Project Number: **034702**
Project Acronym: **BEinGRID**
Project Title: **Business Experiments in Grid**
Instrument: **Integrated Project**
Thematic Priority: **Advanced Grid Technologies, Systems and Services**

Design Pattern for a GT4 Service receiving WS-notifications

Activity1: *Technical Common Cross Activities*

WP 1.7: *Component Development*

Submission Date:		25/02/2009
Start Date of Project:		01/06/2006
Duration of Project:		42 months
Organisation Responsible for the Deliverable:		Atos Origin
Version:		0.3
Status		Final
Author(s):	Igor Rosenberg	Atos Origin
	Roland Kübert	HLRS

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	18/02/2009	Initial draft based on Globus documentation	Igor Rosenberg (Atos)
0.2	24/02/2009	Updates based on first review of Roland	Igor Rosenberg (Atos) Roland Kübert (HLRS)
0.3	25/02/2009	Updates based on second review of Roland – Final version	Igor Rosenberg (Atos) Roland Kübert (HLRS)

© BEinGRID consortium

Table of Contents

1. Executive Summary

Making a GT4 service receive WS-Notification notifications is tricky. Indeed, a WS-service has no permanent existence; a WS-service is a service/resource pair, which is coupled on-demand. Assuming that such a pair is available for a long time is bad practice: computers hosting the pair may crash. The programming model proposed below addresses this issue, by presenting how a service can generate stand-alone listener threads, which must contain the necessary logic to re-instantiate the destination service/resource pair when a previous instance is no longer available. The client code is provided in Annex A.

The proposed programming model relies on the GT4 framework capabilities. This allows for fault-tolerance and scalability to be handled by the container, not the services. But it may provoke a performance draw-back, which would need to be evaluated. A variation of the model is also presented for completeness, even though it is not recommended as it doesn't provide such clear separation of concern.

2. Introduction

2.1 Purpose

This document presents how a Web Service implemented under GT4 can receive WS-Notification notifications, preserving the service (stateless) and resource (state container) separation, and making sure the listener is independent and located in the web service container.

2.2 References

[1] *Globus: WSRF - The WS-Resource Framework*, http://www.globus.org/wsrf/

[2] Ian Foster et. al, *Modelling Stateful Resources with Web Services*, Version 1.1, 03/05/2004, http://www-128.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf

[3] Borja Sotomayor, *The Globus Toolkit 4 Programmer's Tutorial*, 2004, http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html

[4] Borja Sotomayor, Lisa Childers, *Globus® Toolkit 4 : Programming Java Services*, Morgan Kaufmann; 1st Edition (December 16, 2005)

[5] "Using Eclipse to develop grid services" of IBM developerWorks, tutorial combining GT4, apache tomcat and eclipse for integrated development. One version can be found at www.datamininggrid.org/wdat/works/att/ljudoc005.content.05301.pdf

[6] Globus Documentation: Section 5.2.1.2. Setting up and receiving notifications (Notification Consumer) http://globus.org/toolkit/docs/4.0/common/javawscore/developer-index.html

2.3 Definitions

Special attention should be cast upon the WSRF concepts. This is presented clearly in Chapter 4 of [4]. In the rest of this document, the following (simplified) definitions will be used:

A *resource* is an independent entity stored by the container which stores state information.

A *Web Service* is a stateless open standards interface allowing to access web-based applications.

A *WS-Resource* is the pairing of a Web-Service with a resource, effectively making the pair accessible and stateful.

The short name *service* is used herein to describe an instance of a WS-Resource.

3. Design

GT4 has extensive documentation on how a service can be used as a WS-Notification server, and how to create a stand-alone client (for example, from the command-line). But receiving notifications from a service carries a different execution context, and is not addressed in the documentation. So the following section presents the architecture enabling a service to react on the reception of WS-Notifications.

3.1 High level architecture

The reader should read the official GT4 documentation on using WS-Notifications [6]. The creation of a WS-Notification client seems quite straight-forward. But the documentation does not contemplate the fact that a Web Service must remain stateless, that the resource cannot have an activity of its own and should only be accessed through the corresponding service. This constraining design leaves no room for a background activity, acting as a notification client, which needs to be perpetually idle, listening to its notification server, and performing some action on receiving notifications.

A solution is proposed in Figure 1. GT4 provides a container to store independent notification clients, which can be understood from the documentation as on-the-fly services. For a service needing to receive notifications, such a client is generated, with the task of calling back the service/resource pair through its normal WS interface. This solves the problem highlighted in the previous paragraph, and preserves the access of resources through their canonical access. This allows using the container's fault-tolerance, load-balancing and scalability capabilities transparently. This is performed through the inexistence of the hard link between the notification client and the resource on which it must act upon.

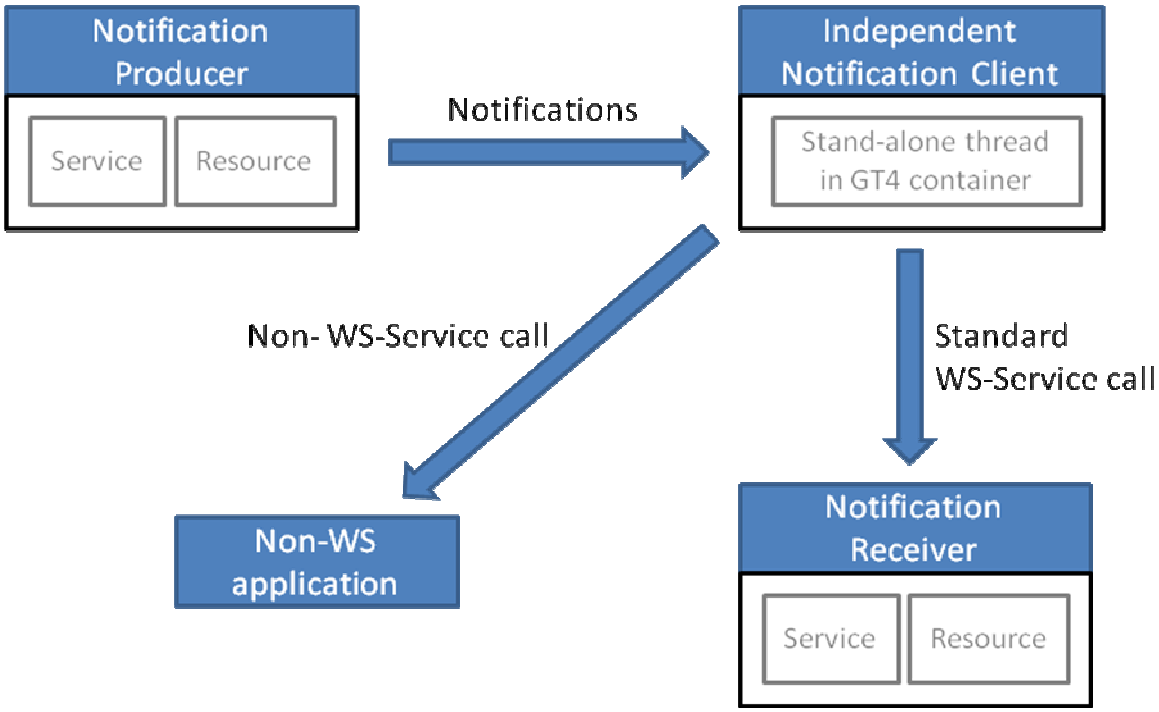

Figure 1 High-level view of the separation of Service and Notification reception

3.2 A notification client

A notification client skeleton is given in [6]. In the context of this paper, we suggest that a client should be an extension of the abstract class `NotificationSubscriber` (presented in Annex A). This instance will execute in the GT4 container (ie independent of the original thread). The notifications are received through calls to its method

```
deliver(List, EndpointReferenceType, Object)
```

This method is flagged as abstract in the class `NotificationSubscriber`. The extension must implement it, and do the needed logic on the notification object.

3.3 A service receiving notifications

When a service wants to be able to react on WS-notification reception, it should create a new instance of a notification client, as described above in section 3.2. This will execute as a thread in the GT4 container (ie separated from the service), and will be the one receiving the notifications through calls to its method

```
deliver(List, EndpointReferenceType, Object)
```

This method should perform the resulting operations. In the case where we actually want a service to react on that notification, we recommend making the calls to the service through its WS interfaces. Implementation shortcuts taking advantage of possible collocation of resource object and notification client thread should be avoided, as they risk coming short of properly reacting to container behaviour for load-balancing, scalability and fault-tolerance.

© BEinGRID consortium

4. Possible critics

4.1 Performance hit

Globus WS-Notifications are slow.

In a simple server/client setup (the client is not a service, the implementation is made following the example defined in [6]), the time between notification emission and notification reception is very variable, and has been measured, on a server prepared for this testing purpose, to be around 2 seconds, with various events lasting more than 10 seconds. This observation must be taken into account, and WS-Notification should not be used if the system implemented is to be reactive. Real-time or near-real-time systems are totally out of reach.

When implementing the design shown in this paper, a new object is created, and the container probably handles it as a new Thread. Upon reception of the notification, the service/resource pair, if un-available, must be recreated, and then the WS-call can be performed. The two operations are time-consuming, and add overhead to the initial notification reception process. If the service is implemented as only an interface to another application (database, legacy application, combination of other services, etc.), this overhead can be spared by implementing directly in the notification client the call to the other application. Duplication of code must be limited to the minimum (calls to other applications should be contained within a class shared by client and service).

4.2 Division of service logic

The service logic is not split between the notification client and the original service implementation. The notification client is only meant to be a proxy to the service, and its implementation should be as minimalistic as possible, relying on the service to perform the business logic. All the code defining behaviour upon reception of notifications should belong to the service code.

4.3 Service-as-Client

One another possible implementation can also rely on registering the service itself as the WS-Notification client. The service should get its resource from the context, and as such can implement directly the deliver interface. The service then has a bit more logic in it, as it contains the listener and the logic. The question which arises in this case is how the service is stored by the container: the service exists in its usual location as a service instance (it has been created and used by its creation call, with its resource), but is also referenced in the “on-the-fly” service location, which is meant to contain notification clients.

```
public class MyService implements NotifyCallback {

      public MyReturnObject myMethod(MyInputObject x) {
            MyResource r = getResource();
            // do any treatment involving the resource and the input
            MyReturnObject o = new MyReturnObject ();
            // set return values
            return o;
```

```
        }

        public void deliver(List topicPath,
                    EndpointReferenceType producer, Object message) {
              ...
        }

        public void subscribe(EndpointReferenceType epr,
                    GSSCredential cred, Calendar expiration) {
              ...
        }

        /**
         * Private method that gets a reference to the resource
         * specified in the endpoint reference.
         */
        private MyResource getResource() throws RemoteException {
              Object resource = null;
              try {
                    resource =
                    ResourceContext.getResourceContext().getResource();
              } catch (NoSuchResourceException e) {
                    throw new RemoteException(
                          "Specified resource does not exist", e);
              } catch (ResourceContextException e) {
                    throw new RemoteException(
                          "Error during resource lookup", e);
              } catch (Exception e) {
                    throw new RemoteException("", e);
              }
              MyResource myResource = (MyResource) resource;
              return myResource;
        }
  }
```

Some may argue the solution of section 3 is an unnecessary indirection. In any case, the design must contemplate the possibility of hardware failure, possibly between calls. The implementation must rely on the fault-tolerance capabilities of the container, and should allow failures to be without impact.

Annex A. NotificationSubscriber.java

```
import java.rmi.RemoteException;

import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceException;

import org.apache.axis.message.addressing.EndpointReferenceType;
import org.apache.axis.types.URI.MalformedURIException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.globus.wsrf.NotificationConsumerManager;
import org.globus.wsrf.NotifyCallback;
import org.globus.wsrf.ResourceException;
import org.globus.wsrf.WSNConstants;
import org.globus.wsrf.container.ContainerException;
import org.globus.wsrf.core.notification.SubscriptionManager;
import
org.globus.wsrf.core.notification.service.SubscriptionManagerServiceAddress
ingLocator;
import org.oasis.wsn.NotificationProducer;
import org.oasis.wsn.Subscribe;
import org.oasis.wsn.SubscribeResponse;
import org.oasis.wsn.TopicExpressionType;
import org.oasis.wsn.WSBaseNotificationServiceAddressingLocator;
import org.oasis.wsrf.lifetime.Destroy;

// import de.hlrs.beingrid.slaaccounting.util.ClientHelpers;

/**
 * Create a container which is independent of its instantiator, and then
 * acts upon receiving notifications.
 * @see NotifyCallback#deliver(java.util.List,
 *                             EndpointReferenceType, Object)
 * This class is implemented as described in
 *http://globus.org/toolkit/docs/4.0/common/javawscore/developer-index.html
 * section 5.2.1.2. Setting up and receiving notifications
 * (Notification Consumer)
 **/

public abstract class NotificationSubscriber implements NotifyCallback {

      private final Log logger = LogFactory.getLog(this.getClass());

      private NotificationConsumerManager consumer = null;
      private EndpointReferenceType consumerEPR = null;
      private SubscribeResponse subResponse = null;
      private boolean successfullyStartedConsumer = false;


      public void startConsumer() {
            try {
                  consumer = NotificationConsumerManager.getInstance();
                  // maybe replace this with new
                  // ServerNotificationConsumerManager() ?
```

```
                try {
                        consumer.startListening();
                } catch (ContainerException e) {
                        logger.error("Cannot create "   +
                                "NotificationConsumerManager", e);
                        return;
                }

                logger.info("Notification Consumer is listening:"
                        +consumer.getURL());

                try {
                        consumerEPR =
                                consumer.createNotificationConsumer(this);
                } catch (ResourceException e) {
                        logger.error("Cannot create notification consumer",
                                        e);
                        return;
                }
                successfullyStartedConsumer = true;
        } finally {
                // clean up in case any step failed
                if ( successfullyStartedConsumer ) return;

                try {
                        if ( consumerEPR != null )
                                NotificationConsumerManager.getInstance()
                                  .removeNotificationConsumer(consumerEPR);
                } catch (ResourceException e) {
                        logger.error("Error during cleanup", e);
                } finally {
                        consumerEPR = null;
                }

                try {
                        if ( consumer != null )
                                consumer.stopListening();
                } catch (ContainerException e) {
                        logger.error("Error during cleanup", e);
                } finally {
                        consumerEPR = null;
                }
        }
}

public void subscribe(
        EndpointReferenceType notificationProducerEPR,
        QName topic) {
        if ( ! successfullyStartedConsumer )
                throw new IllegalStateException("Subscriber not started"+
                                                " (see start())");

        // subscribe to callback
        TopicExpressionType topicExpression =
                new TopicExpressionType();
        try {
```

```
                topicExpression.setDialect(
                      WSNConstants.SIMPLE_TOPIC_DIALECT);
          } catch ( MalformedURIException e ) {
                logger.error("CANNOT HAPPEN");
                throw new RuntimeException("CANNOT HAPPEN", e);
          }
          topicExpression.setValue(topic);

          Subscribe request = new Subscribe();
          request.setUseNotify(Boolean.TRUE);
          request.setConsumerReference(consumerEPR);
          request.setTopicExpression(topicExpression);

          NotificationProducer prod = null;
          try {
                prod = new WSBaseNotificationServiceAddressingLocator()
                  .getNotificationProducerPort(notificationProducerEPR);
          } catch (ServiceException e) {
                logger.error("Cannot create port for specified "
                             +"producer epr", e);
                return;
          }

          // ClientHelpers.makeSecure((Stub)prod);

          try {
                subResponse = prod.subscribe(request);
          } catch (RemoteException e) {
                logger.error("Cannot subscribe to topic + " + topic, e);
                return;
          }
    }

    public void unsubscribe() {
          // cleanup
          try {
                if ( subResponse != null ) {
                SubscriptionManagerServiceAddressingLocator sLocator =
                   new SubscriptionManagerServiceAddressingLocator();
                SubscriptionManager manager =
                   sLocator.getSubscriptionManagerPort(
                            subResponse.getSubscriptionReference());
                      // ClientHelpers.makeSecure((Stub)manager);
                      manager.destroy(new Destroy());
                }
          } catch ( Exception e ) {
                logger.error("Error during unsubscribe", e);
          } finally {
                subResponse = null;
          }
    }

    public void stopConsumer() {
          successfullyStartedConsumer = false;

          try {
```

```
                    if ( consumerEPR != null )
                          NotificationConsumerManager.getInstance().
                                removeNotificationConsumer(consumerEPR);
              } catch (ResourceException e) {
                    logger.error("Error during unsubscribe", e);
              } finally {
                    consumerEPR = null;
              }

              try {
                    if ( consumer != null )
                          consumer.stopListening();
              } catch (ContainerException e) {
                    logger.error("Error during unsubscribe", e);
              } finally {
                    consumerEPR = null;
              }
        }

        /**
         * The deliver method is the one which should be implemented,
         * and which is called upon WS-Notification reception.
         **/
        /*
        @Override
        public abstract void deliver(List arg0, EndpointReferenceType arg1,
Object arg2) {
        }
        */
}
```