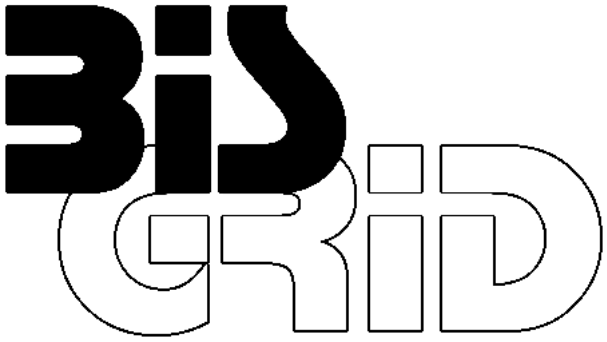

Betriebliche Informationssysteme:
Grid-basierte Integration und Orchestrierung

Deliverable 3.2

Work Package 3: WS-BPEL Engine

Documentation
BIS-Grid Engine Prototype

GEFÖRDERT VOM

Promotional Reference: 01IG07005

Authors:

André Höing
Technische Universität Berlin
Faculty of Information Technologies
Complex and Distributed IT Systems

Stefan Gudenkauf, **Guido Scherp**
OFFIS Institute for Information Technology
R&D-Division Energy

Holger Nitsche, **Dirk Meister**
Universität Paderborn
Paderborn Center for Parallel Computing

This work is supported by the German Federal Ministry of Education and Research (BMBF) under grant No. 01IG07005 as part of the D-Grid initiative.

Date:
31.08.2008

Contents

This document is part of Deliverable 3.2 “Vorab-Version der Basis-BPEL-Engine (Software, Dokumentation)” of work package 3 in the project BIS-Grid[1], a BMBF-funded project of the German D-Grid[2] initiative. It represents the documentation of the BIS-Grid middleware prototype. Please note that the document reflects ongoing work and thus may be complemented by document updates, especially by documents that are parts of Deliverables 3.3 “Erweiterte BPEL-Engine (Software, Dokumentation)” and 3.4 “Finale Version der BPEL-Engine, integriert mit UNICORE (Software, Dokumentation)”. For the specification of the BIS-Grid middleware please see Deliverable 3.1. [5].

[1] http://www.bisgrid.de
[2] http://www.d-grid.de

1 Introduction

This document is part of Deliverable 3.2 and represents the documentation of the BIS-Grid middleware prototype. The document is regarded as a documentation of ongoing work and intended to serve as basis for the future Deliverable 3.4, the official documentation of the then-released BIS-Grid middleware final. For the complete specification of the BIS-Grid middleware please see Deliverable 3.1. [5].

In the following we present a short introduction into the BIS-Grid project. This includes a general overview on BIS-Grid, a top-level description of the BIS-Grid middleware (also referred to as BIS-Grid engine), some information on the use of the software development tool Maven 2 that we use, some information on the project structure and the configuration of the BIS-Grid middleware, and information on the deployment of the BIS-Grid middleware's service extensions to the Grid middleware UNICORE 6[3] that represents the development basis for the BIS-Grid middleware. In Section 2 we discuss the Workflow Management Service of the BIS-Grid middleware, and in Section 3 we discuss the implementation of the Workflow Service. In Section 4 we present the so-called Proxy Service. The document concludes with a short overview on near-future work.

1.1 BIS-Grid - an Overview

In order to map business processes to the technical system level the integration of heterogeneous information systems - referred to as Enterprise Application Integration (EAI) - is crucial. Thereby, integration is often achieved by service orchestration in service-oriented architectures (SOA). A means commonly used to create SOA are Web Services since they enable service orchestration and hide the underlying technical infrastructure. Modern Grid middlewares such as UNICORE 6 and Globus Toolkit 4[4] are based on the Web Service Resource Framework (WSRF) [1], a standard that extends classical, stateless Web Services to be stateful. Such WSRF-based Web Services, also called Grid Services, provide a basis to build SOAs using Grid technologies.

In BIS-Grid we focus on realising EAI using Grid technologies. One major objective is to proof that Grid technologies are feasible for information systems integration. Small and medium enterprises (SMEs) shall be enabled to integrate heterogeneous business information systems and to use external Grid resources and services with affordable effort. To do so, we develop a workflow engine, the *BIS-Grid workflow engine*, that is capable to integrate Grid Services. This engine is based upon service extensions to the UNICORE 6 Grid middleware, using an arbitrary WS-BPEL workflow engine and standard WS-BPEL to orchestrate Grid Services. Also, it propagates service orchestrations as Grid Services. The main reason that led us to the decision to use UNICORE 6 is that UNICORE 6 is a pioneer in adopting Grid standards, since the support of standards is essential for us, especially regarding security. The WS-BPEL workflow engine to be

[3] http://www.unicore.eu
[4] http://www.globus.org/toolkit/

used is ActiveBPEL[5] since it exhaustively supports the WS-BPEL standard, and is well accepted in the business domain as well as in the Grid domain. We refrain from extending well-adopted standards and technologies as far as possible to increase sustainability. Instead, we use service extensions to UNICORE 6 to conceal the WS-BPEL engine by wrapping the message exchange between the engine and Grid Services.

1.2 BIS-Grid Engine in a Nutshell

We developed the BIS-Grid engine as a set services to be deployed to a UNICORE 6 installation. These service extensions mainly consists of two service types, *Workflow Management Service* and *Workflow Service*, and an arbitrary standard WS-BPEL workflow engine. Thereby, the service extensions are WSRF services. Also, in our case the WS-BPEL engine is the open source workflow engine ActiveBPEL. Together, these service extensions and the arbitrary WS-BPEL engine represent the *BIS-Grid workflow engine*. The service extensions are deployed as Grid Services within UNICORE 6's service container, the UNICORE/X component. For each workflow deployed with the Workflow Management Service one Workflow Service will be created using a hot deployment mechanism without restarting UNICORE/X. These services manage and access ActiveBPEL. As a standard WS-BPEL workflow engine, it typically orchestrates stateless Web Services and supports only basic security mechanisms, e.g. username-based and password-based authentication. Therefore, advanced security concepts must be provided by the service extensions in the UNICORE/X service container. In [4] we illustrate some considerations on security within the BIS-Grid solution.

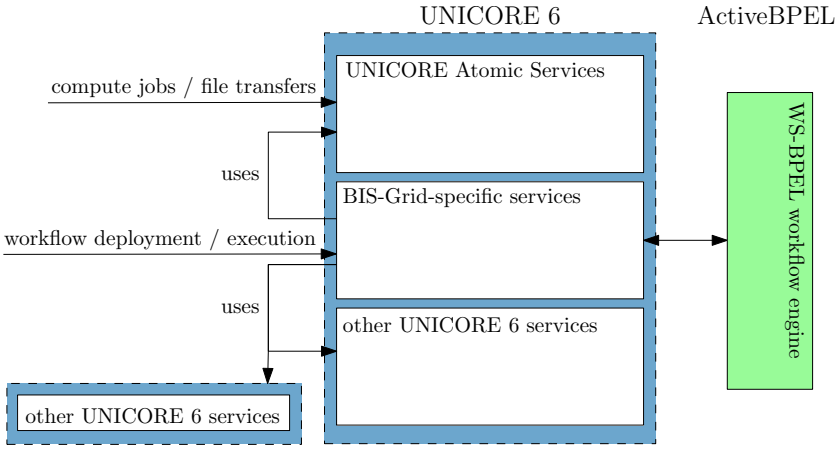

Figure 1: Overview on the Architecture of the BIS-Grid solution

Figure 1 presents an overview on the architecture of the BIS-Grid workflow engine. Within UNICORE/X, the BIS-Grid service extensions are placed beside so-called *UNICORE Atomic Services* which provide basic functionalities to support Grid computing,

[5] http://www.activevos.com/community-open-source.php

and beside other Grid Services that, e.g. may provide access to information systems. One important design decision was to neither extend the WS-BPEL standard nor to modify ActiveBPEL for Grid Service orchestration, although the WS-BPEL 2.0 specification provides an extensibility mechanism that allows to integrate additional functionality without declining the standard. However, the use of proprietary extensions would conclude in a solution that may not be interoperable with future versions of the standard as well as with the engine.

Leaving the WS-BPEL standard and the engine untouched ensures sustainability and flexibility, and allows to exchange the WS-BPEL engine by any other WS-BPEL engine. Figure 1 shows that the ActiveBPEL engine is located behind UNICORE 6. Hence, it can be deployed separately on backend nodes to support load balancing. In [4] we also present our considerations on load balancing the BIS-Grid solution.

Beside these advantages problems arise when using such a decoupled architecture without extending the WS-BPEL language. The WS-BPEL code that is necessary to call a Grid Service is far more complex than it would be if we had introduced new proprietary Grid-specific activities. For example, even if one wants to use a WSRF resource of a Grid Service for one single Grid Service invocation, one has to explicitly create, use, and destroy it using several WS-BPEL invoke activities. We address this by hiding the complexity of the code from the user as far as possible, especially from the workflow designer. To do so, we propose to extend an existing WS-BPEL editor by introducing new "Grid Activities" that encapsulate the WS-BPEL code for Grid Service invocation. Furthermore, there is the need to address some implementation-specific aspects such as a UNICORE 6/WS-BPEL process mapping problem. These aspects have to be addressed in the WS-BPEL code but are not part of the actual (fuctional) workflow. Therefore, we propose to inject these aspects automatically at workflow deployment, thereby concealing them from the workflow designer. In detail, these aspects are described in the BIS-Grid Specification [5] and in Deliverable 2.1 [6], which discusses these aspects in form of BPEL patterns.

1.3 Maven 2

Maven 2 (http://maven.apache.org/)is a software development tool, similar to build tools such as ant (http://ant.apache.org/). Such tools are designed to help developers to compile, test, and install their software during development. The advantage of using Maven 2 in contrast to other tools is the support of library dependency resolution by automatically downloading depending libraries in the correct version. This mechanism grealtly eases the management of development complexity. Since all libraries and the corresponding metadata are available for development in BIS-Grid, including UNICORE 6, it is guaranteed that we automatically receive library updates if, for example, UNICORE 6 takes a new version. The configuration of Maven 2 is done for each project separately in a so-called `pom.xml` file. It lists all dependencies to other projects and the needed versions, and the repositories where Maven 2 can find the corresponding libraries. For example, in our case we had to list the *WSRFlite* project on which UNICORE 6 is based and the *Unicore Atomic Services* project. Maven 2 then analyses the projects and

fetches the corresponding libraries recursively.

Maven2 has further advantages. There is a lot of plugins available that help us, for example, to create XMLBeans for WS-Resource Properties and messages, or to create Eclipse (http://www.eclipse.org/) projects with all necessary libraries included. Finally, we intend to provide releases of the BIS-Grid engine in a Maven 2 repository to facilitate dissemination. Other projects may then use the BIS-Grid engine easily.

1.4 Project/Package Structure

Basically the BIS-Grid engine consists of two mostly independent services: the Management Service and the Workflow Service. Therefore, both services are packaged as separate projects. They can use libraries that provide functions used by both services. We plan to develop new functions, other UNICORE 6 users can also use, e.g. the management of dynamic XACML [8] certificates for UNICORE 6 services.

name	description
`service.management`	Workflow Management Service including the Workflow Management Service Factory.
`service.workflow`	Workflow Service including the Workflow Factory Service.
`service.proxy`	The BIS-Grid proxy service used to catch messages from the BPEL engine and map them to the other BIS-Grid services.
`service.messages`	This project hosts the XML Schema descriptions for all messages used by the BIS-Grid services. For clearness, each service uses an own xsd-File. Furthermore the schema files for the Resource Properties are located in this package
`service.common`	package with common classes, e.g. WS-BPEL Workflow Engine Management, Exceptions, or WS-BPEL Adapters.

Table 1: Project Structure

The BIS-Grid engine therefore consists of 5 projects, shown in Table 1. Figure 2 shows the dependencies between the specific projects. The packages `service.messages` and `service.common` are completely independent. They serve as support packages for the other projects. The `service.proxy` package is used to import the proxy interface to receive messages from the proxy, and by some test classes. `service.management` and `service.workflow` both use the support packages for message handling and using common support classes.

After defining the project structure, we had to agree to a common package structure underlining the modularity of the components. We agreed upon `de.dgrid.bisgrid` as structure prefix to reflect the fact that BIS-Grid is a subproject of the D-Grid that again is a German initiative. The following structure name describes the component it belongs to, e.g. `service.management` for the Workflow Management Service. The then-following elements are arbitrary (sub-project-specific).

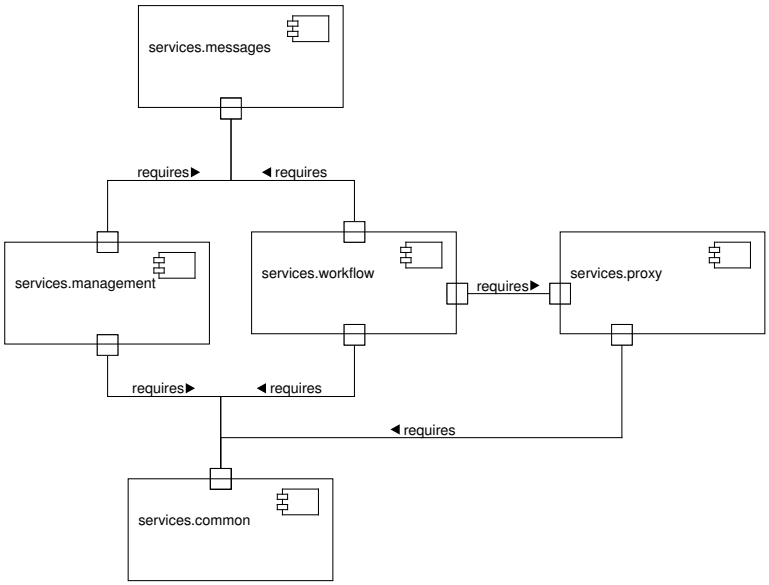

Figure 2: BIS-Grid Components and Dependencies

1.5 BIS-Grid Configuration

BIS-Grid provides configurable services. This is done with the help of the Java Properties concept. A `.properties` file and the corresponding `Properties` object implementation that reads the file is used. A `BISGridProperties` object, which is created when the services load, retrieves the configuration information. It is implemented as a Singleton [2]. This object is responsible for the configuration of all BIS-Grid services. The properties are loaded in two steps, first the default properties and then customised properties. The location where the customised, site-specific BIS-Grid configurations are stored must be set in the UNICORE Atomic Services properties file `uas.properties`. This can be found in the UNICORE 6 configuration folder "unicorex/conf/":

1. Read **default** from the BIS-Grid jars in the classpath (`bisgrid.properties.all`)
2. Read custom properties from file given in `UAS.properties` by property `bisgrid.properties.file`

WS-BPEL workflow engines must be described by the following four properties. For each engine, the `<id>`-part must be substituted with an ID for the engine, starting with 0 and incremented by 1 for each new engine.

- `BPEL_WORKFLOW_ENGINE_HOST_<id>`: The hostname of the computer, on which the WS-BPEL engine is running on (e.g. localhost, FQDN).
- `BPEL_WORKFLOW_ENGINE_PORT_<id>`: The port where the WS-BPEL engine is listening on (e.g. 8000, 8443).

- BPEL_WORKFLOW_ENGINE_SSL_<id>: Information on if the engine is using SSL description (TRUE/FALSE).
- BPEL_WORKFLOW_ENGINE_TYPE_<id>: The type of WS-BPEL engine (e.g. ActiveBPEL). The type is used to find the correct WS-BPEL engine adapter.

The BIS-Grid engine is capable to support new WS-BPEL engine adapters. These adapters must implement the adapter interfaces that are located in the package de.dgrid.bisgrid.commom.bpel.adpater and that must be available in the classpath at startup time of the UNICORE 6 environment. The Adapter interface is the central entry point to a WS-BPEL adapter. The services use this it to retrieve the respective specialised adapters, for example the MonitoringAdapter or the DeploymentAdapter. Hence, the configuration file needs to include information where to find the implementation of the adapter interface. With the help of the type information of the WS-BPEL engine, the class BisGridProperties searches for a property that is named <type>_Adapter. BIS-Grid is shipped with an adapter for the ActiveBPEL engine "ActiveBPEL_Adapter" by default.

- <type>_Adapter: The class of the implementation of interface Adapter for a WS-BPEL Engine Adapter[6].

The BIS-Grid service extensions can be configured with an own keystore to enable the invocation of external Grid Service. The location and configuration of this keystore is addressed by the following properties:

- BIS_GRID_Engine_Keystore_File: The location of the keystore
- BIS_GRID_Engine_Keystore_Password: The password for the keystore
- BIS_GRID_Engine_Keystore_Type: The type of the keystore (e.g. jks, pkcs12)
- BIS_GRID_Engine_Keystore_Alias: The name of the certificate alias used for client authentication
- BIS_GRID_Engine_Truststore_File: The location of the truststore
- BIS_GRID_Engine_Truststore_Password: The password of the truststore
- BIS_GRID_Engine_Truststore_Type: The type of the truststore (e.g. jks, pkcs12)

The following additional configurations are also possible:

- bisgrid.deployment.directory: The folder where the deployment package is stored.
- BPEL_SOAP_ACTION_URL: The URL used to mark and detect WS-BPEL operations with the help of SOAP Actions. (e.g. http://bisgrid.de/bpel-operation)

[6] e.g. de.dgrid.bisgrid.common.bpel.adapter.activebpel.ActiveBPELAdapter

- BIS_GRID_PROXY_PORT: The port of the proxy service that receives messages from WS-BPEL engines.

The following properties are currently needed for testing and debugging:

- TEST_BPEL_WORKFLOW_NAME: The workflow name as to be found in the WS-BPEL workflow engine with ID TEST_BPEL_WORKFLOW_ENGINE_ID.
- TEST_BPEL_WORKFLOW_ENGINE_ID: The ID of the WS-BPEL workflow engine to be used for test cases (e.g. 0 for the engine with id 0).

1.6 BIS-Grid Services Deployment

To integrate the BIS-Grid service extensions into a UNICORE 6 installation, and to start the then so-called BIS-Grid engine, one must perform the following steps:

- Install a UNICORE 6.
- Install a Tomcat and deploy the ActiveBPEL engine in it.
- Configure the Tomcat to use a Proxy. Add `http.proxyHost=localhost` and `http.proxyPort=8089` into the `catalina.properties`-file.
- Copy the BIS-Grid Jars into the `unicorex/lib`-Folder.
- Add the class: `de.dgrid.bisgrid.services.common.startup.BIS-Grid` to the startup code in the `UAS.properties` configuration file in `unicorex/conf`.
- Adjust the `bisgrid.properties` file and add the location into the `UAS.properties` configuration file with the property: `bisgrid.properties.file`.
- Start the Tomcat service with the ActiveBPEL Engine.
- Start the UNICORE/X container.

2 Workflow Management Service

As part of the BIS-Grid service extensions, the Workflow Management Service is responsible for deploying, redeploying and undeploying workflows to the BIS-Grid engine. In general, the Workflow Management Service consists of two separate services to be deployed within an existing UNICORE 6 installation, a factory service that creates Workflow Management Services instances, and the actual Workflow Management Services. Thereby, the factory service is realised as a plain (stateless) Web Service, and the Workflow Management Service is realised as a (stateful)WSRF Grid Service. After the creation of a new (empty) Workflow Management Service instance it can be used to deploy workflows upon receiving a so-called deployment package. Upon deployment, the Workflow Management Service instance creates an instance of the Workflow Service that can be regarded as a WSRF service proxy to the WS-BPEL workflow, and itself deploys the WS-BPEL workflow that is part of the deployment package in the WS-BPEL engine located behind UNICORE 6. After successful deployment, the Workflow Management Service instance is mapped to the corresponding workflow, and is responsible for further workflow management actions such as redeploying and undeploying. This section comprises implementation details of the Workflow Management Service. More documentation can be found in the Javadoc for the Workflow Management Service library.

In the current prototype of the BIS-Grid engine, only the deployment of workflows is realised. So far, the development goal was to provide a tracer bullet prototype that correctly and reliably performs the deployment process as the main example of the deployment mechanism. Since workflow undeployment and redeployment is very similar to deployment, they are expected to be added with low effort in very-near future. Also, additional management services such as retrieving lists of currently deployed and/or running workflows shall be implemented. Figure 3 illustrates the deployment process in general. This can be understood as a call of a `deploy` method that expects a BIS-Grid deployment package and responds a positive or negative result of deployment. Please note that the figure only shows the relevant activities of the process without further compensation and error handling. These activities are (1) the preparation of deployment, e.g. the initialisation of an empty response message and of temporary directories, (2) storing the incoming deployment package in a temporary directory, (3, 4) unpacking the deployment directory and filling internal properties with the package's contents, (5) inserting implementation-specific WS-BPEL patterns in the WS-BPEL workflow description[7] such as a UNICORE 6/WS-BPEL engine Mapping pattern and a pattern to propagate the IDs of WS-BPEL workflow instances as a prerequisite for higher monitoring services, (6) sending the relevant contents of the deployment package to a WS-BPEL engine adapter to deploy the actual WS-BPEL workflow in a WS-BPEL engine like ActiveBPEL, (7) creating a corresponding WorkflowServiceFactory instance and Workflow Service instance for the deployed WS-BPEL workflow, (8) registering these

[7] These WS-BPEL patterns are implementation-specific in such way that they usually shall not be modelled by the workflow designer but concealed from them since they do not represent process information that is relevant to the corresponding real-world/business process. The WS-BPEL workflow description is a part of the BIS-Grid deployment package.

services to UNICORE 6's UNICORE/X service container, and (9) filling and returning the response to the method caller indicating that the process was finished successfully. This includes endpoint references to the respective workflow services.

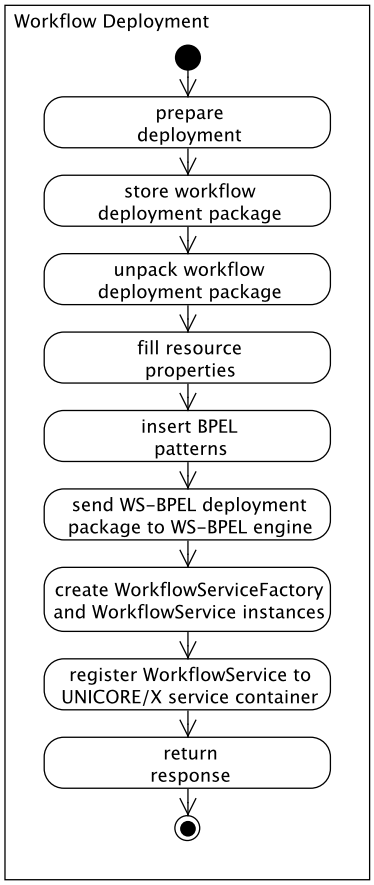

Figure 3: Activity Diagram - Workflow Deployment

2.1 Class Hierarchy

The factory service of the Workflow Management Service is realised as a plain Web Service, meaning that it is stateless. The class `WorkflowManagementServiceFactory` implements the interface `IWorkflowManagementServiceFactory`. The interface is marked to be a Web Service via the annotation `@WebService`. The service's methods that are accessible as Web Service methods that must also be annotated. Namely, these methods are `CreateWorkflowManagementServiceInstanceRespDocument` to create new instances of the Workflow Management Service and `SearchWorkflowServiceInstancesRespDocument` to search for existing Workflow Management Service instances.

The Workflow Management Service is realised as a stateful WSRF Grid Service. Its state is represented by so-called *resource properties*. The Workflow Management Service implements the interface `IWorkflowManagementService`, which inherits methods and variables from the `WSResource` interface that is a combination of three interfaces: `ResourceLifetime`, `WSRFInstance`, and `ResourceProperties`. The interface is marked to be a Web Service via the annotation `@WebService`. The service's methods that should be accessible as Web Service methods must also be annotated. The class hierarchy is presented in Figure 4.

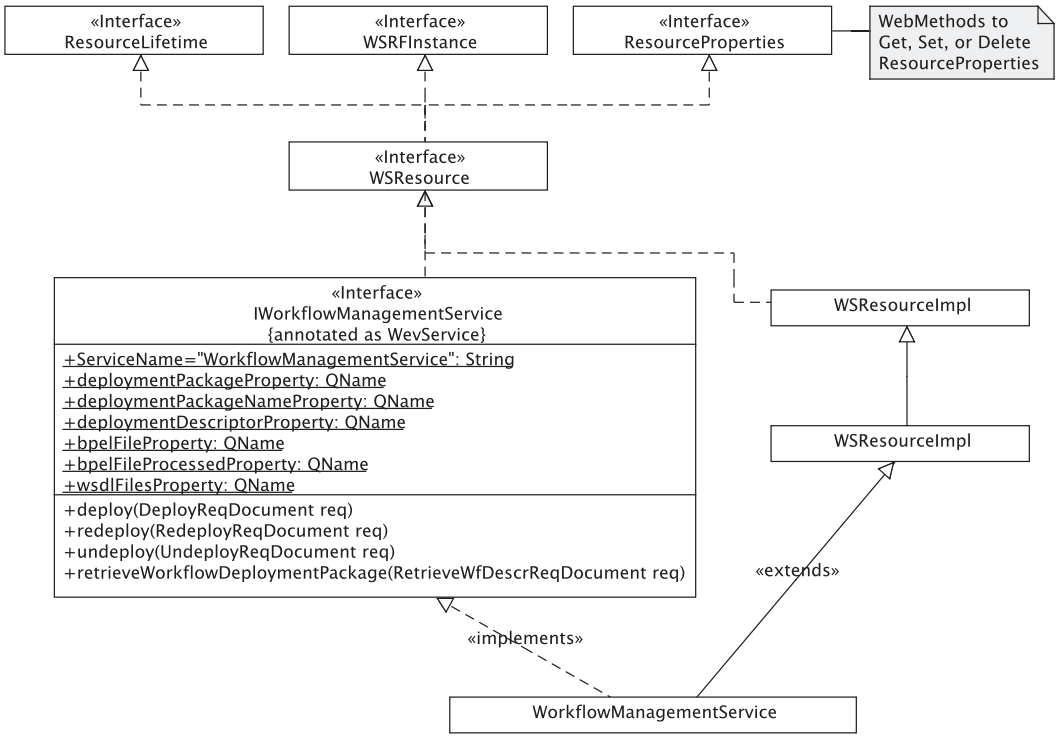

Figure 4: Class Diagram - Workflow Management Service

The interface defines the names of the WSRF resource properties that are used in the Workflow Management Service. The Workflow Management Service implements the interface `IWorkflowManagementService` and furthermore extends the class `WSResourceImpl`. By doing so, it inherits all functions defined in the WSRF specification, e.g. to handle resource properties.

2.2 BIS-Grid Deployment Package

The BIS-Grid deployment package is a ZIP archive that contains all necessary files to deploy one workflow. Currently, the following files are recognised by the Workflow Management Service.

- WS-BPEL process file (mandatory): One file with the file extension `.bpel` is expected that contains the WS-BPEL description of the workflow.
- WSDL files (optional): Several files with the file extension `.wsdl` may be included each containing a WSDL definition that either describes an external service interface used in the workflow or the workflow service interface itself. These WSDL definitions must not be included in the deployment archive itself. It is also possible to reference external WSDL locations in the BIS-Grid deployment descriptor which are fetched at deployment time.
- BIS-Grid deployment descriptor file (mandatory): One file with the file extension `.bdd` is expected that inter alia contains binding information about partner links and references to (external) WSDL and XML schema locations. It is a generic and vendor-independent deployment descriptor which has to be transformed to a vendor-specific deployment descriptor like ActiveBPEL during the deployment process. Currently, the deployment descriptor's structure is proprietary and close to the ActiveBPEL deployment descriptor. Within a diploma thesis a generic deployment descriptor will be developed considering existing deployment descriptors of several vendors.

In the next implementation phase the BIS-Grid deployment package will be extended regarding additional security issues as a fine-grained role-based access control (RBAC) for each workflow method based on XACML policies (and SAML assertions for roles) that can be partially modified during runtime. For this purpose a Process Security Policy file with the file extension `.psp` will be introduced. Furthermore, credentials needed for external service invocations can be configured in the Process Security Policy as username/password, an X.509 certificate (including the corresponding private key) or a proxy certificate. These credentials may be included directly in the Process Security Policy but due to security reason they can also be provided by the workflow user at workflow instance creation time. For more information please refer to the BIS-Grid workflow engine's specification[5].

2.3 BPEL Pattern Injection

As part of the workflow deployment process for the BIS-Grid engine, one step is to insert implementation-specific WS-BPEL patterns into the WS-BPEL workflow description. This is done by inserting the pattern skeletons via XSL Transformations (XSLT), and adding the workflow-specific information afterwards using conventional XML processing based on the JDOM library [7]. Also, there is the need to adapt some other XML files

that are involved in the workflow deployment process, such as the BIS-Grid deployment descriptor.

Currently, the XML processing mechanism is located in the subpackage `xmlprocessing`. Figure 5 illustrates the class structure of the most important classes of the package. The main interface is `IXmlProcessingFacade`, representing the facade interface that offers all functionalities of the package. The methods provided by the interface are `insertGridBpelMapping(...)` and `insertProcessIdRetrieval(...)` in different variants. These two methods represent the insertion of two different WS-BPEL patterns, a pattern to enable the mapping of UNICORE 6 service instances to WS-BPEL workflow instances located in the WS-BPEL engine, and a pattern to propagate the ID of WS-BPEL workflow instances to enable higher monitoring services. The interface is implemented by the actual facade class, `XmlProcessingFacade`. This class is realised as a Singleton [2], meaning that the whole functionality of the package is only accessible via one single facade object.

For the implementation of the pattern insertion methods, the facade makes use of the so-called `XsltBasedBpelProcessor` to insert the pattern skeletons into the original WS-BPEL file. After that the inserted skeletons are filled with the necessary service instance information, if necessary. To insert the skeletons, the `XsltBasedBpelProcessor` itself utilises XSLT files that describe the individual transformation steps of the WS-BPEL file. Thereby, we follow the convention that each WS-BPEL pattern is represented by one single XSLT file. Namely, the developed and currently used XSLT files are *bpel-grid-transformation*, *process-id-retrieval-transformation*, *active_bpel_deployment_descriptor_transformation*, and *active_bpel_catalog_transformation*. Further details on the WS-BPEL patterns and the content of the XSL Tansformations files can be found in Deliverable 2.1 [6], the BIS-Grid WS-BPEL pattern catalogue.

For some of the patterns, it is necessary to insert workflow-specific information using conventional JDOM-based XML processing after the XSL Transformations (XSLT). One case is a pattern that inserts a UNICORE 6/WS-BPEL mapping into a WS-BPEL workflow description (cp. Section 5.1 in [6]. The first part of the patern insertion is to apply the transformation *bpel-grid-transformation* . After the transformation, information on a particular Workflow Service that shall be mapped to the WS-BPEL workflow and that is located in a UNICORE/X service container must be inserted in the pattern skeleton. This information is represented by the class `UcServiceEndpointReference` which also represents its own Factory [2]. The class is statically initialised holding the empty service information `NO_SERVICEINFO`, and new `UcServiceEndpointReference` objects can be retrieved via `getUcServiceInformation`. The method retrieves (if in existence) or creates (if not) a new `UcServiceEndpointReference` objects for the supplied parameters. It is possible to add validity checks to the method implementation that check the input parameters. Since the class is final, it cannot be subclassed. This implies that it is not possible to introduce anomalous checking behaviour by providing an own `getUcServiceInformation` method. Currently, `UcServiceEndpointReference` is an input parameter of the facade class methods `insertGridBpelMapping`. All in all, the implementation of the class `UcServiceEndpointReference` is similar to the implemen-

tation of `org.jdom.Namespace` of the JDOM library.

At the current state of development, the pattern injection mechanism relies on XSLT processing that inserts pattern skeletons in the respective WS-BPEL files, and filling the skeletons afterwards. Since the XSLT processing is general in such way that it is only depending on the XSLT files to be used for transformation and the input XML files to be processed, and since it is also necessary to adapt other XML files such as the deployment descriptor, we plan to move the XSLT processing to the package `service.common`.

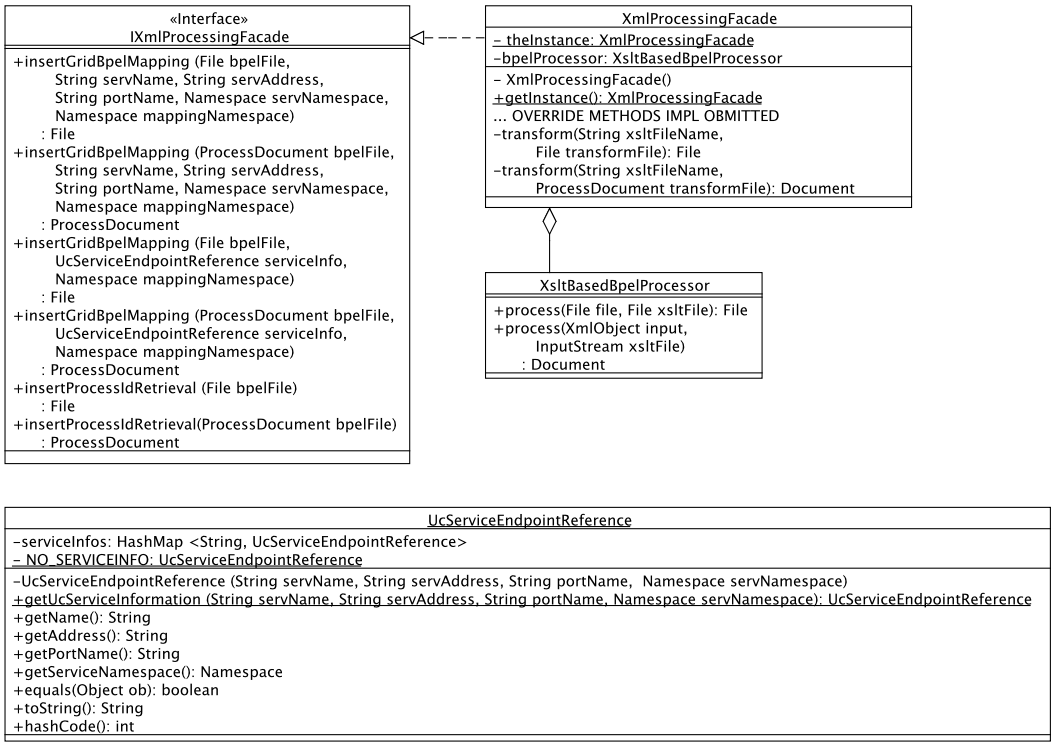

Figure 5: Class Diagram - Workflow Management Service Pattern Injection

2.4 Hot Deployment

As described in the specification of the BIS-Grid engine [5], instances of the Workflow Service and its corresponding Workflow Factory Service must be deployed when a workflow is deployed within the BIS-Grid engine. Each set of the two services represents one deployed workflow or one deployed BIS-Grid deployment package, respectively, and correspond to the actual workflow that is deployed in the WS-BPEL engine that is part of the whole BIS-Grid engine. The design of the deployment mechanism must meet the requirement that the UNICORE 6 must not be restarted upon the deployment of

WSRF Grid Service instances such as instances of the Workflow Management Service, and that the user must not reconfigure the UNICORE 6 service container directly. The following properties are extracted from the specification [5] and holds for the Workflow Management Service:

- Workflow Factory Service
  - One service per deployed workflow package.
  - Must know the so-called *Service Home*[8] of the corresponding Workflow Service.
- Workflow Service
  - One service per deployed workflow package.
  - One instance per currently running workflow.
  - Must propagate the operations of the WS-BPEL workflow.
  - WS-BPEL workflow operations are not known during development time.

We developed a generic Grid Service for workflow execution that performs an automatic online configuration. The implementation of this service is described in Section 3. Here only the mechanisms to deploy a new Workflow Factory Service and Workflow Service is described. The first idea was to use a simple Java interface for hot deployable services that should be implemented by the Workflow Factory Service as a plain Web Service. After service creation, we would retrieve the service instance from the `Kernel` and cast it to this interface. Finally, we could configure the service by using this interface. Unfortunately, this idea failed. The problem is that a plain Web Service is no real Java object in UNICORE 6's `XFire` component[9]. So, Java Reflection is used to call the operations on the service class directly, not on a service object. Without having an object, it is impossible to configure the Workflow Factory Service so that it creates instances of the corresponding Workflow Service.

Instead of that, we realised the configuration of the Workflow Factory Service by using a WSRF-conform factory service that consists of stateful Workflow Factory Service instances. This service is deployed on startup time together with the Workflow Management Service. Each newly-created workflow instance is configured by it's resource properties. Normally, a WSRF service has a factory, but in this case the WSRF Workflow Factory Service consists of only a `Service Home` class that manages the factory instances. New Workflow Factory Service instances can only be created through an direct Java call on this Service Home inside UNICORE 6's virtual machine. This method is named `createWSRFServiceInstance(imap)` where `imap` is a `HashMap` containing arbitrary initialisation parameters.

The deployment package (send by the Workflow Designer to the Workflow Management Service) includes information that is necessary for hot deployment but also for

[8] The Service Home is responsible to load the resources that are attached to a service.

[9] The SOAP processing framework XFire (xfire.codehaus.org) is an internal component of the original UNICORE 6

the creation of the workflow instances. Table 2 gives an overview about the information included in the `imap` variable that initialises the Workflow Service. The following information originates from the deployment package.

key	value description
de.fzj.unicore.wsrflite.uniqueid	The unique id of the factory e.g. CallCenterWorkflowFactory.
WorkflowServiceName	Name of the workflow service. Must be the same name as used to deploy the workflow service.
BPELWorkflowName	Name of the workflow as it is deployed in the WS-BPEL engine.
ServiceCallDescriptions	Array of `ServiceCallDescriptionDocuments` that describes the BIS-Grid configuration for the external service calls of the workflow.

Table 2: Initialisation parameters for the Workflow Service

When calling the method `createWSRFServiceInstance`, the new factory instance is available as a WSRF service instance of the WSRF factory service identified by an unique ID, which is equal to the Workflow Service name with the suffix "Factory", e.g. "CallCenterWorkflowFactory" if the Workflow Service's name is "CallCenterWorkflow". Future work may require to adapt the ID-naming to include more diverse information in order to sustain uniqueness.

When the hot deployment process is finished, the user can access the factory service at the address `<site url>/BIS_Grid_Workflow_Factories?res=<uniqueid>`[10]. The Workflow Service can be found at the address `<site url>/<WorkflowName>`[11].

[10] e.g. `https://bisgrid.de:8080/BISGRID-SITE/BIS_Grid_Workflow_Factories ?res=CallCenterWorkflowFactory`

[11] e.g. `https://bisgrid.de:8080/BISGRID-SITE/CallCenterWorkflow`

3 Workflow Service

This section describes the implementation of the Workflow Service. First an overview about how the Workflow Service fits into the UNICORE 6 implementation. After that we describe the most important modifications that are necessary to support the special requirements. This includes he combination of the WSDL, the modular adapter concept to support different WS-BPEL engines as well as changes in the HandlerPipeline for the Workflow Service. Then we describe how the Workflow Service is used to modify and forward calls to external UNICORE 6 services but also to external web services. This can be found in Section 3.6.

3.1 Class Hierarchy

The Workflow Service implements the `IWorkflowService`-Interface. It inherits methods and variables from the `WSResource` Interface that is a combination of three interfaces: `ResourceLifetime`, `WSRFInstance`, and `ResourceProperties`. The interface is annotated as WebService. Methods that should be accessible as WebMethod must also be annotated. The class hierarchy is depicted in Figure 6. The interface itself defines the

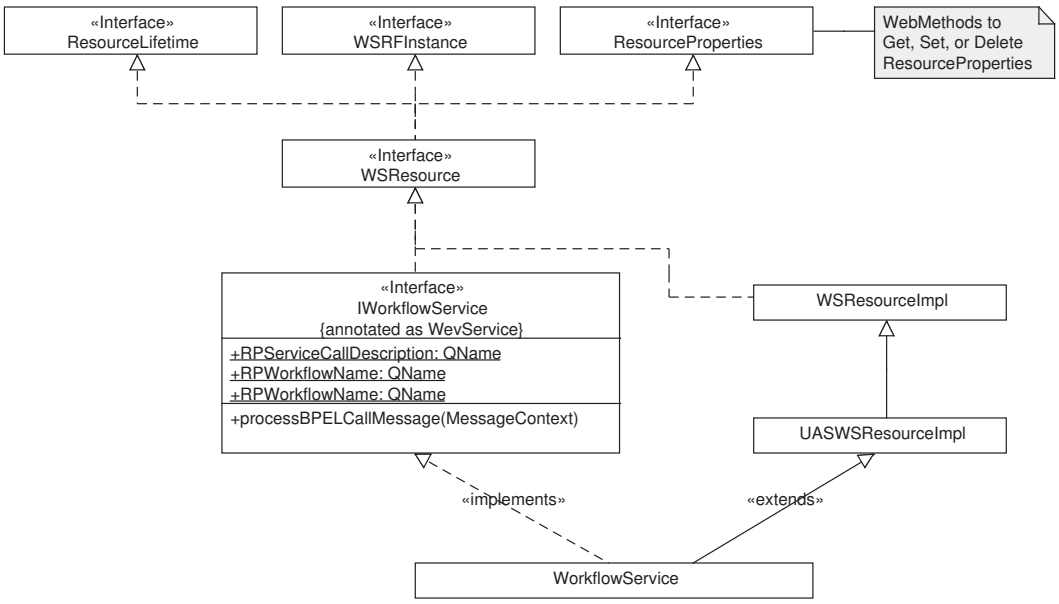

Figure 6: Class Diagram - Workflow Service

names of the WSRF resource properties that are used in the Workflow Service. Furthermore the `processBPELCallMessage`-Method is defined as a non Web-Method. It is used by the BisGridInvoker to submit WS-BPEL Call messages to the Workflow Service. The Workflow Service implements the interface `IWorkflowService` and furthermore extends

the class `WSResourceImpl`. By doing so, it inherits all functions defined in the WSRF specification, e.g. to handle resource properties.

3.2 WSDLWriter

Each UNICORE 6 service has a `WSDLWriter` that is responsible for generating its WSDL and therefore it defines the interface described in the WSDL and used by clients. The BIS-Grid Workflow Service offers a WSDL that is a combination of the regular interface for the operations directly implemented via annotated methods in the `IWorkflowService` Java interface plus additional methods that are necessary to execute the workflow and actually offered by the WS-BPEL workflow engine. The latter part is highly dynamic and depends on the deployed workflow. It can not be generated during development time. This causes a complex substitution of the regular WSDLWriter.

To create such a combined WSDL, we developed a new `WSDLWriter`. It is named `CombiningWSDLWriter`. It uses the original WSDLWriter that generates the WSDL for the regular service operations. In a second step, the combination, it adds the new parts for the workflow operations into the existing WSDL definition. This is done by integrating information from the workflow WSDL: types, imports, namespaces, bindings, porttypes, and messages. This information are integrated in the WSDL as follows: The service gets a new port called `WorkflowServiceBpelPort` that points to a new binding, the `BpelBinding` (depicted in Listing 1).

Listing 1: Example Service Port for WS-BPEL Operations

```
<wsdl:port
  name="WorkflowServiceBpelPort" binding="tns:BpelBinding">
  <wsdlsoap:address location=".../
     CallCenterWorkflowWorkflowService"/>
</wsdl:port>
```

The `BpelBinding` is of type `BpelHttpPort`. The operations in this binding gets a newly generated SOAP-Action-Tag, because ActiveBPEL does not support SOAP Actions in their generated WSDLs. All WS-BPEL operations SOAP-Actions start with `http://bisgrid.dgrid.de/bpel-operation` (configureable in the BIS-Grid properties) and are concatenated with the operation name. As SOAP style "document/literal" is used (same as for the regular operations). An example for a generated binding with one operation named "call" can be found in Listing 2:

Listing 2: Example BpelBinding

```
<wsdl:binding name="BpelBinding" type="tns:BpelHttpPort">
  <wsdlsoap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="call">
    <wsdlsoap:operation
    soapAction="http://bisgrid.dgrid.de/bpel-operation/call"
    style="document"/>
    <wsdl:input>
```

```
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

The corresponding abstract port type look like shown in Listing 3:

Listing 3: Example BPELHttpPort

```
<wsdl:portType name="BpelHttpPort">
  <wsdl:operation name="call">
    <wsdl:input message="tns:callWorkflowRequest">
    </wsdl:input>
    <wsdl:output message="tns:callWorkflowResponse">
    </wsdl:output>
  </wsdl:operation>
</wsdl:portType>
```

Messages and schema information for the messages are simply copied from the WS-BPEL Workflow WSDL into the Workflow Service WSDL. The new WSDL definition is used in the `BISGridServiceSetterHandler` to fake the not really existing implementation of the WS-BPEL-operations. More about this can be found in Section 3.4.2 and in the Specification [5].

3.3 Adding a new WS-BPEL-Engine-Adapter

The BIS-Grid engine is designed flexible to support new Adapter to new WS-BPEL Workflow engines without changing the code of the BIS-Grid services. Figure 7 a class diagram about the Adapter framework implementation.

The only entry point to the different adapters is the `AdapterFactory`. If a BIS-Grid module needs non-standard WS-BPEL Workflow engine functions, it has to use the factory to get the matching adapter to the current WS-BPEL Workflow engine. The factory uses the `BISGridProperties` to find the correct `Adapter` implementation using the type of the engine, uses the class loader to load the adapter dynamically and returns it back to the calling object. If specialized operations are needed, the `Adapter` implementation can be used to get implementation of special sub-interfaces, like an implementation of `DeploymentAdapter` or `MonitoringAdapter`. If one wants to support a new WS-BPEL Workflow engine, he has to do the following steps:

- Implement the Adapter Interfaces and all Sub-Interfaces (for the calls it is possible to use generated stubs).
- Copy the Jar with the implementation into the `unicorex/lib` folder of the UNICORE 6 installation.

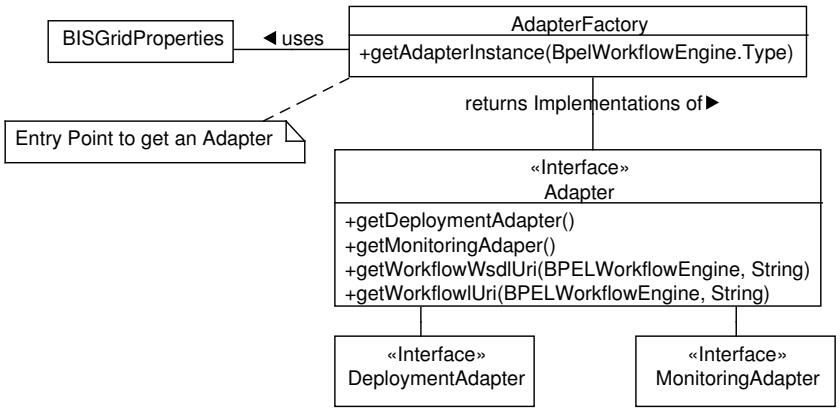

Figure 7: Class Diagram - Adapter-Concept

- Add the full class name (including package) to the `bisgrid.properties`-file.
- Add the new BPELEngine Configuration to the `bisgrid.properties`-file.

3.4 Handler Pipeline

Handler Pipelines are a very important concept in each Web Service Framework. They are used to analyse and process SOAP messages before and after the actual web service call. This subsection describes the standard pipeline used for the UNICORE ATOMIC SERVICES and how we modify this pipeline for our BIS-Grid purposes.

3.4.1 UNICORE 6 Standard Handler Pipeline

Normally, the Handler Pipeline for UNICORE 6 services can be configured in the `wsrf.xml` configuration file, where also the services are deployed. Some Handlers are fixed in the handler pipeline by deploying a service as WSRFLite service or using an service implementation the inherits from `UASWSResourceImpl`. Further Handlers can be added using the service configuration. The standard pipeline looks like shown in the Figure 8.

Each handler realizes a small part of the SOAP message processing. Thereby, some handlers need other handler invoked in advance. Some handler are more complex as listed below, but they generally realizes the following functions:

- **StartTimeHandler**: Stores the start time in the `MessageContext` (for message processing analysis).
- **ReadHeadersHandler**: Reads the SOAP Message until the `Body` tag is reached. Stores the Namespaces in the `MessageContext`. Stores the SOAP Header in the

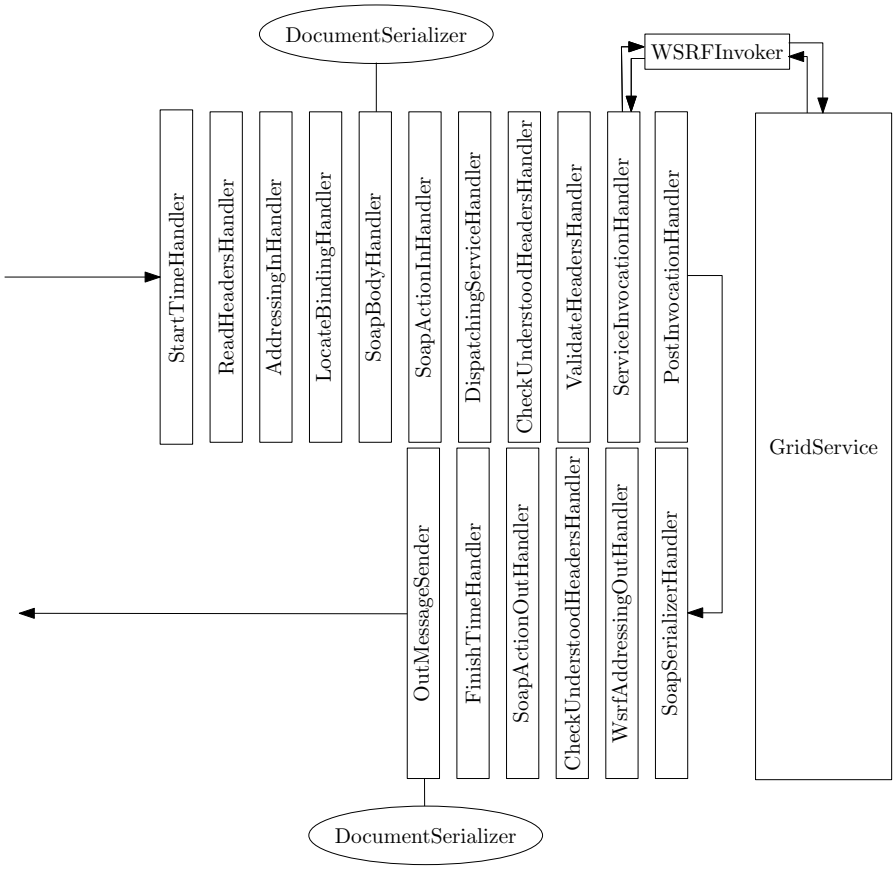

Figure 8: Grid Service Handler Pipeline

in message's header variable. Checks if the message is a fault message, reads the fault and throws an XFire Exception.

- **AddressingInHandler**: Uses the header from the message's header variable. Processes the WS-Addressing header, if included in the SOAP header.
- **LocateBindingHandler**: Finds the appropriate binding to use when invoking a service. This is delegated to the transport via the `findBinding` method. Sets the Binding in the textttMessageContext. Normally, this will be a Soap11Binding or Soap12Binding.
- **SoapBodyHandler**: Uses the Binding set by the LocateBindingHandler. Gets the MessageSerializer by using this Binding and operation information. Deserializes the messages's body.
- **SoapActionInHandler**: If there is no WS-Addressing information, there is no service and operation set till now. This handler inspects the SOAPAction from the HTTP message header and selects the appropriate operation.

- **DispatchingServiceHandler**: Prepares the in hander pipeline by adding handlers additionally inserted by the service. If it is an operation with output, it prepares the complete out handler chain using service, xfire, and transport handlers.
- **CheckUnderstoodHeadersHandler**: An FZJ hack that adds the information to the handler pipeline that this pipeline understands e.g. WS-Addressing namespace. This information is used in the `ValidateHeadersHandler`.
- **ValidateHeadersHandler**: Evaluates the different "must understood" namespaces in the SOAP header. Therefore it asks each handler in the pipelines whether it understands a namespace.
- **ServiceInvocationHandler**: Creates the parameter objects and uses the invoker to process the Java call on the service object. Calls are done using Threads. Sets the result for the PostInvocation handler in the `MessageContext`.
- **PostInvocationHandler**: This is the last handler in the incoming pipeline. Sets some information, e.g. the result as message body, in the OutMessage, if necessary. Invokes the out pipeline.
- **SoapSerializerHandler**: Uses the message serializer of the `OutMessage` in combination with a `SoapSerializer`. This creates a new serializer for the `OutMessage` that generates the SOAP Envelope, Header, and Body tags. The out message's body object represents only the content in the body tag of the outgoing soap message (without the body tags).
- **WsrfAddressingOutHandler**: A specialization of the AddressingOutHandler. Repairs a bug of the upper class if the sender is a client. n case of a service outgoing pipeline, the `AddressingOutHandler` invoke operation is called.
- **SoapActionOutHandler**: Creates the WS-Addressing headers using the incoming WS-Addressing header information and server side information.
- **FinishTimeHandler**: Calculates the duration of the call and sends the result to the KernelAdmin class for statistical issues.
- **OutMessageSender**: Sends the message over the HTTP channel set in the `OutMessage`.

3.4.2 BIS-Grid Workflow Service Handler Pipeline

For BIS-Grid we need a customized handler pipeline, because there are several implementation specific problems we have to deal with. This is for example the combination of the WSDL and the call of a common method to process calls to the WS-BPEL engine. This is already described in the specification [5].

The Workflow Management Service deploys the Workflow Service and also does the following configuration for the handler pipeline. The purposes of the new handlers are described further below.

- Substitute **AddressingInHandler** by **BISGridAddressingInHandler**.
- Substitute **PostInvocationHandler** by **BISGridPostInvocationHandler**.
- Add new **BISGridServiceSetterHandler**.
- Add new **DOMInHandler**.

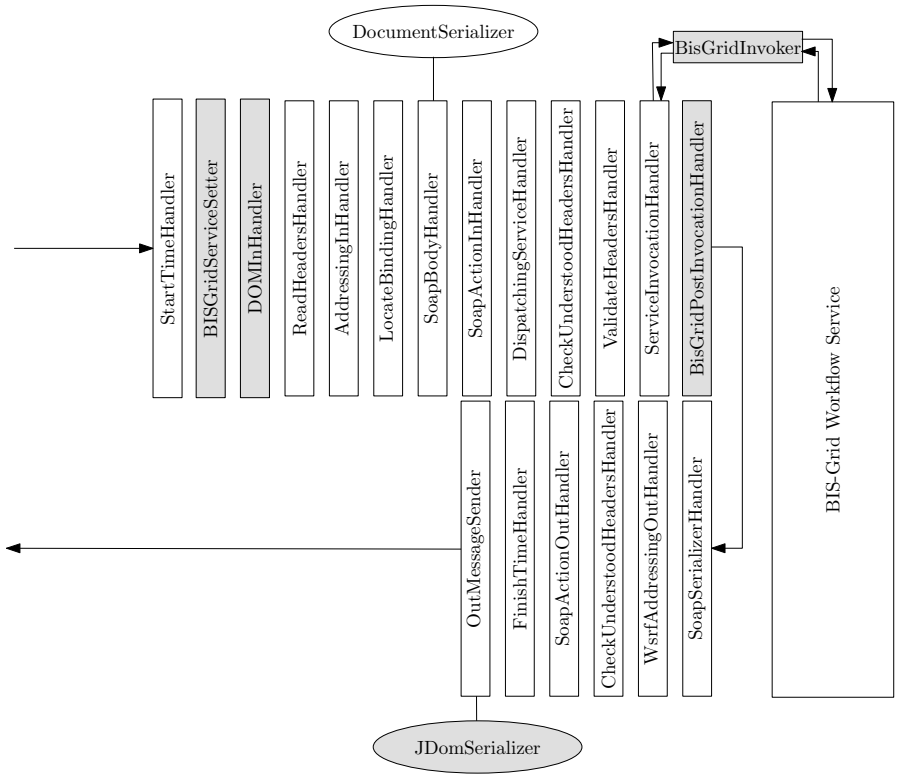

Figure 9: BIS-Grid Workflow Service Handler Pipeline

Furthermore, we change the Invoker, the object that does the Java reflection calls on the Workflow Service instances by a new `BisGridInvoker`. This results in the new Handler Pipeline for the BIS-Grid Workflow Services as shown in Figure 9. The changes are marked with a grey background.

The `BISGridServiceSetterHandler` checks the SOAP Action information from the HTTP-Header and sets a flag ("isBPELCall"-flag) in the properties of the incoming message if it starts with the configured WS-BPEL action string: `|http://bisgrid.dgrid.de-/bpel-operation`. Furthermore it uses the WSDL definition from the CombingWSDL-

Writer to fake a service description that provides all information for the following handlers.

The `BISGridPostInvocationHandler` and the `BISGridInvoker` use the "isBPELCall"-flag to distinguish between standard calls and WS-BPEL calls. The `DOMInHandler` is needed to save the whole message in the `MessageContext` object. This `MessageContext` is passed to the common "processBPELCall"-Method that processes all WS-BPEL calls (cp. Figure 6 at page 20).

The `DocumentSerialiser` does not work properly in case of a WS-BPEL operation. For outgoing WS-BPEL call messages, we havte to change the serializer to a `JDOMSerialser` because we get the answer message represented as JDOM in the OutMessageObject. This is done by the WorkflowService instance itself in the "processBPELCall"-Method.

3.5 Workflow Service State

At the moment, each Workflow Service only has a small amount of state information, but it will grow when we implement further features. We have to distinguish between mutable and immutable state information. Immutable state information only serves as information source for the user or other services, mutable state information can also be used for configuring the Workflow Service. Changes in the mutable ResourceProperties can be done by using service calls specified in WS-Resource-Properties specification [3].

3.5.1 Immutable Resource Properties

The following list gives an overview about the immutable Resource Properties and their intentions.

- **BpelEngineAddress:** The address of the WS-BPEL Engine in the Backend. So the user is informed what workflow engine is used for workflow execution. This creates a bigger transparency for the user.
- **WorkflowName:** The name of the workflow as it is deployed in the WS-BPEL Workflow Engine. So the user can control whether the correct workflow will be executed.

Both properties should help the user to trust in the BIS-Grid engine.

3.5.2 Mutable Resource Properties

A user can use Mutable Resource Properties, beside the in the interface regularly offered operations, to modify the state of the Workflow Service instance. One mutable state is the configuration of the external service calls. This configuration is realised as one Resource Property that is a list of service call configurations. If there is no extra configuration given, the Workflow Service will use a standard configuration for trying to call the external service. At the moment the configuration state of an external service call comprises the following:

- **ServiceAddress:** The external service address. It can be used as identifier for finding the correct description.
- **ServiceType:** The type of this external service. This must be one of the following values: `WebService` or `UNICORE6`.
- **CredentialDelegation:** Id of an previously inserted SAML Assertion [9] Credential that can be used for Trust Delegation.
- **UserNamePasswordCredentials:** A complex XML type to specify the username and password to provide simple credentials for an external WebService call.

3.5.3 Workflow Factory Service State

As already described in Section 2.4 the Workflow Factory Service is also implemented as a WSRF Service that instances are created by a direct Java call of the Workflow Management Service. Each factory also has a state, that determines the **ServiceHome-name** of the corresponding Workflow Service and the name of the corresponding **BPEL workflow name** deployed in the WS-BPEL engine. Both ResourceProperties are immutable properties.

3.6 External Service Calls

If the WS-BPEL engine calls an external service, the message is intercepted by the proxy (see Section 4). The Workflow Service implements the `WFCallReceiver` interface and delegates the message directly to the `BPELEngine2ExternalServiceProcessor`, that is responsible for the message processing (cp. Figure 10).

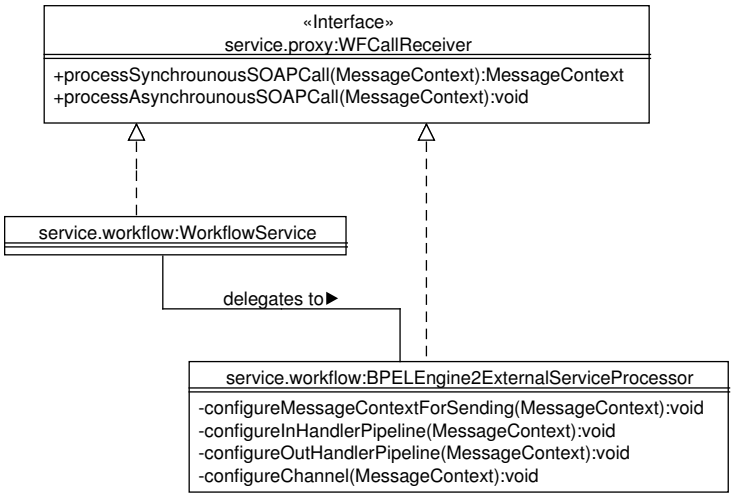

Figure 10: Class diagram - External service call processing

This class uses the information from the BIS-Grid deployment package to configure the handler pipelines for the external call. After that, it also configures the transport channel and finally send the message to the external service. The configuration consists of the steps *message context preparation*, *transport channel configuration*, *outgoing handler pipeline configuration*, and *incoming handler pipeline configuration*, but is not yet implemented.

4 Proxy Service

As described in Section 1.4 the proxy receives messages from the ActiveBPEL engine and forwards the requests to the Workflow Service. As the proxy is the "glue component" to let the ActiveBPEL engine interact with the UNICORE 6 workflow service, ActiveBPEL has to be aware of the existence of the proxy service. This is done in the Tomcat environment, where the ActiveBPEL engine runs in. Therefore Tomcat is configured to use an instance of the workflow proxy service as HTTP proxy for outgoing HTTP requests. This is done by adding `http.proxyHost=<IP-Adress | hostname> e.g. localhost` and `http.proxyPort=<port> e.g. 8080` into the `catalina.properties`-file.

Since WS-BPEL method calls are SOAP messages and these are HTTP requests (on a lower level of abstraction) these method invocations are redirected to the workflow proxy. After receiving the HTTP request the work carried out by the proxy is to decide if the HTTP request is a SOAP message and if this SOAP message is part of a workflow. If the received request is a WS-BPEL method call the request is intercepted and forwarded to the Workflow Service. The workflow instance to which the request is forwarded is selected based on the contents of the workflow ID (WWID) tag that can be found at the position defined by the following XPATH expression `Envelope/Header/WWID`. After delegation of the request the original request is acknowledged.

Besides the interception of WS-BPEL calls the proxy behaves like an ordinary HTTP proxy for all other HTTP requests. It forwards the request to the original URL and passes the response to the calling WS-BPEL Engine. This is necessary if the WS-BPEL Engine requests any other HTTP resource e.g. a WSDL file. The workflow proxy is managed by a proxy service which is a WSRF Service. The proxy itself uses a new instance of the Jetty servlet container per active proxy session and a special servlet implementation. The processing of a request by the proxy is shown in Figure 11. The structure of the proxy module is shown in Figure 12.

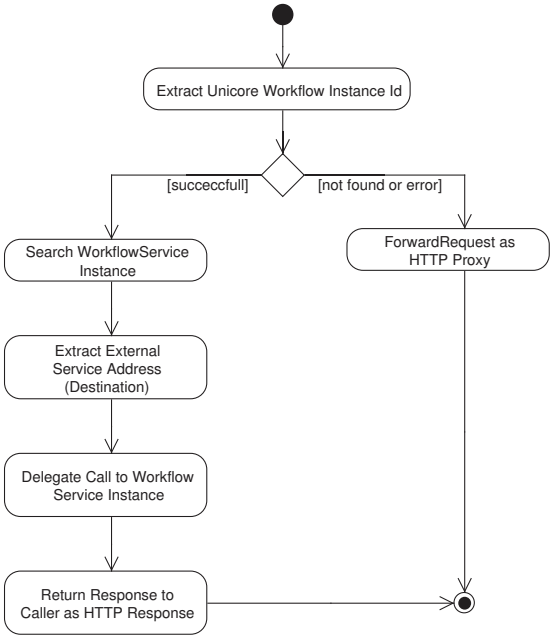

Figure 11: Activity Diagramm describing the processing of a HTTP request by the Workflow Proxy

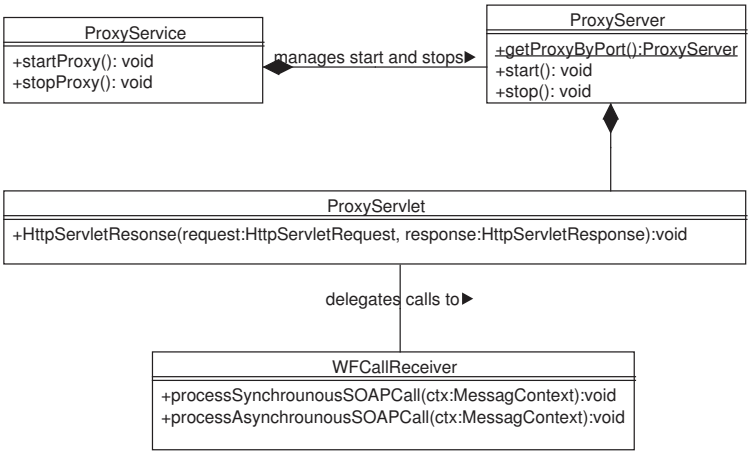

Figure 12: Class Diagramm describing the Workflow Proxy

5 Future Work

This document is part of Deliverable 3.2 and represents the documentation of the BIS-Grid engine prototype. The document has to be regarded as a documentation of ongoing work. It is intended to serve as basis for the future official documentation of the then-released BIS-Grid middleware final. In the following, we give a short overview on our near-future work on the BIS-Grid engine as far as conceivable:

- Implement undeployment and redeployment of workflows, as well as other management services such as retrieving a list of currently deployed workflows. This is discussed in the introductory part of of Section 2.
- Extend the BIS-Grid deployment package by additional security issues such as a fine-grained role-based access control (RBAC) mechanism for each workflow method based on XACML policies. This is discussed in Section 2.2.
- Generalise and move the XSLT-based XML processing from the package `service.management` to the package `service.common`. This is discussed in Section 2.3.
- Implement a variable configuration framework to configure external service calls (UNICORE 6 Services and Web Services).
- Implement authorisation mechanisms for WS-BPEL operations that can be configured by the XACML policies provided by the BIS-Grid deployment package, and that is based not only on certificates but only on RBAC.

Glossary

ActiveBPEL ActiveBPEL is a BPEL engine designed by ActiveEndpoints and is used by BIS-Grid to control the workflow execution.

BIS Business Information Systems

BPEL workflow engine The BPEL workflow engine is an engine providing the execution of workflows described in BPEL. In our case, the BPEL workflow engine used in the context of BIS-Grid is ActiveBPEL.

EAI Enterprise Application Integration

Enterprise Grid The adoption of Grid concepts and technologies within an enterprise, usually used to improve the scalability of a single enterprise's it-infrastructure and to optimize its resource utilisation (workload balancing). Enterprise Grids have properties that do not play a significant role in traditional scientific data processing, for example the dependable traceability of performed actions, the centralised control of distributed applications, support of non-batch processing types, the contractual guarantee of qualities of services, and the need to consider legacy systems.

Enterprise Service Bus An Enterprise Service Bus is an architecture to enable the communication and service mapping mechanisms between different services within a single context, for example an enterprise.

ESB see *Enterprise Service Bus*

FZJ Forschungszentrum Jülich

Grid Service A Grid Service is an extended version of a Web Service. The main difference is the fact, that a Grid Service, in contrary to a Web Service, can have a state. This state can be stored as resource properties and the user can request the current values. Each Grid Service offers a functionality, but it does not necessarily return the same answer, because of possible different states. A Grid Services describes its interface in WSDL.

Handler Handlers are used to process a SOAP message. They are ordered in a so-called handler chain (ingoing or outgoing handler chain). Each handler receives the so-called message context and may modify it. After that, the message is forwarded to the next handler in the chain.

Service Home The Service Home is a concept introduced by UNICORE 6. Each WSRF service has one Service Home, that is responsible to manage the instances of the service. When deploying a WSRF service in UNICORE 6, the class name of the Service Home must be given instead of the implementation of the service

SOA Service Orientated Architecture

SOAP SOAP is a stateless, one-way network protocol designed for the exchange of XML-based messages over computer networks, whereas more complex interaction patterns can be created by combining such one-way exchanges with an underlying protocol's features and/or application-specific information. The SOAP protocol contains several types of message exchange patterns, with remote procedure calls being the most common. SOAP does not specify the semantics of application-specific data it conveys, nor issues such as SOAP message routing, reliable data transfer, and firewall traversal. SOAP makes use of XML for representing data, and other protocols of the transport and application layer for message transport. The SOAP specification is a W3C recommendation.

UNICORE 6 A WSRF-conform middleware. See `http://www.unicore.org`.

Virtual Organization In Grid computing, a group of individuals or institutions who share the computing resources of a Grid for a common goal.

Workflow Service The Workflow Service represents one workflow in the Grid layer of the BIS-Grid engine. It realises all features needed to enable Grid Service orchestration with the help of a standard BPEL engine, thereby hiding the underlying WS-BPEL engine completely.

Workflow Service Factory The BIS-Grid workflow service is a WSRF-conform service. Therefore it consists of a factory, beside the actual service. The factory services for the Workflow Service is called Workflow Service Factory and creates instances of the corresponding Workflow Service.

WSRF Web Services Resource Framework. See [1].

References

[1] Tim Banks. Web Services Resource Framework (WSRF) - Primer v1.2. http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf, May 2006.

[2] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995.

[3] Steve Graham and Jem Treadwell. Web Services Resource Properties 1.2 (WS-ResourceProperties). `http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf`, April 2006. OASIS Standard.

[4] Stefan Gudenkauf, Wilhelm Hasselbring, Felix Heine, André Höing, Odej Kao, and Guido Scherp. BIS-Grid: Business Workflows for the Grid. In *The 7th Cracow Grid Workshop*. Academic Computer Center CYFRONET AGH, 2007.

[5] Felix Heine, Andre Höing, Stefan Gudenkauf, Guido Scherp, Holger Nitsche, and Jens Lischka. BIS-Grid Deliverable 3.1: Specification. Technical report, BIS-Grid, December 2007.

[6] Andre Höing, Stefan Gudenkauf, and Guido Scherp. BIS-Grid Deliverable 2.1: Catalogue of WS-BPEL Design Patterns. Technical report, BIS-Grid, August 2008.

[7] Michael Kay (editor). XSL Transformations (XSLT) Version 2.0. W3C Recommendation, World Wide Web Consortium (W3C), January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[8] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`, Februrary 2005.

[9] Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo. Security Assertion Markup Language (SAML) V2.0 Technical Overview. `http://www.oasis-open.org/committees/download.php/22553/sstc-saml-tech-overview-2%200-draft-13.pdf`, February 2007. Working Draft.