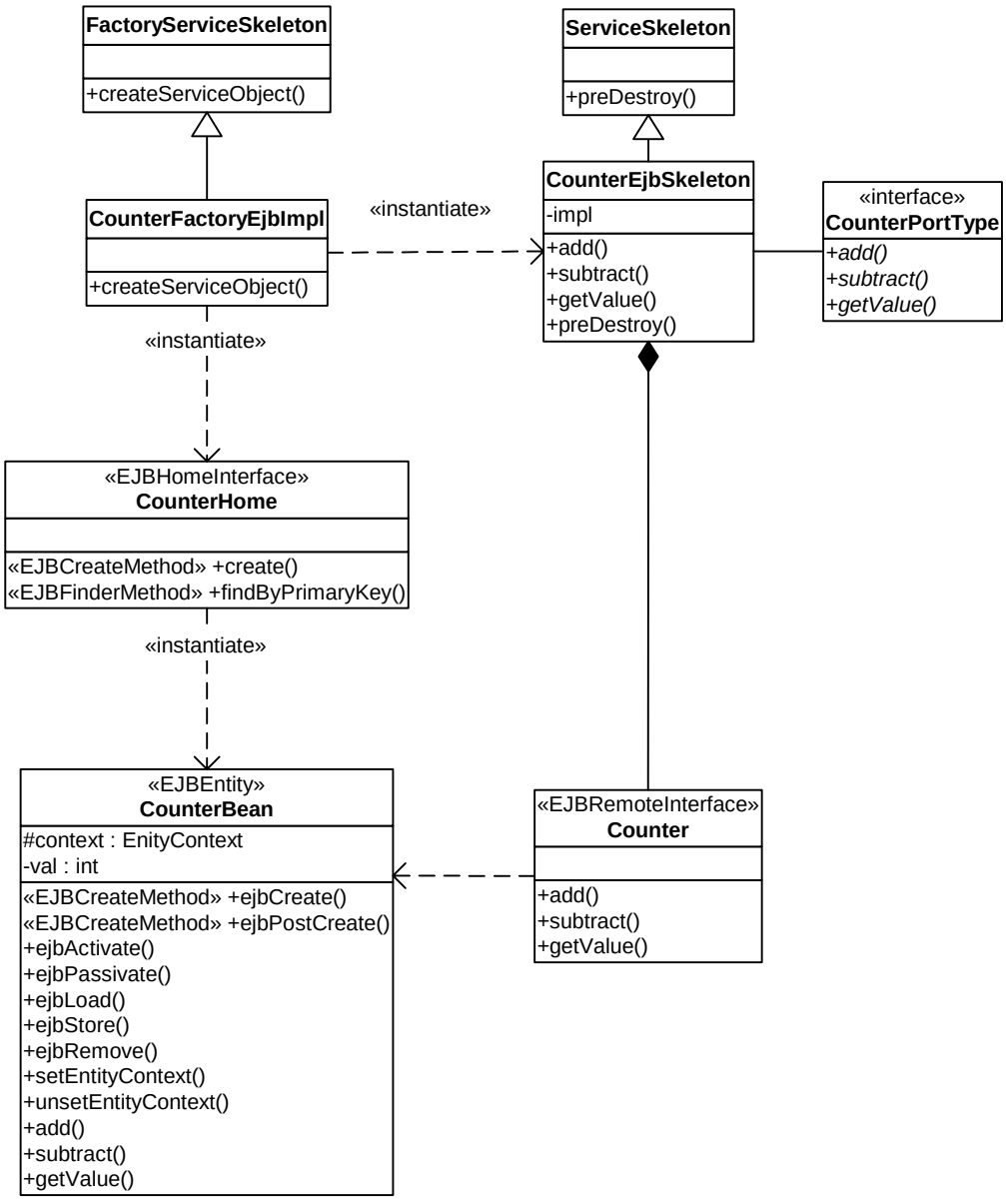

Figure 5: EJB implementation of Counter Grid service

`CounterFactoryEjbImpl` is the factory implementation of the EJB counter Grid service. This factory encapsulates the details associated with locating the EJB home

interface, creating the counter bean, and passing the reference to the EJB remote interface to the service skeleton. The `createSeviceObject` operation creates the service skeleton in addition to the actual counter EJB and returns the `CounterEjbSkeleton` with the remote EJB reference.

11.1.3 CounterEjbSkeleton.java Source Code

```
package com.ibm.ogsa.impl.demo.counter.ejb;

import org.gridforum.ogsi.provider.ServiceSkeleton;
import org.gridforum.ogsi.GridServiceException;
import org.globus.ogsa.samples.CounterPortType;
import com.ibm.ogsa.sample.Counter;

import javax.xml.rpc.handler.MessageContext;
import javax.ejb.RemoveException;
import java.rmi.RemoteException;

public class CounterEjbSkeleton extends ServiceSkeleton
                        implements CounterPortType {
    private Counter impl;
    public CounterEjbSkeleton(Counter impl) {
        this.impl = impl;
    }
    public int add(int val) throws RemoteException {
        return this.impl.add(val);
    }
    public int subtract(int val) throws RemoteException {
        return this.impl.subtract(val);
    }
    public int getValue() throws RemoteException {
        return this.impl.getValue();
    }
    public void preDestroy() throws GridServiceException {
        try {
           impl.remove();
        } catch (RemoveException re) {
            throw new GridServiceException(re);
        }
    }
}
```

`CounterEjbSkeleton` contains the remote interface to the Counter Session EJB (impl) and is responsible for delegating operations to business methods implemented by the EJB.The `preDestroy` operation is the callback method that is used to notify this service skeleton that the service will be destroyed. In this case, the Counter EJB is removed. Note that the framework will call `preDestroy` when clients explicitly destroy the Grid service, as well as when the soft-state timeout of the Grid service expires.

12 References

1.	Globus Open Grid Services Architecture Development Framwork (OGSADF). www.globus.org/ogsa/TechPreview
2.	Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002, www.globus.org/research/papers/ogsa.pdf
3.	Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S. and Kesselman, S. Grid Service Specification. Globus Project; Draft 2, 6/13/2002. www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft02_2002-06-13.pdf
4.	Unicore OGSA Demonstrator. Unicore, 2002. www.unicore.org/downloads.htm
5.	Web Services Description Language (WSDL) 1.1: www.w3.org/TR/wsdl
6.	W3C: SOAP 1.2: www.w3.org/TR/2001/WD-soap12-20010709/
7.	Sun Microsystems. Java API for XML-based RPC. JAX-RPC 1.0, JSR 101. java.sun.com/xml/jaxrpc/
8.	Sun Microsystems. Java Servlet 2.3 Specification. JSR 53. www.jcp.org/aboutJava/communityprocess/final/jsr053/
9.	W3C Recommendation. XML Schema Part 2: Datatypes. www.w3.org/TR/xmlschema-2/
10.	IBM. Web Services for J2EE, Version 1. JSR 109. www.jcp.org/aboutJava/communityprocess/review/jsr109/
11.	Sun Microsystems. Enterprise Java Beans Specification, Version 2. JSR 19 www.jcp.org/aboutJava/communityprocess/final/jsr019/
12.	Sun Microsystems. Java APIs for XML Messaging, Version 1.1 www.jcp.org/aboutJava/communityprocess/final/jsr067/index2.html
13.	Sun Microsystems. Java 2 Enterprise Edition (J2EE). java.sun.com/j2ee/
14.	OMG. Common Object Request Broker: Architecture and Specification, Revision 2.2. Object Management Group Document 96.03.04, 1998.
15.	Sun Microsystems. Java 2 Standard Edition. (J2SE). java.sun.com/j2se/
16.	Sun Microsystems. Java 2 Micro Edition (J2ME). java.sun.com/j2me/

Appendix A: Open Issues

Service Data Container Hosting in EJB	sandholm	07/13/2002
Service Data XML Collection Representation	sandholm	07/13/2002
Service Data Container Hosting in EJB	sandholm	07/13/2002
Example showcasing activtion/passivation and EJB primary key mapping	sandholm	07/13/2002
PersistentProperties integration	sandholm	07/13/2002
Delegation Skeleton Portability and ServiceProperties propagation	sandholm	07/13/2002
Representation of Passivated Services in Container Registry	sandholm	07/13/2002
How to get a reference to a ContainerRegistry	sandholm	07/14/2002
Transient Factory Support	sandholm	07/14/2002
Define GridServiceException hierachy	sandholm	07/15/2002