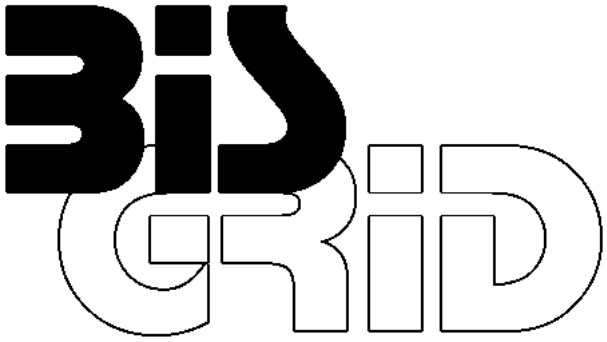

Betriebliche Informationssysteme:
Grid-basierte Integration und Orchestrierung

Deliverable 3.3

Work Package 3: Erweiterte 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:
30.04.2009

Contents

This document is part of Deliverable 3.3 “Erweiterte 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 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[6].

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

1 Introduction

This document is part of Deliverable 3.3 and represents the documentation of the BIS-Grid middleware prototype. The document is to be 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[6].

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), information on the use of the software development tool Maven 2, information on the project structure, information on the configuration of the BIS-Grid middleware, and information on the deployment of the service extensions to the Grid middleware UNICORE 6[3] that represents the development basis of the BIS-Grid engine. In Section 2 we discuss the Workflow Management Service of the BIS-Grid engine 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 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 developed a workflow engine, the *BIS-Grid 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 used is ActiveBPEL[5] since it exhaustively supports the WS-BPEL standard, and is well

[3] http://www.unicore.eu
[4] http://www.globus.org/toolkit/
[5] http://www.activevos.com/community-open-source.php

accepted in the business domain as well as in the Grid domain. We refrain from extending well-adopted standards and technologies 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 Overview

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, the 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 [5] we illustrate our considerations on security within the BIS-Grid solution.

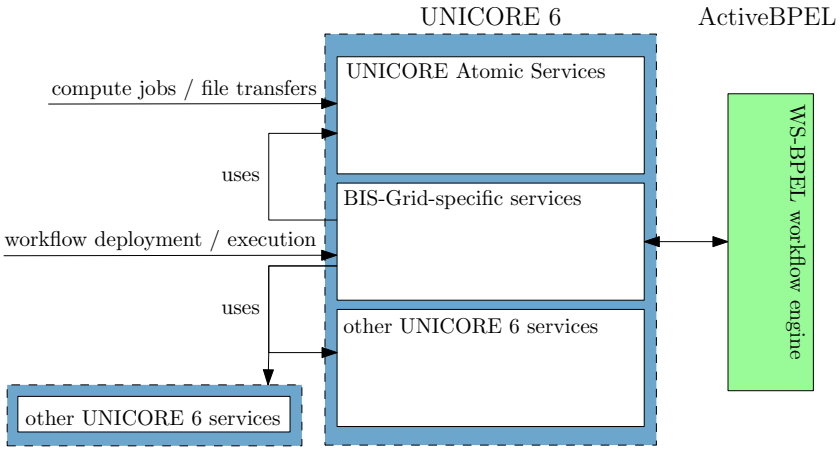

Figure 1: Overview on the Architecture of the BIS-Grid solution

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, 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 standard WS-BPEL engines.

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 [5] we also present our considerations on load balancing the BIS-Grid solution.

Beside these advantages there are issues that have to be addressed when using such a decoupled system architecture. The WS-BPEL code that is necessary to call a Grid Service is more complex than it would be with proprietary language extensions. For example, even if one wants to use a WSRF resource of a Grid Service for one single invocation, one has to explicitly create, use, and destroy it resulting in 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 [6], and in [2] in Deliverable 2.1 [7], 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 greatly 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 library updates are automatically retrieved. This is, for example, the case if 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, we list the *WSRFlite* project on which UNICORE 6 is based and the *Unicore Atomic Services* project. Maven 2 then analyses the listed projects and fetches the corresponding libraries recursively.

Additionally, there are many plugins available to Maven 2 that, for example, allow to create XMLBeans for WS-Resource Properties and messages, and to create Eclipse (http://www.eclipse.org/) projects automatically with all necessary libraries included.

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 BIS-Grid engine Structure

Basically the BIS-Grid engine consists of two mostly independent services: the Workflow Management Service and the Workflow Service. Beside this service there is also a Proxy Service managing instances of proxies where WS-BPEL engines can connect to. This services are implemented in the following package module structure.

name	description
`service.management`	The Workflow Management Service including the Workflow Management Service Factory.
`service.workflow`	The 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.proxy.redirect`	Because of a bug in the Jetty Servlet container, the standard proxy is not able to handle secured SSL connections. The redirect proxy is used to accept secured connections and forward the request with working http-headers to the standard proxy.
`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 used by more than one other package, e.g. WS-BPEL engine adapters, exceptions, or entity classes.

Table 1: Project Structure

The BIS-Grid engine consists of 6 maven projects, each presenting a module (cp. Table 1). Figure 2 shows the dependencies between the modules. The packages `service.messages` and `service.common` serves as utility packages for the BIS-Grid engine services. The `service.proxy` package is the implementation of the proxy service. The `service.proxy.redirect` is independent from the other packages (cp. Section 4). `service.management` and `service.workflow` are the implementation of the Workflow Management Service and the Workflow Service, the main services for managing and execution workflows in the BIS-Grid engine.

The java packages are named in according to the modules. We agreed upon `de.dgrid.bisgrid` as prefix for each module to reflect the fact that BIS-Grid is a subproject of

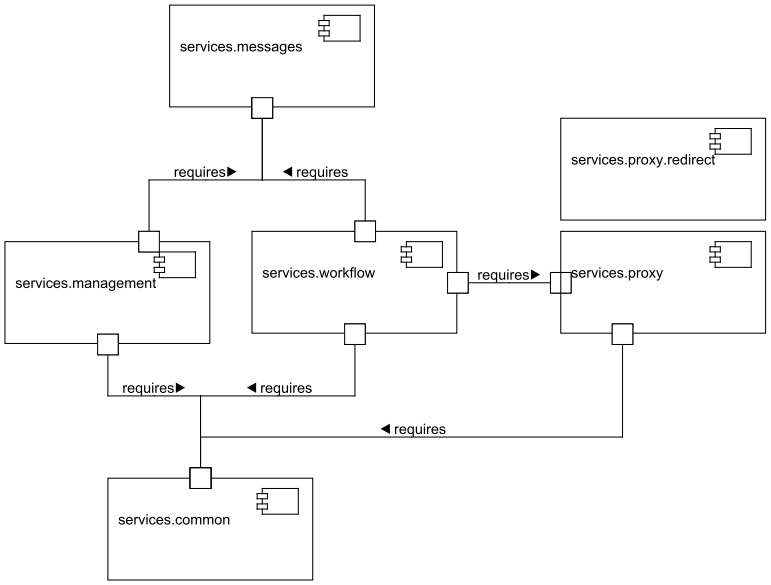

Figure 2: BIS-Grid Components and Dependencies

the D-Grid that again is a German initiative. Hence all services-modules are named as `de.dgrid.bisgrid.service` and followed by one package for the above presented modules. As example, `de.dgrid.bisgrid.service.management` is the package name for the Workflow Management Service. The other modules are named analogously.

1.5 BIS-Grid Configuration

The BIS-Grid engine is configurable by Java Properties. A `.properties` file and the corresponding `Properties` object implementation is used. A `BISGridProperties` object, which is created when the services start, retrieves the configuration information. This object, implemented as a Singleton [3], is responsible for the configuration of all BIS-Grid services. The properties are loaded in two steps, the default properties first and user customised properties second. The location where the customized, 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 jar archives in the classpath (`bisgrid.properties.all`)
2. Read custom properties from file given in `UAS.properties` by property `bisgrid-.properties.file`

The remainder of this section describes the configuration parameters. The WS-BPEL workflow engine is 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 host name 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 encryption (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 following parameters are engine-specific for WS-BPEL engines that use databases for persistence issues, e.g. the ActiveBPEL engine. This configuration is used during undeployment to process a clean up of persistence data.

- `bpel.workflow.engine.deployment.dir_<id>`: The directory, where the BPEL engine stores the deployment packages. This is used for undeploying workflows from ActiveBPEL. (At the moment, it is assumed the default case that the BIS-Grid engine and the WS-BPEL engine run on the same machine, and that WS-BPEL process undeployment functions by removing the deployment package vom the deployment directory of the respective WS-BPEL engine. For remote undeployment, a web service that deletes such packages could be used.)
- `bpel.workflow.engine.persistence_<id>`: Indicates if the engine is configured to use persistence (true/false).
- `bpel.workflow.engine.persistence.db.driver_<id>`: The JDBC/ODBC Driver to connect to the database (e.g. org.git.mm.mysql.Driver).
- `bpel.workflow.engine.persistence.db.host_<id>`: The host of the persistence database.
- `bpel.workflow.engine.persistence.db.port_<id>`: The port of the persistence database.
- `bpel.workflow.engine.persistence.db.username_<id>` Database user username.
- `bpel.workflow.engine.persistence.db.password_<id>` Database user password.
- `bpel.workflow.engine.persistence.db.database_<id>` Database name.

The BIS-Grid engine is ready for using a service certificate that can be used to establish secured SSL connections to other hosts providing Grid services or even to enable a secured communication between the BIS-Grid engine and the WS-BPEL workflow engine or external services. The following configuration defines a key- and truststore that includes a BIS-Grid engine certificate.

- `bisgrid.keystore.file`: The file with the server certificate.
- `bisgrid.keystore.password`: The password to the keystore.
- `bisgrid.keystore.type`: The type of the keystore (e.g. jks or pkcs12).
- `bisgrid.keystore.alias`: The alias of the certificate that should be used to enable client authentification.
- `bisgrid.truststore.file`: The file of the server's truststore.
- `bisgrid.truststore.password`: The password to the truststore.
- `bisgrid.truststore.type`: The type of the truststore (e.g. jks or pkcs12).

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 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 to retrieve specialized adapters modules, for example the `MonitoringAdapter` or the `DeploymentAdapter`. Hence, the configuration file needs to include information where to find the implementation of the adapter interface. Using the type information of the WS-BPEL engine, the class `BisGridProperties` searches for a property that is named `<type>_Adapter`, where `type` denotes the type of the WS-BPEL engine from the `bpel.workflow.engine.type` configuration. BIS-Grid is shipped with an adapter for the ActiveBPEL engine "ActiveBPEL_Adapter" by default. Information on how to implement an adapter can be found in Section 3.3).

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

Furthermore, the BIS-Grid engine supports configurations that affect the Workflow Management Service:

- `bisgrid.deployment.directory`: The folder where the deployment packages are stored permanently on hard disk.
- `bisgrid.cleanup.working.directory.on.startup`: When the UNICORE 6 container is restarted, all information of deployed workflows will be deleted. This affects only the directories for deployed workflows in the deployment directory. Such an directory contains, for example, the created ActiveBPEL deployment package or the unpacked BIS-Grid Deployment Archive (true/false).
- `bisgrid.deploy.workflow.packages.on.startup`: It is possible to put BIS-Grid deployment archives into the deployment directory. If this configuration parameter is set true, the engine deploys all of the archives on startup.

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

- `check.for.duplicate.workflow.names`: If activated, the Workflow Management Service checks if there already is a workflow deployed with the same name so that it is impossible to overwrite an existing workflow.

The following configurations are used in the Workflow Service:

- `bpel.soapaction.url`: The URL is used to mark and detect WS-BPEL operations with the help of SOAP Actions. (e.g. http://bisgrid.de/bpel-operation)
- `bisgrid.externalcall.handler.out`: A comma-separated list of handler class names that should be added into the out-handler pipeline for calling external services. Note that it is not possible to insert handlers that require parameters.
- `bisgrid.externalcall.handler.in`: A comma-separated list of handler class names that should be added into the in-handler-pipeline for calling external services. Note that it is not possible to insert handlers that require parameters.

The following configurations are used in the proxy service:

- `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.bpelengine.id`.
- `test.bpelengine.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 UNICORE 6.
- Install Apache Tomcat and deploy an 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 jar archives into the folder `unicorex/lib`.
- Add the class `de.dgrid.bisgrid.deployment.DeployBisGrid` to the startup code in the `UAS.properties` configuration file located in `unicorex/conf`.
- Configure the file `bisgrid.properties` and add the location into the `UAS.properties` configuration file with the property `bisgrid.properties.file`.

- Create the previously configured deployment directory.
- Start the Tomcat service with the ActiveBPEL Engine.
- Start UNICORE 6's UNICORE/X service container.

2 Workflow Management Service

This section describes technical details on the so called Workflow Management Service. As part of the BIS-Grid service extensions, the Workflow Management Service is responsible for deploying, undeploying, and redeploying workflows. It is implemented as a WSRF service, meaning there exists a factory service (plain web service) to create service instances of the WSRF service, whereby each instance manages one single workflow.

In order to deploy a workflow a so called BIS-Grid deployment package (ZIP compressed archive, see Section 2.1) must be submitted to the Workflow Management Service. The deployment process is based on a so called BIS-Grid deployment descriptor, see Section 2.2. The deployment, undeployment, and redeployment processes are described in Sections 2.3, 2.4, and 2.5. The preparation of the Workflow Factory Service and the corresponding Workflow Service as part of the deployment process is described in Section 2.6 in more detail. Finally, Section 2.7 is about the class hierarchy of the Workflow Management Service.

2.1 BIS-Grid Deployment Package

The BIS-Grid deployment package is a ZIP-compressed archive that contains all necessary files to deploy a workflow in the BIS-Grid engine using the Workflow Management Service. Currently, the BIS-Grid deployment package comprises the following elements:

- WS-BPEL process file (mandatory): A file with file extension `.bpel`, containing the WS-BPEL workflow description, is expected at the top level directory of the deployment package.

- WSDL files (optional): The deployment package may include several files with file extension `.wsdl`, each containing a WSDL definition that either describes the interface of an external service that is used in the workflow, or that describes the workflow service interface itself. Alternatively, it is possible to omit the WSDL files in the deployment package but to reference external WSDL locations in the BIS-Grid deployment descriptor from where the WSDL files are retrieved at deployment time.

- BIS-Grid deployment descriptor file (mandatory): The deployment package requires a file with file extension `.bdd` to be located at the top level directory - containing binding information on partner links, references to internal or external WSDL, XML schema locations, and security information. For more information on the BIS-Grid deployment descriptor see Section 2.2.

2.2 BIS-Grid Deployment Descriptor

The BIS-Grid deployment descriptor is an XML file (see Appendix A for the corresponding XML schema file) that contains information on the deployment of one workflow.

At present, the deployment descriptor's structure is proprietary and close to the ActiveBPEL deployment descriptor, but shall be generalized in future. To do so, at the partner TU Berlin a diploma thesis is currently being conducted aiming at developing a generic deployment descriptor under consideration of existing deployment descriptors used in several commercial products.

Generally, the BIS-Grid deployment descriptor consists of the same information that can be found in vendor specific descriptors, but is extended by additional information related to security issues (credential and service description). All in all, the BIS-Grid deployment descriptor contains the following information.

- `Process information`: The process information specifies the name of the WS-BPEL process and the location of the corresponding file in the BIS-Grid deployment archive.

- `Binding of WS-BPEL partner links`: WS-BPEL partner links define roles in a web service communication where each role is bound to one abstract WSDL port of a WSDL interface. In this section binding information is specified for each role with its corresponding WSDL port/interface. This means, a role is either bound to the WS-BPEL process itself (the WS-BPEL engine must offer a corresponding web service endpoint) or to an external web service endpoint which can be specified as *static* (i. e., as a WS-Addressing endpoint reference within the deployment descriptor) or *dynamic* (i. e., the WS-Addressing endpoint reference is constructed during the process execution).

- `WSDL and XML schema file references`: WSDL and XML schema files are used to describe the web service interface and data structures of the WS-BPEL process itself or of external web services that are invoked within the process. These and further referenced files[7] must be available at runtime. From the technical view a WS-BPEL engine requires a corresponding description file (WSDL or XML schema) for each namespace of the elements used. If the location of a description file is missing, its location can be configured in the BIS-Grid deployment descriptor either as a local reference, if the BIS-Grid deployment package contains the description file, or as an external reference, if the description file is accessable via an URL by HTTP or FTP, for example.

- `Credential descriptions`: In this section different credentials can be configured. Here, the alternatives are username/password, SAML delegation or a X.509 certificate located in a keystore. This information is used to provide initial values for the `CredentialDescriptionResiourceProperty` (cp. Section 3.5.2).

- `Service descriptions`: This section describes processing call configurations for services that are used in a workflow. This information is used to to provide initial values for the `ServiceDescriptionResourceProperty` (cp. Section 3.5.2).

[7] Each WSDL and XML schema can import other files

For subsequent implementation iterations we plan to extend the BIS-Grid deployment descriptor by additional security issues such as a fine-grained role-based access control (RBAC) for each workflow method based on XACML policies (and SAML assertions for roles). It is intended that these policies can be partially modified (configurable) during runtime. Please refer to the BIS-Grid workflow engine's specification[6] and Deliverable 2.5 "Spezifikation der Anforderungen an Sicherheit und Service Level Agreements im Zusammenhang mit WS-BPEL" [8] (in German) for more information. Furthermore, the support for invoking GSI-secured services (using proxy certificates), which are used in Globus Toolkit 4, will be added to service and credential description.

2.3 Deployment

The deployment process describes deploying a WS-BPEL process in the BIS-Grid engine, thereby creating a corresponding Workflow Service for the deployed workflow (see Section 3) to the UNICORE 6 service container. Figure 3 illustrates the deployment process in general. The deployment process consists of the following steps:

1. `Preparation`. A BIS-Grid deployment package is received, stored in the deployment directory (see Section 1.5), and unpacked. All contents of the deployment package (see Section 2.1) are stored in corresponding (WSRF) resource properties.

2. `Pattern injections`. The WS-BPEL process and WSDL files must be modified in order to cope with implementation-specific issues that originate from the BIS-Grid engine's architecture - the use of UNICORE 6 as a WSRF proxy for an arbitrary WS-BPEL engine for Grid workflow execution, to be more specific. These modification are held transparent to the workflow user and are applied in form of WS-BPEL patterns that are mainly based on XSLT transformations. For more information on these patterns please refer to Deliverable 2.1 "Catalogue of WS-BPEL Design Pattern". The modified WS-BPEL process and WSDL files are also stored in corresponding (WSRF) resource properties.

3. `WS-BPEL engine deployment`: The corresponding WS-BPEL process is deployed via the configured WS-BPEL engine adapter (see Section 1.5).

4. `UNICORE 6 deployment`: A Workflow Factory Service instance is created and a corresponding Workflow Service (see Sections 2.6 and 3) is deployed to the UNICORE 6 service container. Please note that UNICORE 6.2 supports deployment/undeployment of services during runtime (hot deployment) directly.

If no error occurs an appropriate deployment success response is sent to the workflow designer/manager. Otherwise, the Workflow Management Service tries to roll back previous deployment steps and returns an appropriate error message, which may include additional error information received from the WS-BPEL engine adapter.

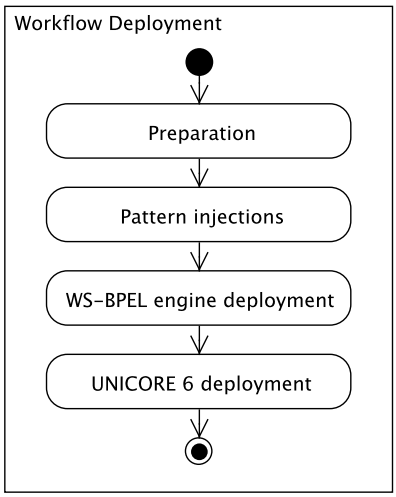

Figure 3: Activity Diagram - Workflow Deployment

2.4 Undeployment

In general, workflow undeployment can be understood as a roll back of a previously executed deployment process. Figure 4 illustrates the undeployment process, consisting of the following steps:

1. `Factory instance removal`. The Workflow Factory Service instance of the corresponding Workflow Service is removed from the UNICORE 6 service container to prevent the creation of new Workflow Service instances.

2. `Termination`: Regarding the handling of active service instances the workflow designer/manager has the following alternative termination options.

   - `Instant`: All active service instances are destroyed immediately and the Workflow Service is removed completely from the UNICORE 6 service container.
   - `ByDate`: Wait for the normal termination of all active service instances until a certain date at which all instances are destroyed. Therefore, the (WSRF) resource lifetime of each service instance is set to the corresponding date.
   - `ByPeriod`: Wait for the normal termination of all active service instances within a given time period. After the time period expired, the service instances are destroyed. Therefore, the (WSRF) resource lifetime of each service instance is set to the corresponding date.

   Please note that if the termination type is set to `Instant` the following substeps are performed immediately. In all other cases, these steps are triggered after the termination (normal or forced) of the last active service instance.

3. `Workflow Service undeployment`: The workflow-specific Workflow Service is removed completely from the UNICORE 6 service container.

4. `WS-BPEL engine undeployment`: The corresponding WS-BPEL process is undeployed via the configured WS-BPEL engine adapter.

5. `Cleanup`: All files related to the workflow are removed from the deployment directory and all corresponding resource properties are deleted.

If no error occurs an corresponding success response is sent back to the workflow designer/manager. Otherwise, the Workflow Management Service stops the undeployment process and returns an appropriate error message, which may include additional error information received from the WS-BPEL engine adapter. In that case, an administrator might be required for undeployment error handling.

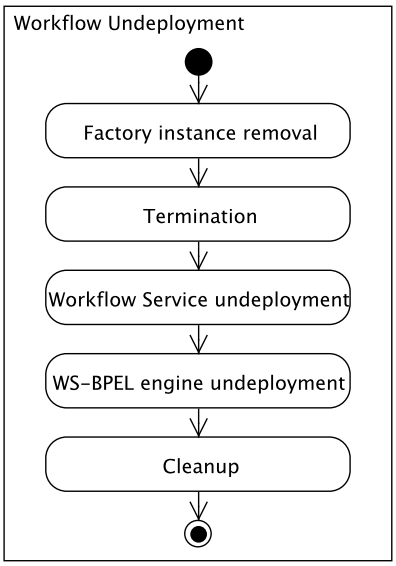

Figure 4: Activity Diagram - Workflow Undeployment

2.5 Redeployment

The redeployment process consists of an undeployment process followed by a deployment process, see Figure 5. If no error occurs the success response of the deployment process is sent to the workflow designer/manager. Otherwise, the error message of either the undeployment or the deployment process is returned by the Workflow Management Service, depending on which subprocess - the undeployment or the deployment process - failed.

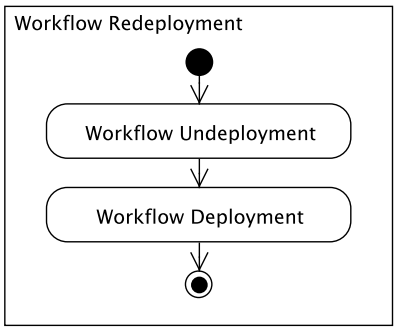

Figure 5: Activity Diagram - Workflow Redeployment

2.6 Workflow Service deployment

As described in Deliverable 3.1 "WS-BPEL Engine Specification" [6], when a workflow is deployed with the Workflow Management Service, a Workflow Factory Service instance is created and a corresponding Workflow Service is deployed in the UNICORE 6 service container of the BIS-Grid engine (see Section 2). The Workflow Factory Service instance is responsible for creating Workflow Service instances. Both, the Workflow Factory Service instance and its correspondingWorkflow Service, represent one single deployed workflow (one deployed BIS-Grid deployment package) on the UNICORE 6 side of the BIS-Grid engine. Furthermore, each workflow execution is represented by one single Workflow Service instance.

Based on the requirements to deploy and undeploy workflows at runtime, the design of the Workflow Factory Service and the Workflow Service turned out to be quite challenging. However, since "hot deployment" of services during runtime is officially supported by the UNICORE middleware since UNICORE 6.2, one essential technical prerequisite is given. A general design overview is presented in the following - for implementation details please refer to Section 3. We employed an iterative design process regarding several design alternatives towards the final design of the Workflow Factory Service and the Workflow Service. This final design is based on the creative use of stateful WSRF-based services, described as follows.

The Workflow Factory Service is implemented as a WSRF service, but it has no corresponding factory service. Instead, to create a Workflow Factory Service instance the corresponding `Service Home` is invoked directly by the Workflow Management Service using Java calls. Thus, the (in this case unnecessary) feature to use web service calls to create Workflow Factory Service instances is not available. Each Workflow Factory Service instance itself represents a workflow-specific service factory to create corresponding Workflow Service instances. Thus, each Workflow Service consists of one WSRF service and has no separate factory service, too. Regarding the deployment process we have the following situation:

- `Startup`: When the UNICORE part of the BIS-Grid engine (i. e., the UNICORE 6 service container with the BIS-Grid service extensions) is started, the Workflow Management Service and the WSRF service of the Workflow Factory Service are deployed automatically.
- `Deployment`: During a deployment process (see Section 2.3) a Workflow Factory Service instance is created by directly invoking the `Service Home`, and a corresponding new Workflow Service (WSRF service) is deployed at runtime.

When the deployment process is finished the created Workflow Factory Service instance and the new Workflow Service are accessible via the following locations:

- `Workflow Factory Service instance`:
  `https://<HOST>:<PORT>/<SITE-NAME>/services/`
  `BIS_Grid_Workflow_Factories?res=<WORKFLOW-NAME>Factory`
- `Workflow Service` :
  `https://<HOST>:<PORT>/<SITE-NAME>/services/`
  `<WORKFLOW-NAME>WorkflowService`

The implementation of the WSRF service of the Workflow Service is generic (see Section 3). During the deployment of a Workflow Service the corresponding WSRF service is configured according to the workflow properties. For example, the workflow-specific interface is offered by this WSRF service.

2.7 Class Hierarchy

The class hierarchy of the Workflow Management Service is shown in Figure 6. Note that only the WSRF service is shown and the factory service is omitted. The Interface `IWorkflowManagementService` offers the three methods `deploy redeploy`, and `undeploy`, corresponding to the deployment, redeployment and undeployment functionality of the Workflow Management Service. As the implementing class `WorkflowManagementService` represents a WSRF service (instance) additional interfaces (as `WSResource`) are implemented and classes are extended (as `WSResourceImpl`) that are specific for UNICORE 6's WSRF implementation. Currently, the following (read-only) resource properties are offered by a Workflow Management Service instance:

1. `deploymentPackageProperty`: The complete BIS-Grid deployment package.
2. `deploymentPackageNameProperty`: The name of the BIS-Grid deployment package.
3. `deploymentDescriptorProperty`: The BIS-Grid deployment descriptor.
4. `bpelFileProperty`: The WS-BPEL process of the workflow extracted from the BIS-Grid deployment package.

5. `bpelFileProcessedProperty`: The WS-BPEL process after the injection of WS-implementation-specific BPEL patterns (see Section 2.3).

6. `wsdlFilesProperty`: The required WSDL files extracted from the BIS-Grid deployment package.

7. `wsdlFilesProcessedProperty`: The WSDL files after pattern injection (see Section 2.3).

8. `workflowServiceFactoryProperty`: Reference to the corresponding Workflow Management Service instance (WS-Addressing endpoint).

9. `isDeployedProperty`: Indicates if a workflow is correctly deployed or not (`true` if successfully deployed, `false` otherwise).

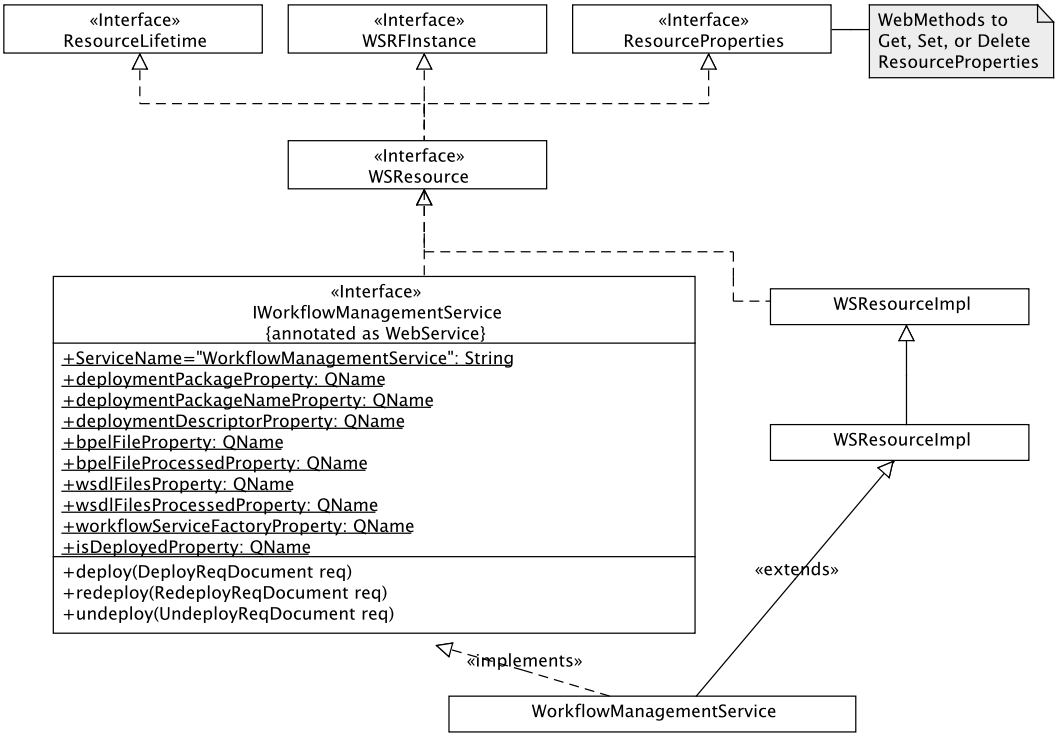

Figure 6: Class Diagram - Workflow Management Service

3 Workflow Service

This section describes the implementation of the Workflow Service. First, an overview of the integration of the Workflow Service into the UNICORE 6 architecture is given in Section 3.1. After that, we describe our most important service extensions to UNICORE 6. This includes a WSDL writer in Section 3.2, which merges WSDL interfaces into one single interface, a modular adapter concept to support different WS-BPEL engines in Section ??, and a specific `HandlerPipeline` of the Workflow Service in Section 3.4. The state representation of the Workflow Service is described in Section 3.5. Section 3.6 illustrates the communication between UNICORE 6 and the WS-BPEL engine, as well as how the Workflow Service is acting as a service call proxy. Finally, Section 3.7 addresses monitoring issues.

3.1 Class Hierarchy

The Workflow Service implements the interface `IWorkflowService`. Since the implementing class `WorkflowService` represents a WSRF service (instance), additional interfaces (as `WSResource`) are implemented and classes are extended (such as `WSResourceImpl`) that are specific to UNICORE 6's WSRF implementation. The corresponding class hierarchy is depicted in Figure 7. As the methods offered by the Workflow Service WSDL interface depends on the deployed workflow, the interface `IWorkflowService` offers only the internally used (non-web service) method `processBPELCallMessage` which is a general method to process messages that should be forwarded to the WS-BPEL engine (see Section 3.4). The resource properties of `IWorkflowService` are described in Section 3.5 in more detail.

3.2 WSDLWriter

Each UNICORE 6 service has a `WSDLWriter` that is responsible for generating its WSDL. Therefore it defines the interface and therewith all operations that can be used by web service clients. The Workflow Service provides a WSDL that is a combination of the regular interface (operations directly implemented via annotated methods) plus additional methods that are necessary to execute the workflow and that are provided by the WS-BPEL workflow engine. The latter part is undefined at development time and depends on the deployed workflow. This entails a complex substitution of the regular `WSDLWriter`.

To create such a combined WSDL, we developed a new `WSDLWriter`, the `CombiningWSDLWriter`. It uses the original `WSDLWriter` for the regular service operations and combines them with the workflow operations in an additional step. This is done by integrating information from the workflow's WSDL interface, such as XML schema types, imports, namespaces, bindings, port types, and messages. The service has the port `WorkflowServiceBpelPort`, pointing to the binding `BpelBinding` (see Listing 1).

Listing 1: Example Service Port for WS-BPEL Operations

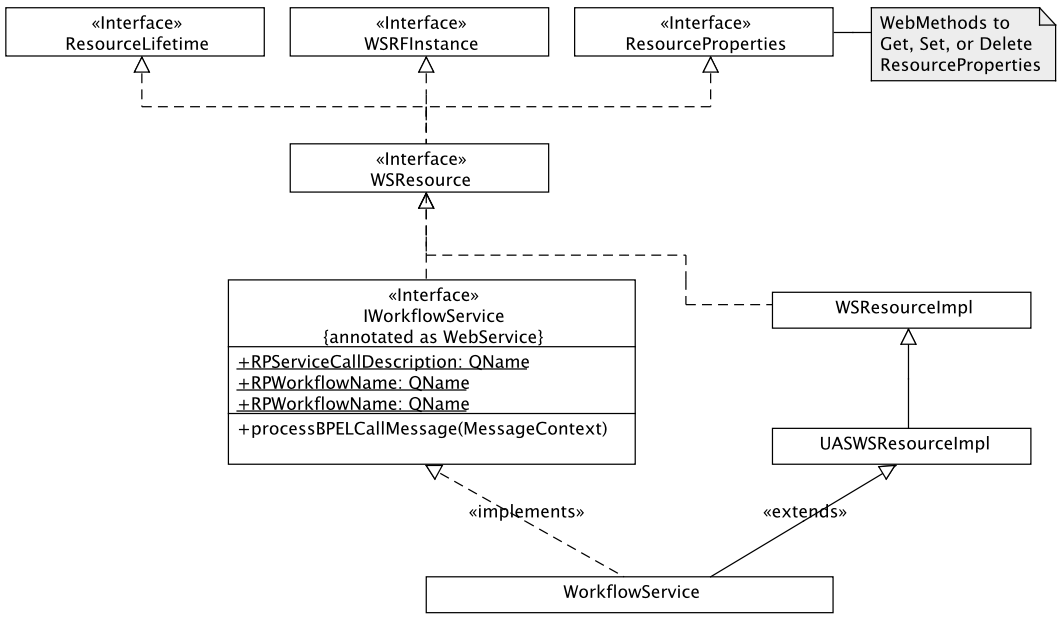

Figure 7: Class Diagram - Workflow Service

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

The `BpelBinding` is of type `BpelHttpPort`. The operations in this binding receive a newly generated SOAP-Action-Tag, because ActiveBPEL does not support SOAP Actions in their generated WSDLs. The names of all SOAP-Actions of the WS-BPEL operations start with `http://bisgrid.dgrid.de/bpel-operation` (configureable in the BIS-Grid properties) followed by the operation name. The SOAP style used is "document/literal" (i. e., the same SOAP style as for regular operations). An example for a generated binding with a single operation named "call" is presented 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 is exemplaryly presented 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 their XML schema information are simply copied from the WS-BPEL workflow WSDL interface into the Workflow Service interface. The new WSDL interface is used in the `BISGridServiceSetterHandler`. More information regarding this topic can be found in Section 3.4.2 and in the Specification of the BIS-Grid engine [6].

3.3 Adding a new WS-BPEL-Engine-Adapter

The BIS-Grid engine is designed flexible to support new adapters to other WS-BPEL workflow engines without the necessity of code modifications to the BIS-Grid service extensions. Figure 8 depicts a class diagram of the adapter framework.
The point of entry to the adapter framwork is the `AdapterFactory`. If a component of the BIS-Grid engine requires non-standard WS-BPEL workflow engine functions, it uses the factory to get an appropriate adapter. The type of the WS-BPEL engine is used as identifying parameter so that the correct implementation is chosen. The factory looks up the matching class at the `BISGridProperties` to find the correct `Adapter` implementation.

For special purposes, the adapter provides special additional interfaces, the `AdapterModules`. All `AdapterModules` offers the possibility to inject special WS-BPEL code into the WS-BPEL process description before the workflow is actually deployed to the WS-BPEL engine. This enables the developer to use engine-specific functions during the execution of the WS-BPEL process.

Currently, we see the need of two module interfaces: the `DeploymentAdapter` and the `MonitoringAdapter`. The `DeploymentAdapter` provides the deployment and undeployment functionalities of the corresponding WS-BPEL engine. All operations to prepare a specialized deployment package, for example, used by web services for deployment, or

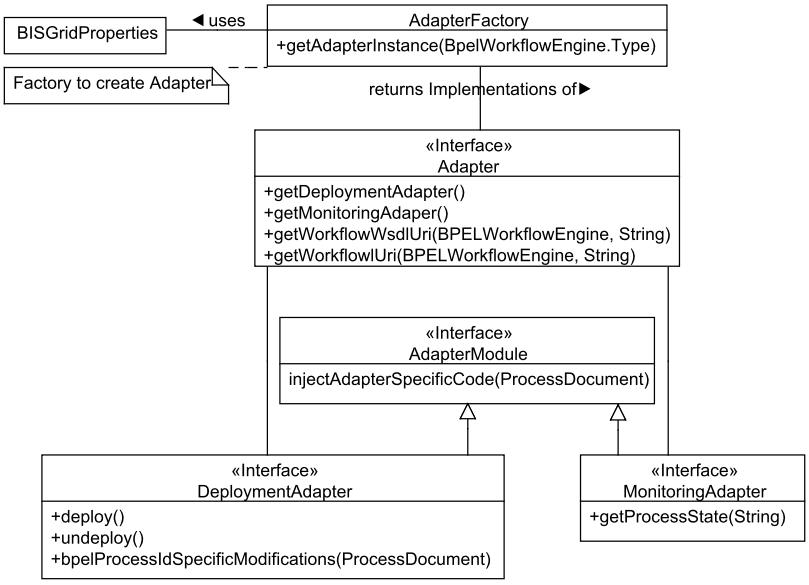

Figure 8: Class Diagram - Adapter Framework

the injection of engine-specific modifications into the WS-BPEL Code, should be implemented here. To support a new WS-BPEL workflow engine, a developer has to execute the following actions:

- Implement the `Adapter` interfaces and all sub-interfaces (for web service calls, it is possible to use generated stubs) and provide the implementation in a jar archive.
- Copy the jar archive representing the adapter implementation into the folder `unicorex/lib` of the UNICORE 6 installation.
- Add the full class name (including package) to the file `bisgrid.properties`.
- Add the new WS-BPEL engine configuration to the file `bisgrid.properties`.

3.4 Handler Pipeline

Handler Pipelines are a very important concept in each web service framework. They are, for example, used to analyze and process SOAP messages before and after the actual web service calls. This section describes the standard pipeline used for UNICORE ATOMIC SERVICES and the BIS-Grid specific modifications.

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, which also lists the services that should be deployed on

startup. Some Handlers are fixed in the handler pipeline by deploying a service as WSRF service (the services inherit `WSResourceImpl`), and some additional handlers are necessary if the service implementation inherits `UASWSResourceImpl`. Further handlers can be added using the configuration file.

WSRFlite uses the so-called `MessageContext` (Java Object) to transport the complete invocation context of the web service call through the handler framework. Besides the in- and out-messages, it can store all kind of information that may subsequently be used in other handlers of the pipeline. The standard pipeline looks like shown in Figure 9.

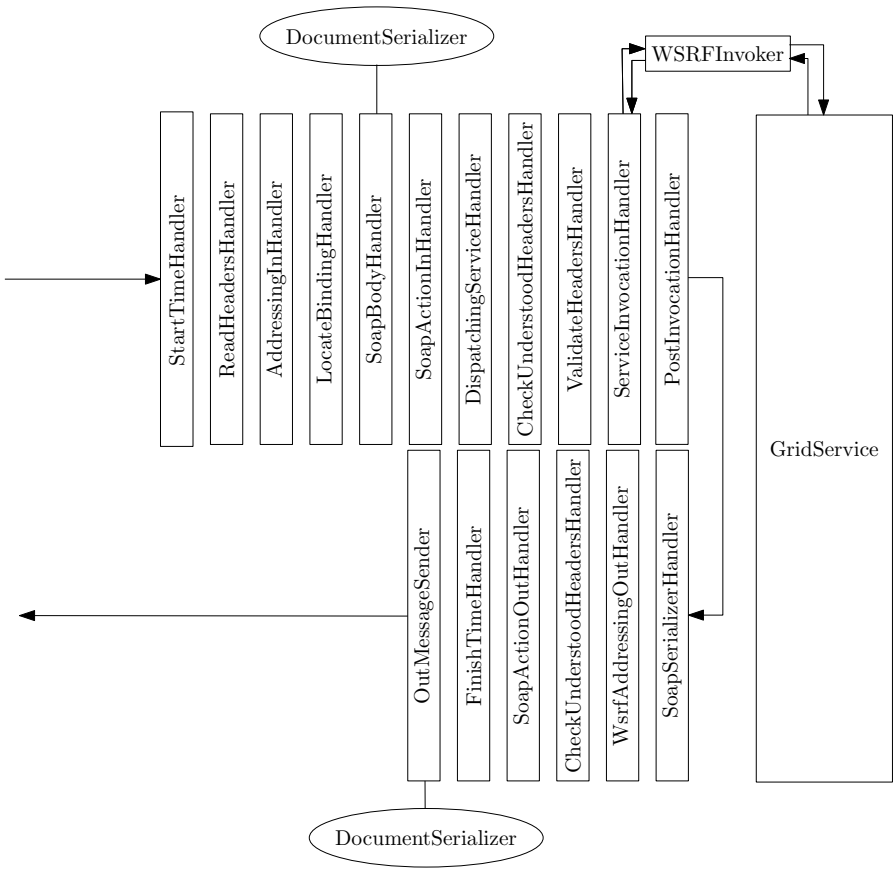

Figure 9: Grid Service Handler Pipeline

Each handler represents a small part of SOAP message processing. Thereby, some handlers need other handler to be invoked in advance. Some handler are more complex than others, but generally, all handlers provide 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

in `MessageContext` header variable. Checks if the message is a fault message and, if it is a fault, reads the fault and throws an XFire exception.

- **AddressingInHandler**: Uses the header from the `MessageContext` 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 `MessageContext`. 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 until now. This handler inspects the SOAP Action 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 certain information to the handler pipeline, for example, that this pipeline understands the WS-Addressing namespace. This information is required and used by 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, for example, 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. In case of a service outgoing pipeline, the `AddressingOutHandler`'s *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

We implemented a customized handler pipeline due to several implementation-specific issues. For example, this is the need for combining a WSDL interface and a call of a common method in order to process service calls to the WS-BPEL engine. This is already described in the specification of the BIS-Grid engine [6]. The Workflow Management Service deploys the Workflow Service and executes the following configuration actions for the handler pipeline. The purpose of each new handler is described afterwards.

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

Furthermore, we change the *Invoker* object that performs the Java 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 10. The changes are marked with a gray background.

The `BISGridAddressingInHandler` is an extension of the original `AdressingInHandler`. The extension checks the WS-Addressing header and the included SOAP-Action element. If the action start with the string "BPEL-SOAP-Action" (see Section 1.5), this message is regarded as a workflow execution message. This means that the BIS-Grid engine must forward it to the WS-BPEL engine. Otherwise the normal handler pipeline is used.

The Workflow Service has no real implementation for workflow operation. Therefore, the generic forwarding method of the Workflow Service must be used. The `BISGridAddressingInHandler` does not use the original service object but creates a fake service object that represents an existing implementation of the called operation.

The `BISGridPostInvocationHandler` checks a flag that is set by the `BISGridAddressingInHandler` if the operation is a WS-BPEL operation. Only if this call is indicated as a standard call, the result values of the invoked operations are used to create a response. For WS-BPEL calls, the response is already set in the `processBPELCallMessage`

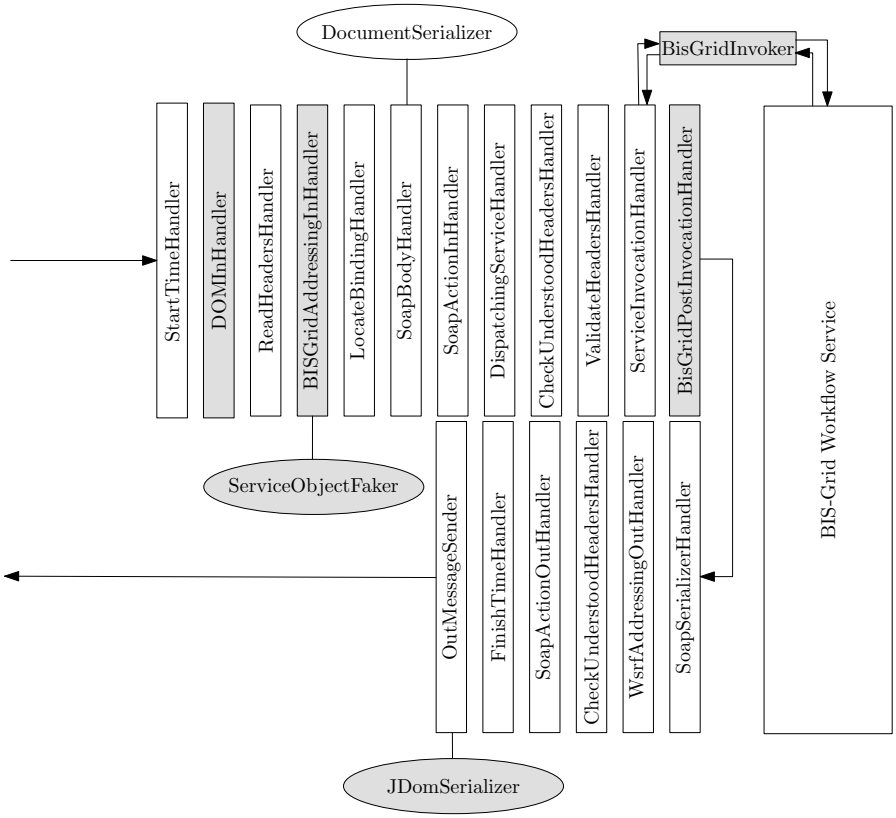

Figure 10: BIS-Grid Workflow Service Handler Pipeline

operation into the `MessageContext`. The operation also changes the serializer into a `JDOMSerializer`.

The `DOMInHandler` is required to save the complete message in the `MessageContext` object. The `MessageContext` is passed to the common method `processBPELCallMessage` that processes all WS-BPEL calls (cp. Figure 7 at page 23). The message forwarding can be done easily with the created DOM Object.

3.5 Workflow Service State

Each Workflow Service instance holds information that represents its current state. Thereby, we distinguish between mutable and immutable state information. Immutable state information only serves as a source of information for the user or for other services, mutable state information can also be used for configuring the Workflow Service. Changes of mutable resource properties can be done using service calls defined in the WS-ResourceProperties specification [4]:

- `SetResourceProperties`
- `DeleteResourceProperties`

- `InsertResourceProperties`
- `UpdateResourceProperties`

3.5.1 Immutable Resource Properties

The following list gives an overview of the immutable resource properties and their intention.

- **BpelEngine**: The address of the WS-BPEL engine used to execute the workflow instance. The engine is defined by the elements *id*, *hostname*, *port*, *engineType*, and the flag *useSSL*.
- **WorkflowName**: The name of the service representing the workflow as it is deployed in the WS-BPEL engine. This information is necessary for forwarding messages to the WS-BPEL engine (all client messages are forwarded to this service).
- **BpelProcessId**: The id (string) of the process in the WS-BPEL engine. In case of the ActiveBPEL Engine, this is a number (long). The ids must be casted to the correct datatype by the adapter. It is only available after the workflow is started and the first external service call is occurred (extracted from the message header, see [7]).
- **WorkflowStatus**: The workflow status represents the current status of the workflow. Each time this property is requested, the `MonitoringAdapter` of the WS-BPEL Engine is used to retrieve the current status of the workflow. The information included in the ResourceProperty comprises the state of each WS-BPEL activity and also error information if an activity failed.

3.5.2 Modifiable Resource Properties

- **ServiceDescription**: The service descriptions are used to configure external service calls. Each description consists of the following configuration elements:
  - ServiceAddress (required): The Service Address is the endpoint reference of the service that is originally addressed by the WS-BPEL engine. The URL of this endpoint is used as an index to describe the correct `ServiceDescription` for the current external call. If no `NewServiceAddress` is given, the Workflow Service will still forward the message to this endpoint, but it might be possible that the address is only virtually addressed - used as an identifier.
  - NewServiceAddress (optional): If given, the BIS-Grid engine redirects the message to the endpoint described by the parameter. Note that the WSDLs of both endpoints, the original and the new one, must be identical.
  - Credential (optional): A string that represents the id of a `CredentialDescription` (also a resource property of the Workflow Service).

  - ServiceType (optional): Determines the type of the external service: *UNICORE6*, *WebService*, or *GT4*. The latter option is intended to be supported in future versions of the BIS-Grid engine.

- **CredentialDescription**: If the external service requires authentication or authorization, the corresponding information is stored in a `CredentialDescriptionProperty`.
  - id (required): The id of this description.
  - CredentialType (required): the type of credential information that should be used for calls described by this credential description. The type defines the content of the CredentialDescription as follows:
    * keystore: All necessary information for a keystore and truststore is defined (*File*, *Password*, *Type*, *Alias*). The keystore must be uploaded by an administrator or part of the deployment archive before executing the workflow (future work).
    * usernamepassword - A username and password is defined.
    * samldelegation - An id of an SAML assertion that was send to the service through a client call.

3.5.3 Workflow Factory Service State

As already described in Section 2.6 the Workflow Factory Service is implemented as a WSRF service. Its instances are created by a direct Java call of the Workflow Management Service. The state comprises information that is used as input for the Workflow Service, and information to create instances of the correct Workflow Service:

- **WorkflowServiceName**: The `WorkflowServiceName` identifies the home of the Workflow Service the factory is responsible for.
- **WorkflowName**: The name of the service that represents the workflow in the WS-BPEL Engine.
- **ServiceDescription**: Initial parameter for the creation of new Workflow Service instances (see above).
- **CredentialDescription**: Initial parameter for the creation of new Workflow Service instances (see above).

3.6 Communication from/to WS-BPEL Engine

As already described in the specification [6], the UNICORE extensions hide the complexity of Grid service invocation from the WS-BPEL engine. To be more specific, Figure 11 presents a detailed view on the communication flow of the Workflow Service:
In Section 3.4 we already described the processing of messages that are directed to the Workflow Service (standard and workflow messages). This section addresses the

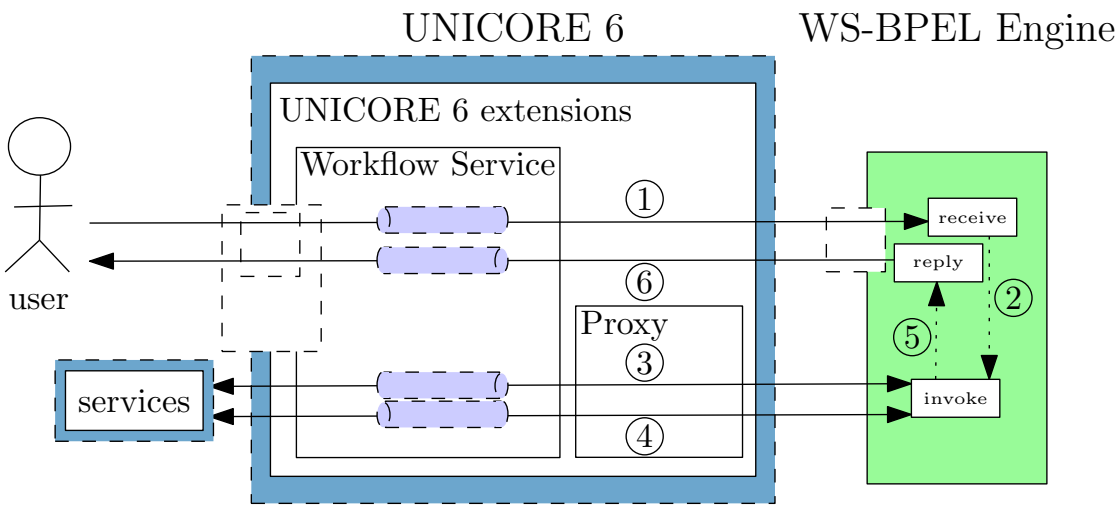

Figure 11: Communication Flow of the BIS-Grid Workflow Service

implementation of the Workflow Service after the method *processBpelMessageCall* is invoked. Figure 11 shows four additional handler pipelines (light blue tubes).

When forwarding a message from a client or external application to the WS-BPEL engine [1], Grid-specific information such as security credentials and accounting and billing information is removed and, if necessary, evaluated in a configurable handler pipeline, thus converting the message into a standard web service call message. The WS-BPEL workflow engine then proesses the message [2].

Vice versa, messages sent from the WS-BPEL workflow to invoke external services [3] are caught by the *proxy*, which is located in the UNICORE service container (UNICORE/X). The proxy reads the name of the workflow and the id of the corresponding Workflow Service instance into the message header, and forwards the message to the correct Workflow Service instance for further processing. By using another handler pipeline that depends on the current configuration of the Workflow Service instance, Grid-specific XML fragments - e.g. SAML assertions [9] - are added to the message and are then forwarded through a certificate-secured SSL channel. In case of a synchronous service call the response is subsequently processed in reverse, removing and processing Grid-specific XML fragments and forwarding it to the WS-BPEL engine through the connection held by the proxy [4]. Regarding asynchronous service calls the connection will be closed after the message is sent. The response message from the external service is also processed by the WS-BPEL engine [5]. If the workflow has a reply activity, the answer is send to the user using another handler pipeline [6].

We distinguish between two communication directions, [1]/[6] and [3]/[4], each implemented by a processor that manages the handler pipeline configuration. The class diagram is shown in Figure 12. Both processors, `Client2BPELEngineProcessor` and `BPELEngine2ExternalServiceProcessor`, are created by the Workflow Service instance that delegates the message processing. The following handlers are already implemeted

handlers that are used in this pipeline:

[1] Client-UNICORE6-WS-BPEL

- **RemoveUnicoreCredentialsHandler**: This handler removes all SAML assertions.
- **RemoveWSAHeaderHandler**: Removes the web service addressing information, e.g. endpoint references, from the SOAP header.
- **IdAssignOutHandler**: Inserts the unique identifier, `WorkflowServiceName` and UNICORE 6 unique instance id, into the message body so that it can be stored in the WS-BPEL engine using the injected WS-BPEL patterns.
- **OutMessageSender**: Serializes and sends the message.

[6] WS-BPEL-UNICORE6-Client

- **ReadHeadersHandler**: Reads the SOAP header and stores it in the message context.
- **BpelCallBodyInHandler**: Reads the SOAP body and stores it in the message context.

[3] WS-BPEL-UNICORE6-ExternalService

- **WSAddressingOutHanlder**: Sets WS-Addressing headers if missing.
- **BpelIndentifierRetrievalOutHandler**: Retrieves the WS-BPEL process identifier from the SOAP header.
- **RemoveBisGridMappingInformationOutHandler**: Removes the mapping information from the header to hide implementation-specific patterns.
- **SoapSerializerHandler**: Serializes the SOAP message into a SOAP envelope.
- **OutMessageSender**: Serializes and sends the message.
- **TDOutHandler** (only if necessary): Adds SAML trust delegation tokens to the message.

[4] ExternalService-UNICORE6-WS-BPEL

- **ReadHeadersHandler**: Reads the SOAP header and stores it in the message context.
- **BpelCallBodyInHandler**: Reads the SOAP body and stores it in the message context.

As described in Section 1.5, additional handlers to [3] and [6] can be added easily using the `bisgrid.properties` configuration file.

The Workflow Service state includes information on `ServiceDescriptions` and `CredentialDescriptions`. This information is also evaluated in the `BPELEngine2ExternalServiceProcessor`.

First, the corresponding `ServiceDescription` is picked from the resource property. With this, we can determine the destination, the type of the external service (*WebService*, *UNICORE6*, *GT4* - the last notbeing supported at the moment), and if needed a `CredentialDescription`. All properties can affect the handler pipeline, for example, by using a secure or non-secure connection. In a second step before sending, the processor evaluates the `CredentialDescription` and inserts handlers, or reconfigures the HTTP client for establishing the connection.

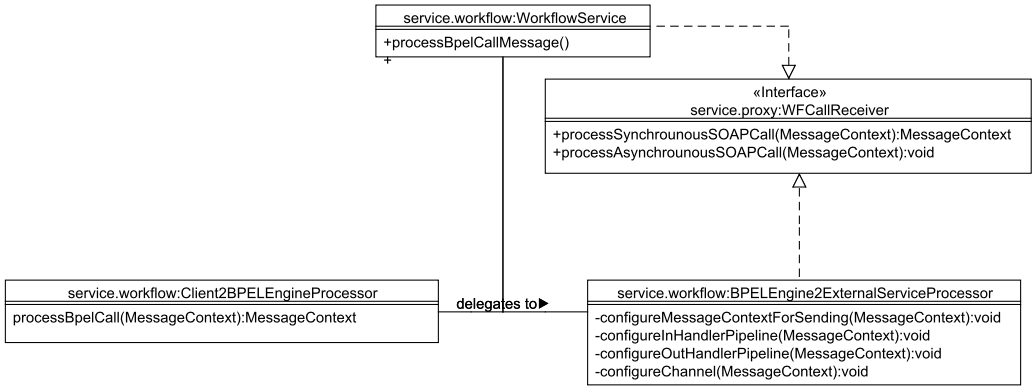

Figure 12: Class diagram - External service call processing

3.7 Monitoring of BPEL Workflows

The BIS-Grid engine supports monitoring running workflow instances via the resource property `WorkflowStatus`. The `WorkflowStatus` is composed of the status of the complete workflow and one state information for each activity in the workflow. Thereby, an activity is defined vy its location path, an XPath expression that uniquely defines the activity in the WS-BPEL process, its executions state, and an optional info string, for example an error message.

To retrieve this information, the WS-BPEL engine must support basic monitoring capabilities because the UNICORE 6 layer is not aware of the execution state of the process or the activities of the process. The `MonitoringAdapter` provides WS-BPEL engine-specific extensions to the BIS-Grid engine to retrieve the execution state. The result of the `getProcessState()` operation of the `MonitoringAdapter` is a `ProcessState` object that contains all necessary information. It can be mapped to the `WorkflowStatus` resource property. A process or a process activity can have one of the following states:

- NOT_KNOWN: The state is unknown or cannot be determined. This state can have different reasons (e. g., the monitoring adapter is not implemented or the workflow is not yet started which means that the monitoring adapter cannot be used since the WS-BPEL process id is not yet available.

- NOT_EXIST: A workflow in this state means that the WS-BPEL process id could not be retrieved. Therewith the Workflow Service cannot find the corresponding WS-BPEL process.
- COMPLETED: The workflow has been completed successfully.
- RUNNING: The workflow is currently active.
- FAULTED: An error occured and the workflow failed.
- SUSPENDED: An administrator has decided to suspend this workflow (this will only be possible if the administrator has direct access to the WS-BPEL engine).

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 13. The structure of the proxy module is shown in Figure 14.

The proxy service is enhanced by a *Redirection Proxy* service to extend the functionality to secure HTTP connections using the HTTPS protocol. A normal HTTPS proxy does not know the content of the request that it forwards. Therefore it cannot analyze the traffic send between the web service client and the web service endpoint to redirect web service requests. To address this, another level of redirection was introduced: The Redirection Proxy. The revised architecture is displayed in Figure 15. The clients sends a HTTPS request to an endpoint (not displayed). Due to the proxy configurations (`https.proxyHost` and `https.proxyPort` in `catalina.properties`) the request is redirected the the Redirection Proxy - e. g., to port 8080. The Redirection Proxy does not forward the request to the endpoint, but opens a connection with the HTTPS port of the Workflow Proxy. This proxy accepts the connection. Since the proxy behaves like an normal endpoint against the client, it now can read the request content unencrypted, analyze it and redirect it towards the Workflow Engine as described above. The HTTPS Redirection Proxy is located in the module `service.proxy.redirect` and is running as independent process. The application entry class is `de.dgrid.bisgrid.services.procy.redirect.HttpRedirect`. The following param-

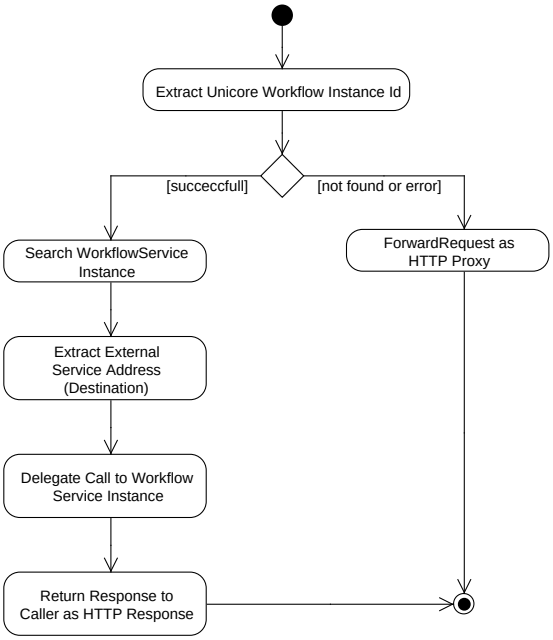

Figure 13: Activity Diagram describing the processing of a HTTP request by the Workflow Proxy

eters are defined:

`-l <port>`: Port number that is listening the HTTP/HTTPS requests (default: 8080).

`-u <hostname>`: Hostname of the Workflow Proxy (default: localhost).

`-up <port>`: Port number of the Workflow Proxy for HTTP traffic (default: 8089).

`-ups <port>`: Port number of the Workflow PRoxy for HTTPS traffic (default: 8090).

`-S <t>`: Server timeout for network communication (default: no timeout).

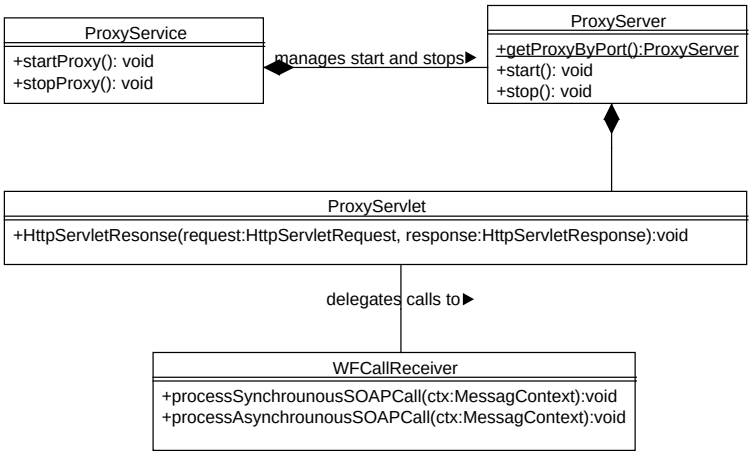

Figure 14: Class Diagram describing the Workflow Proxy

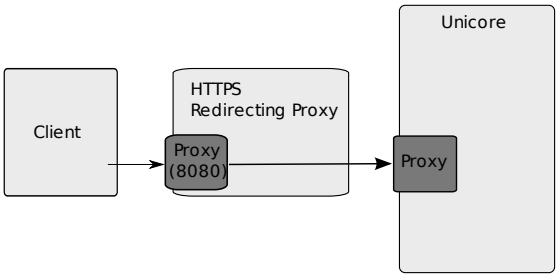

Figure 15: Activity Diagram describing the processing of a HTTPS request by the Redirection Proxy

5 Client Portlets

The BIS-Grid engine is based upon web service technologies resulting in a loosely coupled implementation of several services. The Workflow Management Service is a standard Web/Grid-Service providing a fixed interface for (un)deployment operations. The Workflow Service, or respectively, the concrete hot-deployed occurrences provide a two fold interface: on the one hand the standard WSRF methods and on the other hand workflow specific operations. Hence, it is relatively easy to create a client for the Workflow Management Service but much more harder to implement a generic user friendly client for the Workflow Service.

To demonstrate the BIS-Grid engine we implemented clients based on the Gridsphere framework[8]. Gridsphere offers an API for programming JSR 168-compliant portlets. We created a Workflow Management Portlet that supports all operations of the Workflow Management Service. The user can upload BIS-Grid deployment archives and use all operations like deploy, undeploy, retrieve on the Workflow Management Service. Figure 16 presents a screenshot of the Workflow Management Service portlet.
For Workflow Service instances, we created a client that supports standard WSRF operations such as retrieving resource properties. We regard this client as starting template for extending the Portlet with workflow-specific operations. The disadvantage in using Gridsphere as portlet framework is the currently missing support for the UNICORE 6 security infrastructure. Single sign-on with D-Grid certificates is not supported by Gridsphere in conjunction with UNICORE 6. At the moment we integrated a solution that uses a BIS-Grid Gridsphere certificate for all BIS-Grid engine invocations. This connotes that UNICORE 6 currently cannot distinguish between different users. We intend to examinate the feasibility of extending the portlets with advanced security mechanisms using the UVOS identity management as soon as we set up and configured a UVOS server, and integrated it with the BIS-Grid engine.

[8] http://www.gridsphere.org/gridsphere/gridsphere

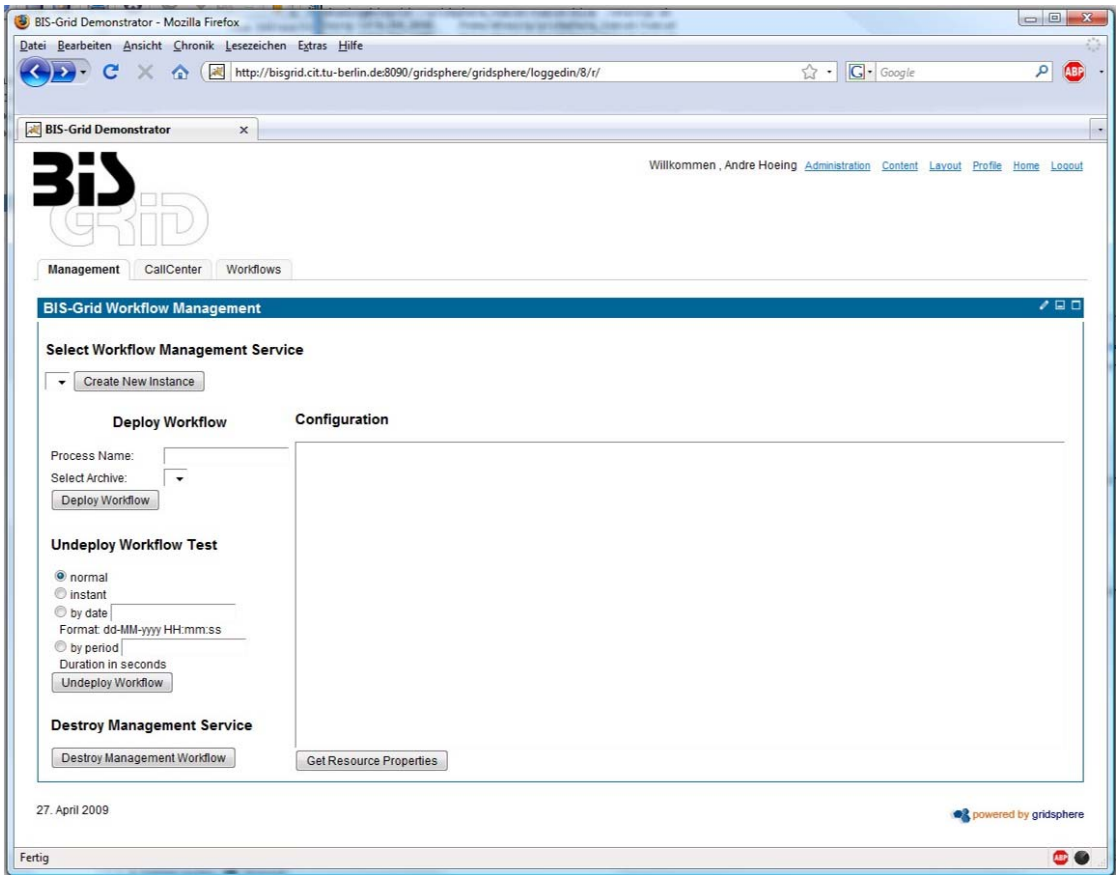

Figure 16: Screenshot Management Portlet

6 Future Work

In this section, we provide a short overview on our planned near-future work on the BIS-Grid engine.

- We intend to introduce mechanisms to support GSI (Grid Security Infrastructure) in order to enable the BIS-Grid engine to use Globus Toolkit 4 Services.
- As described in Deliverable 2.5 [8] we intend to integrate the UVOS VO Management System into the BIS-Grid engine to support fine-grained role based access control. This includes the development of appropriate XACML security policies according to the role model presented in the appendix of the specification [6].
- We intend to extend the deployment descriptor/archive with workflow specific XACML policies, Credentials, etc. (implementing the results of the diploma thesis mentioned in Section 2.2).
- We will extensively test and evaluate the engine in our application scenarios at KIESELSTEIN and CeWe Color, but also within general scenarios such as a generic job submission workflow.

A BIS-Grid deployment descriptor - XML Schema

Listing 4: XML Schema for the BIS-Grid deployment descriptor

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://bisgrid.dgrid.de/services/management/
     deployment"
  xmlns:tns="http://bisgrid.dgrid.de/services/management/
     deployment"
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  elementFormDefault="qualified" xmlns:pref="http://bisgrid.
     dgrid.de/services/workflow/properties/WorkflowService">

  <xsd:import namespace="http://www.w3.org/2005/08/addressing"
    schemaLocation="www.w3.org/2005/08/addressing/ws-addr.xsd" />
  <xsd:import namespace="http://bisgrid.dgrid.de/services/
     workflow/properties/WorkflowService"
    schemaLocation="WorkflowServiceResourceP roperties.xsd" />


  <xsd:element name="process" type="tns:processType" />

  <xsd:complexType name="partnerLinkXSDType">
    <xsd:sequence>
      <xsd:element name="myRole" minOccurs="0" maxOccurs="
         unbounded">
        <xsd:complexType>
          <xsd:attribute name="service" type="xsd:string" use="
             required" />
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="partnerRole" minOccurs="0" maxOccurs="
         unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="wsa:EndpointReference" minOccurs="
               0" maxOccurs="1" />
          </xsd:sequence>
          <xsd:attribute name="endpointReference" type="tns:
             endpointReferenceXSDAttributeType" use="required"
             />
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"
       />
  </xsd:complexType>
```

```
35
    <xsd:simpleType name="endpointReferenceXSDAttributeType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="static" />
        <xsd:enumeration value="dynamic" />
40    </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="processType">
      <xsd:sequence>
45      <xsd:element minOccurs="1" maxOccurs="1" name="
           bpelDescription">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element minOccurs="1" maxOccurs="1" name="
                 location" type="xsd:string" />
            </xsd:sequence>
50        </xsd:complexType>
        </xsd:element>
        <xsd:element minOccurs="1" maxOccurs="1" name="
           partnerLinks">
          <xsd:complexType>
            <xsd:sequence>
55            <xsd:element minOccurs="1" maxOccurs="unbounded"
                 name="partnerLink" type="tns:partnerLinkXSDType"
                  />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element minOccurs="1" maxOccurs="1" name="references"
           >
60        <xsd:complexType>
            <xsd:choice>
              <xsd:element minOccurs="1" maxOccurs="unbounded"
                 name="wsdl">
                <xsd:complexType>
                  <xsd:attribute use="required" name="location"
                     type="xsd:string" />
65                <xsd:attribute use="required" name="type" type="
                     xsd:string" />
                  <xsd:attribute use="required" name="namespace"
                     type="xsd:string" />
                </xsd:complexType>
              </xsd:element>
              <xsd:element minOccurs="1" maxOccurs="unbounded"
                 name="schema">
70              <xsd:complexType>
```

```
                    <xsd:attribute use="required" name="location"
                        type="xsd:string" />
                    <xsd:attribute use="required" name="type" type="
                        xsd:string" />
                    <xsd:attribute use="required" name="namespace"
                        type="xsd:string" />
                  </xsd:complexType>
75              </xsd:element>
             </xsd:choice>
           </xsd:complexType>
         </xsd:element>
         <xsd:element name="serviceDescriptions" type="tns:
            serviceDescriptionsType" maxOccurs="1" minOccurs="1" /
            >
80       <xsd:element name="credentialDescriptions" type="tns:
            credentialDescriptionsType" maxOccurs="1" minOccurs="1
            " />
       </xsd:sequence>
       <xsd:attribute use="required" name="name" type="xsd:string"
          />
     </xsd:complexType>

85   <xsd:complexType name="serviceDescriptionsType">
       <xsd:sequence>
         <xsd:element ref="pref:ServiceDescription" maxOccurs="
            unbounded" minOccurs="0" />
       </xsd:sequence>
     </xsd:complexType>
90
     <xsd:complexType name="credentialDescriptionsType">
       <xsd:sequence>
         <xsd:element ref="pref:CredentialDescription" maxOccurs="
            unbounded" minOccurs="0" />
       </xsd:sequence>
95   </xsd:complexType>
</xsd:schema>
```

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] A. Brinkmann, S. Gudenkauf, W. Hasselbring, A. Hoeing, O. Kao, H. Karl, H. Nitsche, and G. Scherp. Employing WS-BPEL Design Patterns for Grid Service Orchestration using a Standard WS-BPEL Engine and a Grid Middleware. In *CGW'08 Proceedings*, Krakow, Poland, 2009.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] Andre Höing, Stefan Gudenkauf, Guido Scherp, Holger Nitsche, and Dirk Meister. BIS-Grid Deliverable 2.5: Spezifikation der Anforderungen an Sicherheit und Service Level Agreements im Zusammenhang mit WS-BPEL. Technical report, BIS-Grid, March 2009.

[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.