WSAMI Middleware Architecture Guide

Author: Daniele Sacchetti – daniele.sacchetti@inria.fr

Table of contents

LIST OF FIGURES

1. Introduction

The objective of the WSAMI Distributed Middleware Infrastructure is to extend the Web Services architecture (WSA) towards mobile distributed systems, addressing in particular:

- dynamic service composition
- service provisioning over resource-constrained mobile nodes with unstable network connectivity

The WSA's standards on which WSAMI Middleware is based are:

- **WSDL (Web Services Description Language)** that is a language based on XML that is proposed by the W3C for describing the interfaces of Web Services
- **SOAP (Simple Object Access Protocol)** that defines a lightweight protocol for information exchange

WSA is further conveniently complemented by **UDDI (Universal Description,**

Discovery and Integration) that is a specification of a registry for dynamically locating and advertising Web Services.

The WSAMI Distributed Middleware Infrastructure provides a platform based on WSAMI language and some middleware-related services providing essential functionalities to allow the users to develop and run their own application-related services on the top of the WSAMI Middleware.

We introduce the WSAMI (Web Services for AMbient Intelligence) declarative language, based on WSDL, for the specification of both application- and middleware-related services (see section 2 for more details about WSAMI language). Given the WSAMI specification of a service, an instance is automatically composed upon a user request, according to the services (instances) that may be retrieved from the environment, regarding in particular network connectivity.
The WSAMI Middleware architecture is based on the following middleware-related services and can be summed up by Figure 1:

Core middleware infrastructure
offering a Core Broker and the WSAMI language for the declarative specification of services, for enabling interaction among services distributed over heterogeneous terminals (see 3 for details about Core Broker).

Service for naming, discovery and lookup (ND Service)
permitting the retrieval of a service both in the local and in the wide area, according to the service specification (in terms of either a name or a declarative specification) and to the environment in which the service is requested (see 4 for details about Naming and Discovery Service).

We introduce some further middleware-related services:

Universal Repository (UR) Service
as in WSA specification, this service is used as a registry for dynamically locating and advertising application-related services, but in our implementation instead of the UDDI specification, we have adapted this service to WSAMI Middleware specific requirements (see 5 for details about UR Service).

Gateway Service
permitting a multihomed station connected both to an infrastructure-based network and to a wireless ad-hoc network to enable the retrieval of services on the two networks through the ND Service.
It further allows all the stations on the wireless ad-hoc network to be connected to the infrastructure-based network and to have an interaction with services running outside the ad-hoc network (see 6 details about Gateway Service).

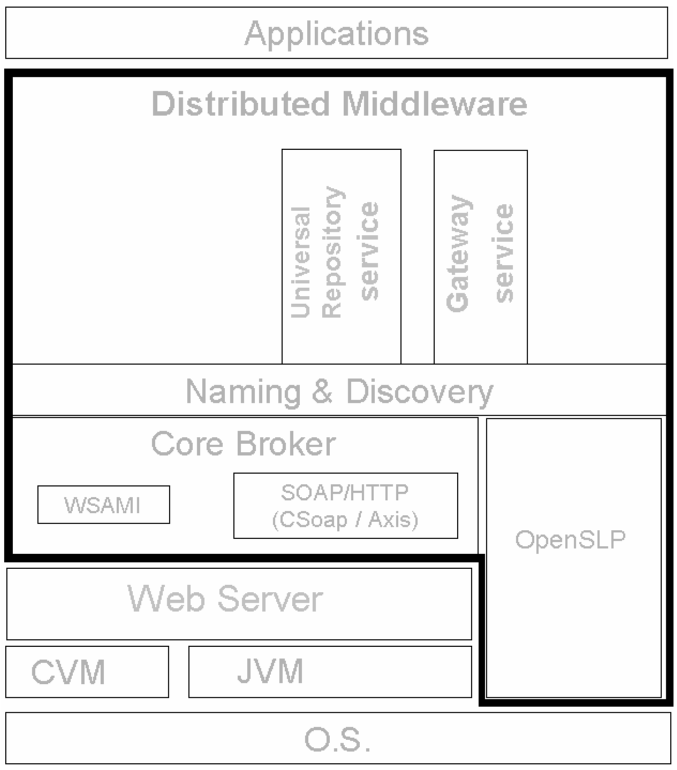

Figure 1 WSAMI Middleware Architecture

In this document we describe the architecture of the WSAMI Middleware and we further detail the implementation with UML notation. In chapter 2 the WSAMI language is defined, in chapter 3 we describe the Core Broker, in chapter 4, the Naming and Discovery service, in chapter 5, the Universal Repository (UR) service and in chapter 6, the Gateway service.

2. WSAMI

In the WSAMI Middleware each service is defined by three WSAMI (an XML-based language) documents (you can see the XSD specification of these documents in section 7.1). We list below these documents and we further detail the information contained in each one. The third one, the WSAMI Customization Document is optional and must be defined only if the service requires the customization feature (for details see section 3.4).

1. **WSAMI Abstract Document (WAD):** defines the Service Abstract Interface
    - **Interface**: the URI of the WSDL document defining only the abstract interface of the service.

- **Conversation**: the URI of the WSCL document defining the interaction protocol with the service[1].
- **ServiceQoS:** the non-functional property associated with the service in terms of QoS criteria (e.g., transactionalService, security, saveBandwidth).
- **ConnectorQoS:** the non-functional property associated with the connector in terms of QoS criteria (e.g, transactionalService, security, saveBandwidth).

2. **WSAMI Service Document (WSD):** defines the Service Concrete Instance
   - **Abstract:** the URI of the associated WSAMI Abstract Document.
   - **Concrete:** the related WSDL document, which includes concrete binding information.
   - **Required:** the set of required services identified with the URI of their WSAMI Abstract documents.

3. **WSAMI Customizer Document (WCD):** defines the service customization connector
   - **QosCriterion**: the specific QoS criteria that are enforced by the embedding customizer (e.g, transactionalService, security, saveBandwidth).
   - **Local**: the WSAMI Abstract URI of the local customizer service.
   - **Remote**: the WSAMI Abstract URI of the remote customizer service.
   - **Colocation**: true if local customizer service must be running on the same node as the client and the remote customizer service on the same node as the service, false otherwise.

For each of these documents we introduce some further data associated with the service and used by WSAMI Middleware to manage services: the document URI locations, named respectively WAU, WSU and WCU that are the URI addresses where the above documents are available.

Two abstract interfaces are said to match if their respective documents are syntactically equal. This allows us to keep to a minimum the processing cost associated with checking specification matching: two abstract interfaces match if the related WSAMI documents have the same URI (WAU). This may be considered as quite restrictive since it does not take into account service instances whose WSDL abstract interface matches syntactically the one associated with the requested service. However, our solution enables an efficient retrieval process that is solely based on comparing URIs. It further allows enforcing that

[1] The Conversation document definition will be integrated to the WSAMI Middleware when a W3C standard will be available

the abstract interfaces that are respectively associated with a requested service and a retrieved service instance do match.

The WSAMI documents described above are represented by `WSAMIAbstractDocument`, `WSAMIServiceDocument` and `WSAMICustomizerDocument` classes (see Figure 2).

`WSAMIDocument` is the base class implementing the main functionalities, that is, to create a WSAMI file from the memory representation and to build a memory representation of the document from the WSAMI file.

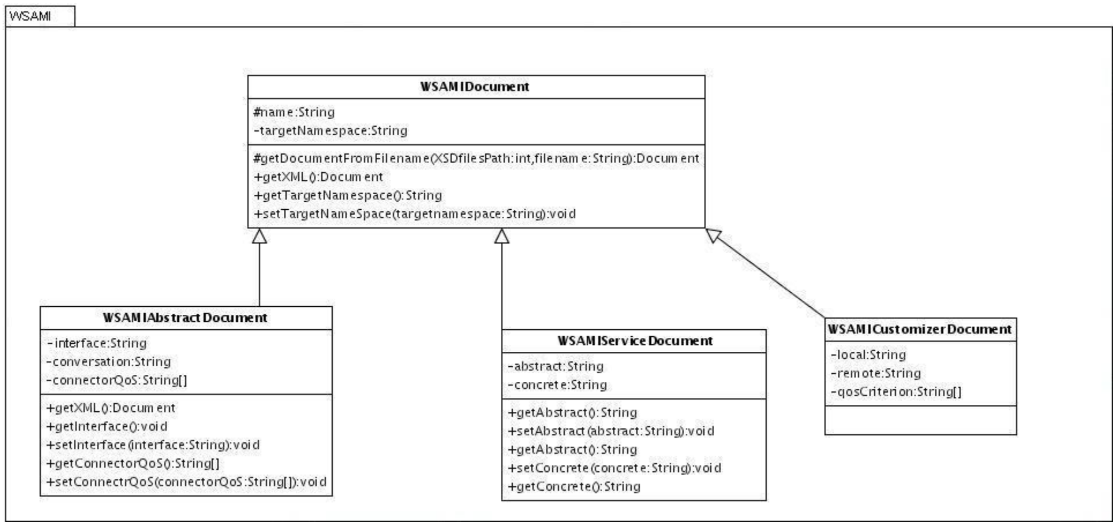

Figure 2 WSAMI class diagram

For the former functionality the `getXML` method is provided by all these classes, while for the latter one, the `WSAMIDocument.getDocumentFromFileName` method is used by `WSAMIAbstractDocument`, `WSAMIServiceDocument` and `WSAMICustomizerDocument` class constructors to obtain an object `Document` representing the WSAMI file. This `Document` object is then used to retrieve the information required to build the `WSAMIDocument` object and fill its data fields (for example `setAbstract()` and `setConcrete()` for `WSAMIServiceDocument` objects).

The implementation of these classes is based on Xerces2 Java Parser. The XML parsers can be divided into two major groups:

- *Tree-based APIs:* map an XML document into an internal tree structure, then allow an application to navigate that tree. The Document Object Model (DOM) working group at the World-Wide Web Consortium (W3C) maintains a recommended tree-based API for XML and HTML documents, and there are many such APIs from other sources.
- *Event-based APIs:* reports parsing events (such as the start and end of elements)

directly to the application through callbacks, and does not usually build an internal tree. The application implements handlers to deal with the different events, much like handling events in a graphical user interface. SAX is the best known example of such an API.

In our implementation we make use of the first type of parsers, based on the DOM standard.

The method `getXML` is based on the default DOM-based parser provided by `DocumentBuilderFactory` and `DocumentBuilder` classes of the Xerces library to build an `org.w3c.dom.Document` object from the values of the class attributes.

The `WSAMIDocument.getDocumentFromFileName` makes use of the DOM-based `org.apache.xerces.parsers.DOMParser` parser to parse the WSAMI files into an `org.w3c.dom.Document` object.

Each WSAMI document contains a reference to the XSD file that defines its XML schema (WSAMI.xsd, see section 7.1). You can see some examples of WSAMI files in sections 4.4, 5.1 and 6.1. The parser offers the option to verify if the WSAMI file matches the XSD specification and in our implementation we enables this feature by setting the values of the properties `http://xml.org/sax/features/validation` and `http://apache.org/xml/features/validation/schema` to true. The parser will validate the WSAMI document against XSD file and raise an error if it does not match the specification. If no errors are raised, the method `getDocumentFromFileName` can build the `Document` object.

3. Core Broker

As you can see in Figure 1, the WSAMI Middleware is based on Java and Web Services technologies. For details about the system requirements in terms of Java VM for PDAs and PCs see [USERGUIDE].

The version of Core Broker for PC stations (in this guide, the term PC refers to an Intel x86 architecture) is based on the Web container Jakarta Tomcat and the Axis SOAP implementation. The implementation of Axis has been included in the provided WSAMI Middleware (see Figure 3).

The version of Core Broker for PDA devices (in this guide, the term PDA refers to a Strong-ARM architecture) is based on the Web container Jetty and the CSoap SOAP implementation (see [CSOAP-GUIDE] and [CSOAP-ARCH]). The implementation of CSoap and Jetty has been included in the provided WSAMI Middleware (see Figure 3) and a minimal version of Jetty web server supporting only servlet and no JSP is provided to minimize the resource consumption.

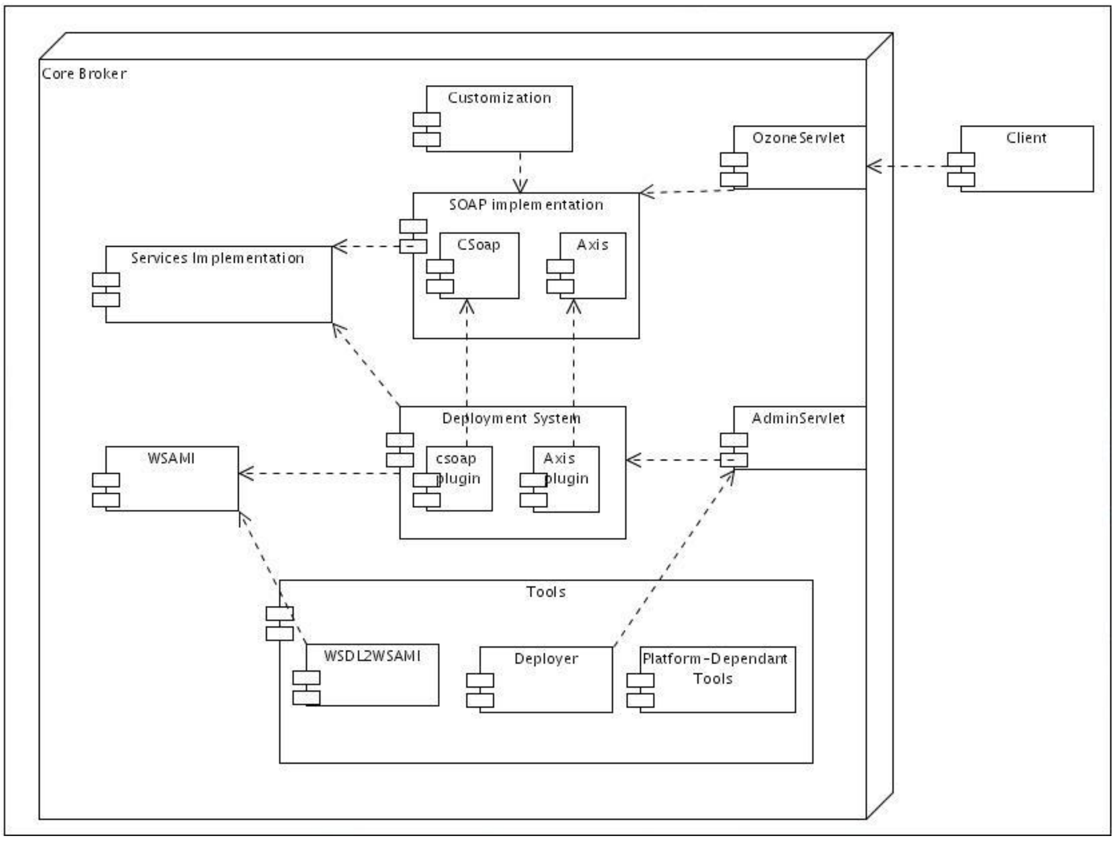

Figure 3 Core Broker components

In Figure 3 you can see the functional components of Core Broker. The arrows connecting the components mean a dependency of the source component on the destination component:

- The `AdminServlet` is the entry point available on the web server and waiting for requests to (un)deploy services and depends on the Deployment System. For more details see Section 3.1.
- The `WSAMIServlet` is the entry point available on the web server for clients of services and waits for method calls addressed to services and its role is to perform the execution of services and reply to clients. For more details see Section 3.3.
- The `Deployer` has the role to send deployment information to `AdminServlet` and is used by users to deploy their services. For more details see Section 3.1.
- The Deployment System is in charge of the management of the service repository (WSAMI and WSDL files associated, endpoint, customization, implementation files) and of the deployment of services on the SOAP implementation module that will have the specific role to execute the services when a method call is received through the `WSAMIServlet`. For more details see Section 3.1.

- The WSAMI module's role is to manage WSAMI documents. For more details see Section 2.
- The customization module's role is to handle the service execution customization. For more details see Section 3.4.
- The tools module provides some functionalities used by Core Broker, Naming and Discovery and Gateway services. For more details see Section 3.5.
- SOAP implementation module consists of the libraries including the implementation of Axis and CSoap. For more details see Section 3.3.
- SOAP plug-ins module is required by deployment component to deploy a service on the specific SOAP implementation engine (see section 3.1), but also to make calls to services running on the specific SOAP engine (see section 3.3) and finally to enable service execution customization (see section 3.4).

The Core Broker defines two contexts inside the Web server it is running on: the `wsami` context and the `services` context. The former contains the `AdminServlet` and is used to store all the service's WSAMI and WSDL files and the Core Broker configuration files (XSD files, `wsami.xml` and `wsamiInstalledServices.xml`), while the latter contains the `WSAMIServlet` and is used to store the services' implementation files. For more information about the configuration of these contexts see [USERGUIDE].

3.1. Deployment System

The system shown in Figure 3 and described above is required to deploy services and let them be executed when a method call is received by the system. A client application that must access a service does not need the whole system, but only a limited set of components of the Core Broker. We can identify two different types of instance of the Core Broker: server side and client side.

For each node where an instance of the WSAMI Middleware is running, only one instance of the Core Broker server-side must be in execution to avoid incoherence in services' definitions (for service deployment and method call execution). With this solution all the deployment command and all the method calls to services will always be managed by the same Core Broker instance.

Another requirement both for client and server side is that an application (both client and Application-related or Middleware-related service – i.e., Naming & Discovery, Gateway and UR services) must not manage the type of Core Broker (PDA or PC) version it is running on and it must not be aware of the underlying SOAP implementation.

The classes that implement the Deployment System are shown in Figure 4. The class `CoreBroker.BrokerWrapper i`mplements the main functions that are independent from the underlying SOAP implementation and defines the methods related to the specific SOAP implementation as abstract methods (for example: `makeClientInstance,` `makeServerInstance, deployService)`.

The concrete implementations of this class, provided by SOAP plug-ins are the two classes `CoreBroker.CSoap.CSoapBrokerWrapper` and `CoreBroker.Axis.AxisBrokerWrapper`.

These classes provide a SOAP-specific implementation of the abstract methods of `CoreBroker.BrokerWrapper` related the SOAP engine they represent. Since their interfaces are the same as `CoreBroker.BrokerWrapper`, the applications will make calls to methods of `CoreBroker.BrokerWrapper` without any knowledge of the real class implementing the interface.

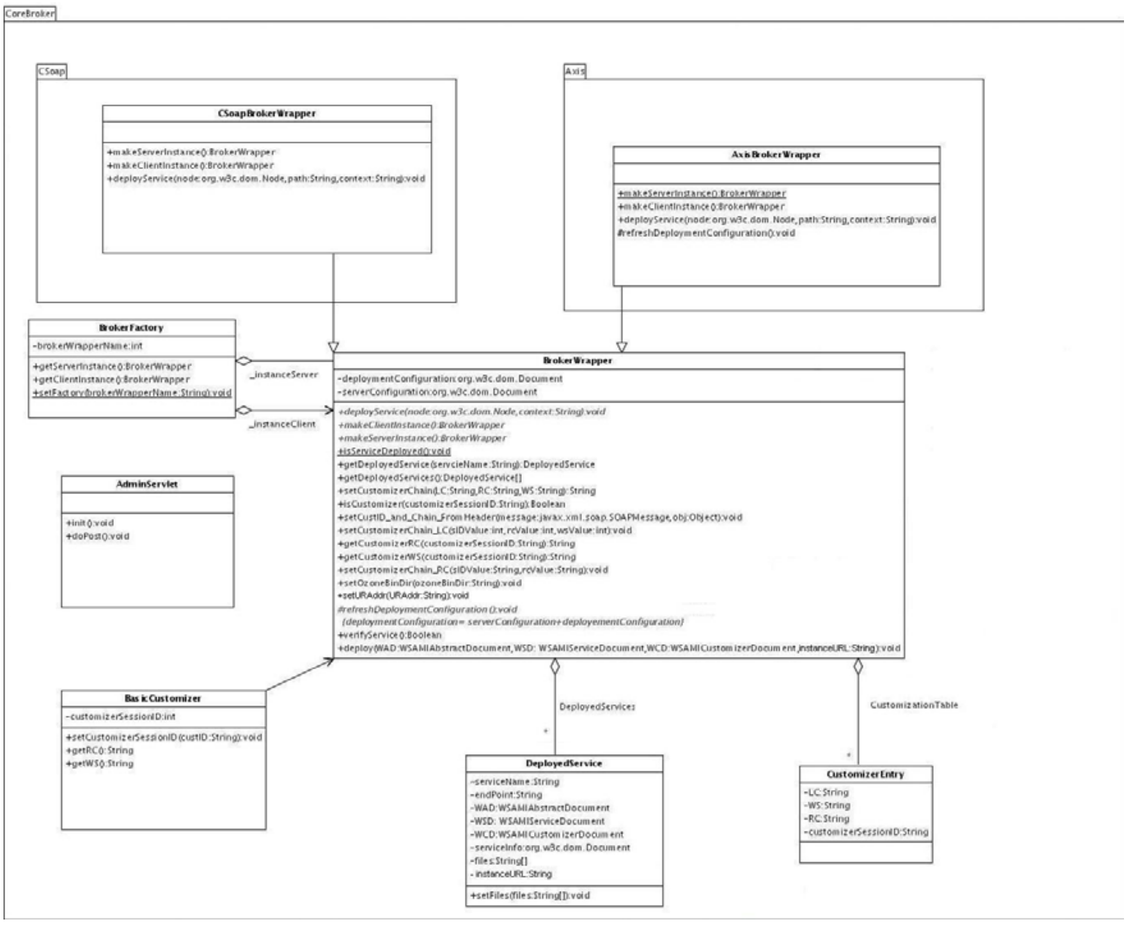
CoreBroker
CSoap
CSoapBrokerWrapper
+makeServerInstance():BrokerWrapper
+makeClientInstance():BrokerWrapper
+deployService(node:org.w3c.dom.Node,path:String,context:String):void
Axis
AxisBrokerWrapper
+makeServerInstance():BrokerWrapper
+makeClientInstance():BrokerWrapper
+deployService(node:org.w3c.dom.Node,path:String,context:String):void
#refreshDeploymentConfiguration():void
BrokerFactory
-brokerWrapperName:int
+getServerInstance():BrokerWrapper
+getClientInstance():BrokerWrapper
+setFactory(brokerWrapperName:String):void
_instanceServer
_instanceClient
BrokerWrapper
-deploymentConfiguration:org.w3c.dom.Document
-serverConfiguration:org.w3c.dom.Document
+deployService(node:org.w3c.dom.Node,context:String):void
+makeClientInstance():BrokerWrapper
+makeServerInstance():BrokerWrapper
+isServiceDeployed():void
+getDeployedService(servcieName:String):DeployedService
+getDeployedServices():DeployedService[]
+setCustomizerChain(LC:String,RC:String,WS:String):String
+isCustomizer(customizerSessionID:String):Boolean
+setCustID_and_Chain_FromHeader(message:javax.xml.soap.SOAPMessage,obj:Object):void
+setCustomizerChain_LC(sIDValue:int,rcValue:int,wsValue:int):void
+getCustomizerRC(customizerSessionID:String):String
+getCustomizerWS(customizerSessionID:String):String
+setCustomizerChain_RC(sIDValue:String,rcValue:String):void
+setOzoneBinDir(ozoneBinDir:String):void
+setURAddr(URAddr:String):void
#refreshDeploymentConfiguration():void
{deploymentConfiguration= serverConfiguration+deployementConfiguration}
+verifyService():Boolean
+deploy(WAD:WSAMIAbstractDocument,WSD: WSAMIServiceDocument,WCD:WSAMICustomizerDocument,instanceURL:String):void
AdminServlet
+init():void
+doPost():void
BasicCustomizer
-customizerSessionID:int
+setCustomizerSessionID(custID:String):void
+getRC():String
+getWS():String
DeployedServices
*
DeployedService
-serviceName:String
-endPoint:String
-WAD:WSAMIAbstractDocument
-WSD: WSAMIServiceDocument
-WCD:WSAMICustomizerDocument
-serviceInfo:org.w3c.dom.Document
-files:String[]
- instanceURL:String
+setFiles(files:String[]):void
CustomizationTable
*
CustomizerEntry
-LC:String
-WS:String
-RC:String
-customizerSessionID:String

Figure 4 Core Broker class diagram

To avoid the creation of multiple instances of classes implementing `CoreBroker.BrokerWrapper` on the same node, according to the ideas described above, the constructor of these two classes are private, so that the only way to create an instance is to call the methods `makeClientInstance` (for a client-side instance) and `makeServerInstance` (for a server-side instance)

To let the developer write code that can make calls to the Core Broker transparently without any knowledge about the type of underlying Broker (for PDA or for PC) we introduce a factory class, `CoreBroker.BrokerFactory`, that has the role to manage the type of Broker and allow the application to get an instance of the Core Broker without requiring the application to be aware of the underlying platform.

The idea behind the abstract factory is that once the type of Broker has been set at the beginning of the system startup (passing the name of the class `BrokerWrapper` to the factory) you can then ask the factory for an instance of the class implementing `BrokerWrapper` so that the management of the creation of `Brokerwrapper` instance for the local node is centralized to solve the two problems explained above.

In sequence diagram in Figure 5 you can see a more detailed view of the sequence of interactions between the classes just described.

The action that is detailed is the startup phase, when the `AdminServlet` is executed for the first time and it creates the instance of the Core Broker that will be used throughout the execution of the system.

The first call to the factory `CoreBroker.BrokerFactory` is the `setFactory` method that is used to set the type of `BrokerWrapper` we want to use in relation of the platform the system is running on.

To make the code of `AdminServlet` independent of the platform, the value of this parameter is retrieved from the configuration file. (For details see startup section 3.2). The `BrokerWrapper` value is set in `BrokerFactory` (`_instanceServer`) and will be used to return all the following instances of `BrokerWrapper`.

The next action is the call to `getServerInstance`, that using the name of class `BrokerWrapper` already set and the reflection, can make a call to `makeServerInstance` method. This method will use the class constructor (that is private, as already said) to create a Core Broker instance.

In the example shown in figure we refer to the case of a Broker based on Axis. The constructor of `CoreBroker.Axis.AxisBrokerWrapper` initializes the structures for the local service repository and for customization. Then the `server-config.wsdd` file (provided by SOAP plug-in) is parsed and used to initialize the SOAP implementation (Axis) wrapped by the Core Broker.

The content of this file will be stored in `CoreBroker.Axis.AxisBrokerWrapper` class instance (`ServerConfiguration`) and used by `refreshDeploymentConfiguration` method to build `deploymentConfiguration`, used to initialize, when required, the underlying SOAP implementation engine (see section 3.3).

The instance of `CoreBroker.Axis.AxisBrokerWrapper` is finally stored in `BrokerFactory._instanceServer` and at each following call to the method `getServerInstance` it will be returned without requiring the execution of the initialization process just described.

For the creation of a client-side instance of Core Broker, the only difference is that the method used is `getClientInstance`, but the Broker used is the same set by `setFactory`.

This method will make use of the reflection to call `makeClientInstance` instead of `makeServerInstance`.

This method does not require the initialization of the local service repository and does not read the file `server-config.wsdd` because it does not require the configuration with the `serverConfiguration` and `deploymentConfiguration` variable member.

It will finally store in `BrokerFactory._instanceClient` and at each following call to the method `getClientInstance` it will be returned without requiring the execution of the initialization process.

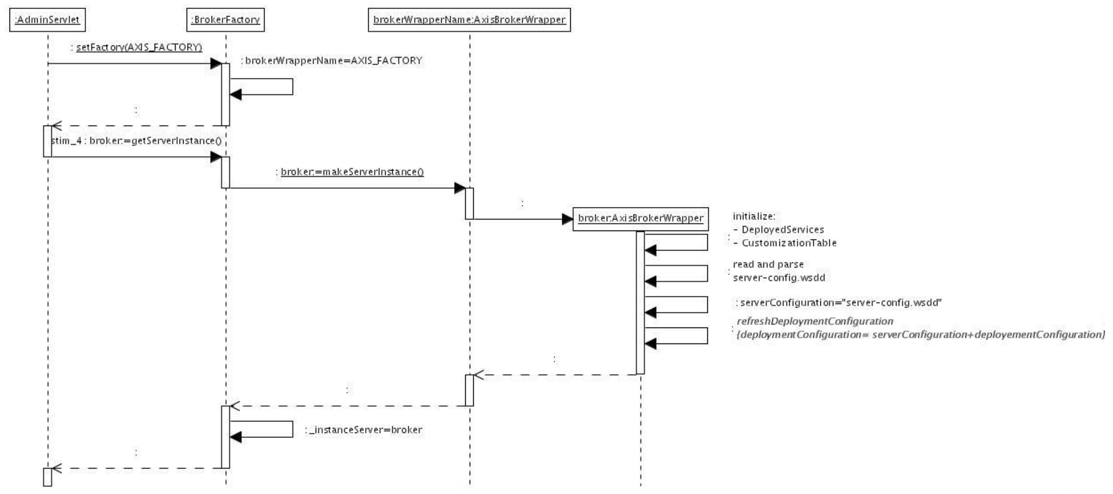
:AdminServlet
:BrokerFactory
brokerWrapperName:AxisBrokerWrapper
: setFactory(AXIS_FACTORY)
: brokerWrapperName=AXIS_FACTORY
stim_4 : broker:=getServerInstance()
: broker:=makeServerInstance()
broker:AxisBrokerWrapper
initialize:
- DeployedServices
- CustomizationTable
read and parse
server-config.wsdd
: serverConfiguration="server-config.wsdd"
refreshDeploymentConfiguration
{deploymentConfiguration= serverConfiguration+deployementConfiguration}
: _instanceServer=broker

Figure 5 BrokerFactory and BrokerWrapper initialization sequence diagram

Service Deployment

The Core Broker contains a repository of the locally deployed services in the data structure `DeployedService` that is a `Hashmap` of `DeployedService`. The `DeployedService` class stores all the information about a deployed service:

- the name used to identify the service inside the Core Broker (`serviceName`)
- the endpoint where it is running (`endPoint`)
- the WSAMI documents defining the service (`WAD ,WSD, WCD`)
- the definition of the service used to deploy it on the specific SOAP engine the Core Broker is running on (`serviceInfo`)
- the list of files that define (.wsdl and .wsami files) and implement the service (.class files) (`files`)
- the URL of the service instance (`instanceURL`). This is an optional value, but if defined, the specific service instance will be identified by the value of this field. All the services deployed with the same `instanceURL` value will be considered by

WSAMI Middleware as the same service instance and will be interchangeable: they will have the same WAU identifier and the same implementation.

The Core Broker also makes a copy of this information on the file system where the two files `wsami.xml` and `wsamiInstalledServices.xml` are kept in a coherent state with the memory representation. When a service is deployed or undeployed both the `DeployedServices` data structure and the two files are updated. At system startup the files are read and the `DeployedServices` structure is initialized to be coherent with the content of the two files.

`wsami.xml` follows the xsd format of `deploy.xsd` (see section 7.2 for more details). It includes the description of each deployed service: service name, identifier (WAU), associated WSAMI files (WAD, WSD and WCD), implementing class and scope.

`wsamiInstalledServices.xml` follows the xsd format of `installed.xsd` (see section 7.4 for more details). It includes the list of files .class that implement and of files WSAMI and WSDL that define each deployed service.

We see now the details about the deployment of a service on the Core Broker.

The deployment requires the interaction of two components of the Core Broker architecture: `Deployer` and `AdminServlet`.

In Figure 6 you can see some details of the `Deployer` sequence diagram. The service XML deploy file generated by tool `WSDL2WSAMI` and matching the `deploy.xsd` definition is modified by adding the list of the WSAMI, WSDL files and all the jar files implementing the service. Each file added to the XML deploy file is just a reference to a file on the local file system, not the actual content of the file, so with the current solution for deployment it is not possible to make a remote service deployment. The new XML deployment file (that matches the `install_deploy.xsd` definition) is finally sent at the local address where the `AdminServlet` is running.

For details about the implementation of `WSDL2WSAMI` & `Deployer` tools see section 3.5.2.

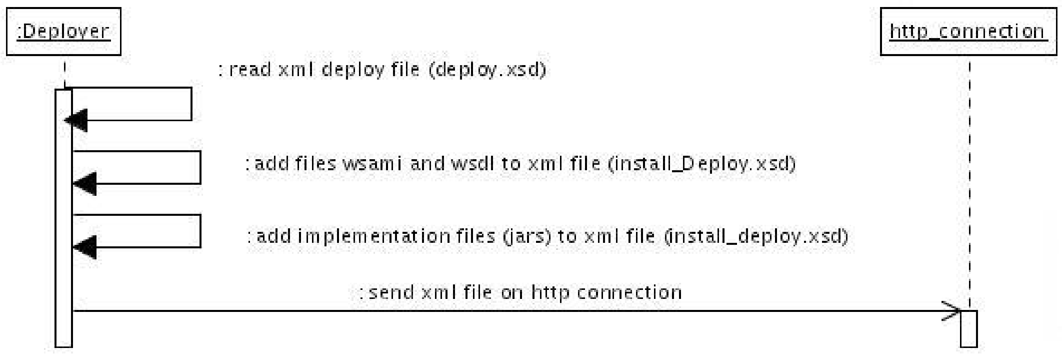
:Deployer
http_connection
: read xml deploy file (deploy.xsd)
: add files wsami and wsdl to xml file (install_Deploy.xsd)
: add implementation files (jars) to xml file (install_deploy.xsd)
: send xml file on http connection

Figure 6 Deployer sequence diagram

The `AdminServlet` runs on the web server and waits for deployment messages containing the deployment document from the `Deployer` tool.

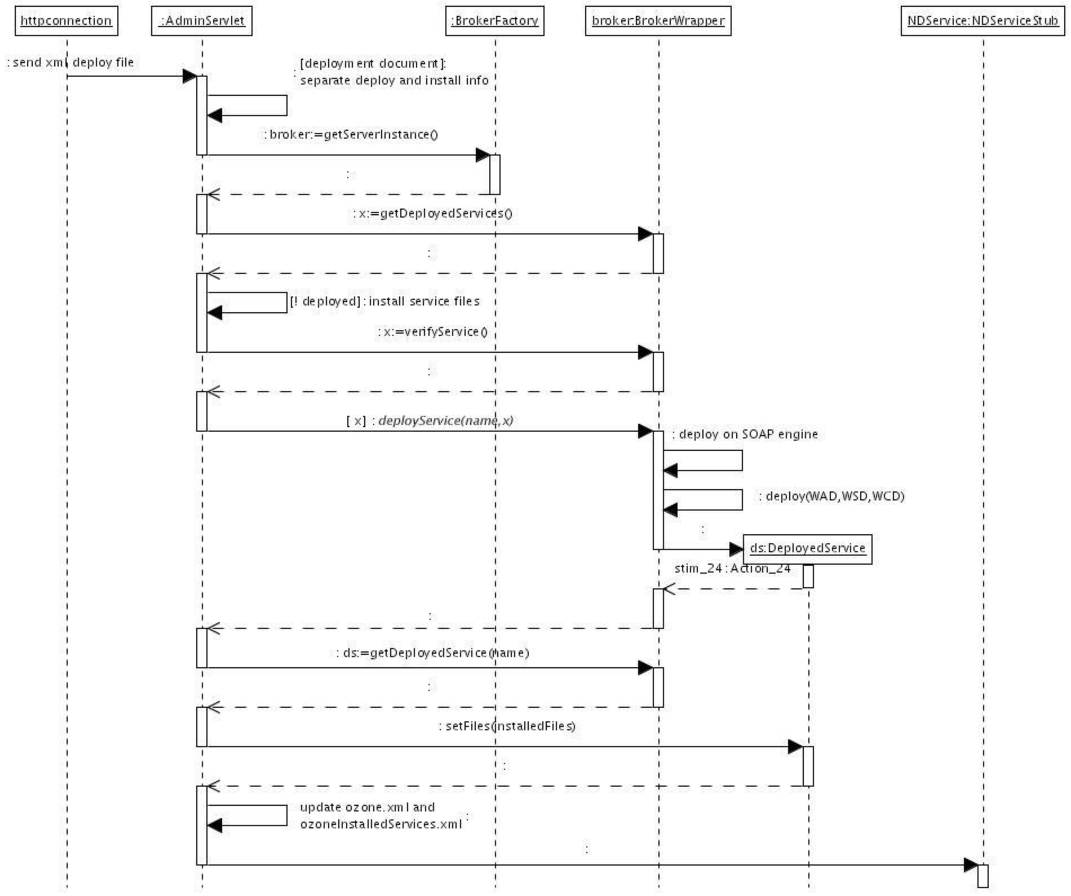
httpconnection
:AdminServlet
:BrokerFactory
broker:BrokerWrapper
NDService:NDServiceStub
: send xml deploy file
[deployment document]: separate deploy and install info
: broker:=getServerInstance()
: x:=getDeployedServices()
[! deployed] : install service files
: x:=verifyService()
[x] : deployService(name,x)
: deploy on SOAP engine
: deploy(WAD,WSD,WCD)
ds:DeployedService
stim_24 : Action_24
: ds:=getDeployedService(name)
: setFiles(installedFiles)
update ozone.xml and ozoneInstalledServices.xml

Figure 7 Service deployment sequence diagram

In Figure 7 we give a detailed view of the service deployment operation:

- The servlet running at the address http://localhost:8080/wsami/AdminServlet receives the deployment document (matching the `deploy_install.xsd` definition file – see section 7.3)

  The sent file can contain the definition of one or more services to deploy/undeploy following the specification of deploy_install.xsd.

- For each service contained in the deployment document, the following operations from 1 to 5 are repeated:

  The first operation is to verify if the service is already deployed on the instance of `BrokerWrapper` running on the node. If it is not already deployed:

  1. Installation of the files listed in the received deployment document. As file references are transmitted as locations in the local file system, this step consists in

copying files in the local file system from the original locations to the right directories so that the Core Broker can make them available:

   - The WSAMI and WSDL files are copied in `wsami` context
   - Jar files are unzipped in `service` context

2. The description of the service contained in the received deployment document and the content of the files describing the service are verified:

   - Verify if WSAMI Abstract Document (WAD), WSAMI Service Document (WSD) and WSAMI Customizer Document (WCD) defined in the deployment document are available on the local file system at the right locations.
   - Verify if the WSDL file defining the service interface file is available on the local file system (or at the same remote http address - not implemented yet) at the same location as specified in the WAD.

     Verify if the WSDL file defining the concrete endpoint location of the service is available on the local file system at the same location as specified in the WSD.
   - Verify if the value of the WSAMI Abstract URI specified in WSD is the same as that specified in the deployment document.
   - Verify the compatibility of the QoS criterions specified in WCD and in WAD.
   - Verify if the endpoint specified in the WSDL concrete file match with the actual endpoint on the SOAP engine after the service deployment.

3. Service deployment:

   - The method `BrokerWrapper.deploy` is called on the right instance of Core Broker with respect to the initialization seen in Figure 5. This method is in charge of the deployment of the service on the engine on the top of which the Core Broker is running. This consists in the generation of the engine deployment file:

     - `CoreBroker.CSoap.CSoapBrokerWrapper` will use all the information in deployment document to generate a WSDD CSoap deployment file (see [CSOAP-ARCH]) with the class provided by the CSoap engine `Csoap.wsdl.generator`.
     - `CoreBroker.Axis.AxisBrokerWrapper` will use all the information in deployment document to generate a WSDD Axis deployment file (see Axis User's Guide) with the class provided by the Axis plug-in `CoreBroker.Axis.WSDL2WSDD` and based on a class provided by Axis engine to generate a WSDD deployment document from a WSDL file. The WSDD deployment document will be stored in `DeployedService.serviceInfo` field and used to build the `deploymentConfiguration` contained in `BrokerWrapper`.

   - The `BrokerWrapper.deploy` method is called to store all the engine independent information about the deployed service on the Core Broker in a

`DeployedServices` data structure.

- If the service is deployed without any mistakes in steps 2 and 3, the list of files associated to the service and copied to the local file system in step 1 is updated in the `DeployedServices` structure associated to the service with the method `DeployedService.setFiles`.

4. Files `wsami.xml` and `wsamiInstalledServices.xml` are updated with the new service.
5. The service is register on the Naming and Discovery service (see section 4).

Service Undeployment

The servlet running at the address http://localhost:8080/wsami/AdminServlet receives the undeployment document (matching the `deploy_install.xsd` definition file) that can contain the definition of one or more services to undeploy.

For each service contained in the undeployment document, the following operations from 1 to 3 are repeated. The first operation is to verify if the service is deployed on the instance of `BrokerWrapper` running on the node. If it is deployed:

1. Delete all files associated to the service. This information can be obtained with the method `DeployedService.getFiles`.
2. Undeploy the service from the Core Broker calling the `BrokerWrapper.undeploy` method that removes the `DeployedService` entry associated to the service from the list of services deployed on the Core Broker.
3. The definition of the service is removed from `wsamiInstalledServices.xml` and `wsami.xml` files.

Service Redeployment

The service redeployment is not supported by WSAMI Middleware. If you want to redeploy a service by undeploying it and deploying a new implementation based on Java classes with the same names, you must restart the system after having undeployed the old service or after having deployed the new one. After these operations the new service implementation will be used instead of the old one.

This limitation of the WSAMI Middleware lies in the following reasons. In Jakarta Tomcat, a Java class loader is associated to each context. This class loader is in charge of loading all the Java classes required for the execution of the context. A `reload` option can be set to `true` for a context and in this case, the context class loader can detect when a class implementation in the context is changed and will unload the old class implementation and load the new one. But because of the JVM implementation, this is not possible without destroying the class loader and reinitializing it, that also involves destroying all the class instances handled by the class loader.

In our case, the `services` context includes the `WSAMIServlet` and all the deployed service instances (see description of Figure 3) and if one of the classes implementing a service was modified, the `services` context class loader would destroy the `WSAMIServlet` instance and all the currently deployed and running instances of services that would lose their internal state.

3.2. System startup

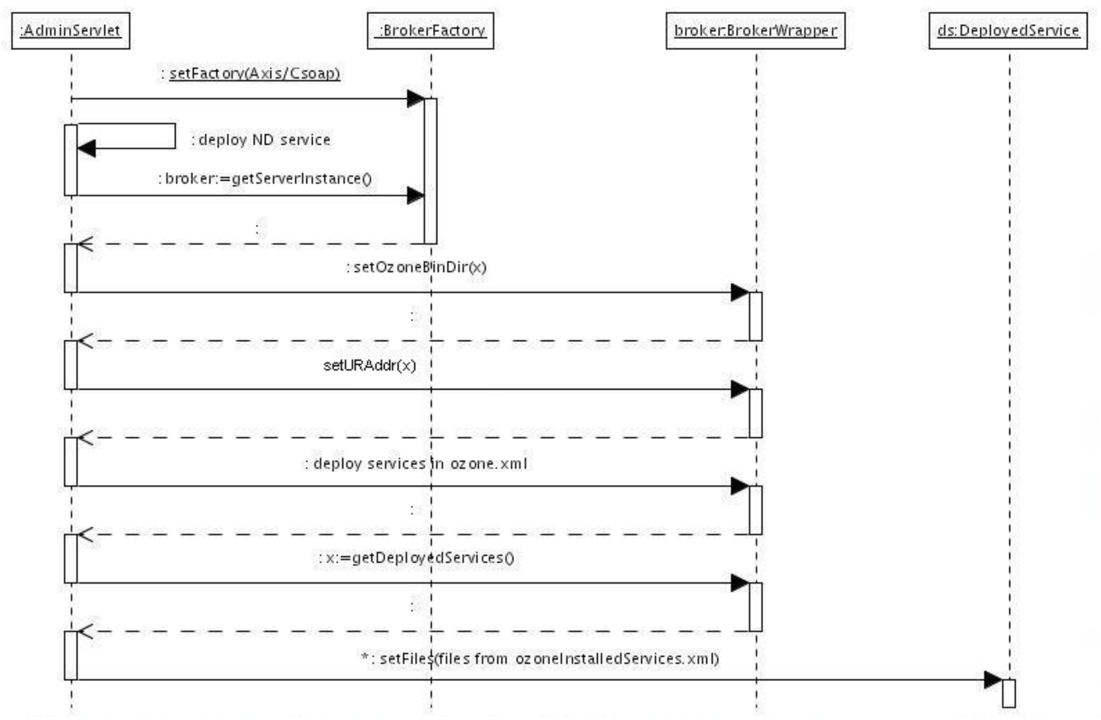

Figure 8 System startup sequence diagram

The servlet `AdminServlet` is defined by the `web.xml` file associated to the `wsami` context. This file sets the `AdminServlet` as the first servlet to be executed when the Web server starts and also defines some parameters' values: the name of the class implementing the Core Broker (Axis or CSoap based), the directory where the WSAMI software has been installed and the address of the Universal Repository service.

In Figure 8 you can see the sequence of operations executed during the initialization of the system. The `AdminServlet` sets the type of SOAP engine the system is running on by calling the `BrokerFactory.setFactory.` As this servlet is part of Core Broker and not SOAP implementation–specific, it is not aware of the underlying SOAP engine and, as already said above, this information is retrieved in `web.xml` file.

The first operation is the deployment of Naming and Discovery Service. For the deployment of this service, the deployment document is retrieve inside the system libraries and not received from the `Deployer` as described in section 3.1. Only the steps 2 and 3 described in that section will be executed, since it does not requires to be installed (step 1), to be registered on itself (step 5) and added to `wsami.xml` and `wsamiInstalledServices.xml` files (step 4).

The following steps in figure set up the value of the directory where the WSAMI software is installed (`wsamiBinDir`) that is used by ND and SLP (see section 4.2) and the address of UR service (see section 4 and 5).

The following action is the deployment of services that have already been deployed with the process described in 3.1. These services are defined in the two files `wsami.xml` and `wsamiInstalledServices.xml.`

All the services found in these files are deployed as seen in the deployment operation in 3.1 with the exception of step 1 because the have already been installed and all the WSAMI, WSDL and class files were already installed in the right positions when the services were deployed.

3.3. SOAP plug-ins and SOAP implementations

In this section we detail how the SOAP implementation and the SOAP plug-ins seen in Figure 3 are used in making service method calls.

On the client side, the SOAP implementation will be used to send a SOAP message to the service. The client application has got a reference to a service endpoint (ex: http://hostaddress:8080/services/service1) and it uses the service stub (based on the SOAP implementation installed on the client node) to send a SOAP message call to the service address.

On the server side, at the address http://hostaddress:8080/services/WSAMIServlet, the servlet `WSAMIServlet` will be running on the Web server and waiting for SOAP calls. The Core Broker configuration file `web.xml` in `services` context associates all the addresses http://hostaddress:8080/services/* to `WSAMIServlet` (see [USERGUIDE]) so that all the service calls will be redirected to this servlet.

Depending on the version of WSAMI installed on your node (for PCs based on Axis/Jakarta or for PDAs based on CSoap/Jetty), the `WSAMIServlet` associated to the address will be either the class `CoreBroker.Axis.WSAMIServlet` (provided by Axis plug-in) or `CoreBroker.CSoap.WSAMIServlet` (provided by CSoap plug-in).

You can see in the class diagram in Figure 9 that both the `WSAMIServlet` provided by Axis plug-in and the one provided by CSoap plug-in are in the same relation with the their SOAP implementation: in both cases `WSAMIServlet` inherit from a servlet provided by the SOAP implementation and overrides the `init` and `doPost` methods that are in charge of the servlet configuration and of the reception and execution of SOAP method calls.

In this section we describe in detail the WSAMIServlet of Axis plug-in (see Figure 10 and Figure 11). For details about the CSoap implementation module see [CSOAP-ARCH].

In Figure 9 you can see the relation between the classes involved in the execution of a service call in Axis SOAP engine: the `AxisServlet` is in charge of the reception of SOAP messages (`doPost` method) and it has got a reference to an instance of `AxisServer` that is a representation of the SOAP server. The current configuration of this server is stored in an object that implements the interface `EngineConfiguration`.

When a SOAP message is received, the `AxisServlet` initializes its engine instance asking for an object of type `EngineConfiguration` to the factory class `DefaultAxisServerFactory` that by default provides a `FileProvider` object.

The `FileProvider` is based on the configuration file `server-config.wsdd` that contains the configuration of the server and the list of the services deployed (it is updated at each service deployment and undeployment). At each service call, before the execution of the method call, the `AxisServer` instance is initialized by `FileProvider` using the information retrieved in this file.

Axis allow the user to develop its own customized class implementing the interface `EngineConfiguration` in order to modify the standard way `AxisServer` deploys and executes services.

In WSAMI Middleware we want to avoid reading from file `server-config.wsdd` at each service call for performance concerns and to avoid managing two files listing the deployed services, one based on WSAMI Middleware and the other on the Axis SOAP on the top of which WSAMI Middleware is running.

Our solution for the integration of WSAMI Middleware over Axis is based on the file `wsami.xml`(already described in deployment section 3.1) containing the WSAMI-related information about deployed services and the class `CoreBroke.Axis.WSAMIProvider` provided by Axis plug-in and implementing the interface `EngineConfiguration`.

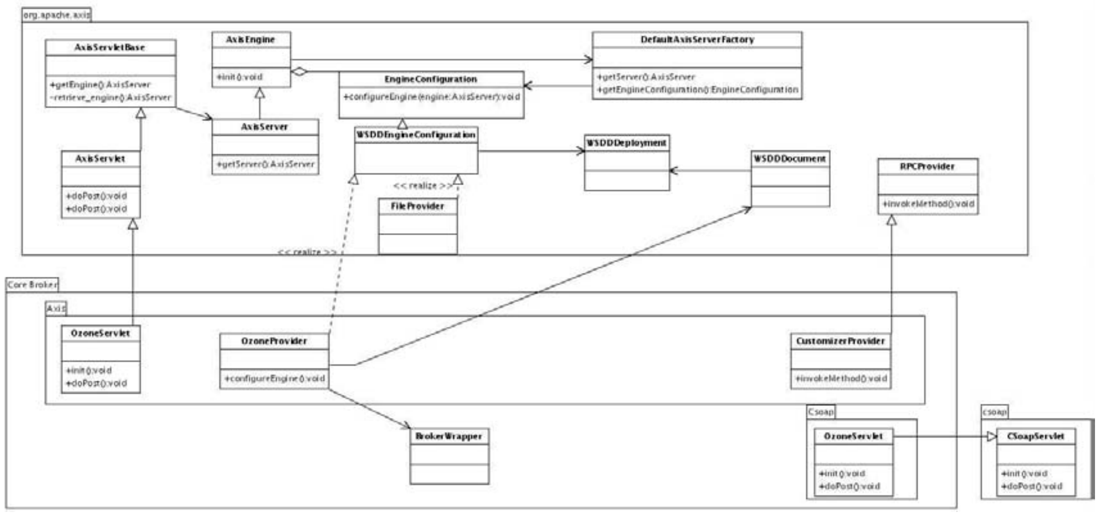

Figure 9 Core Broker SOAP Plug-in class diagram

This class uses the information stored in the Core Broker (`deploymentConfiguration`) and all the WSDD information for each service (`serviceInfo` - see step 3 in service deployment algorithm in section 3.1) to deploy services on `AxisServer`.

In Figure 10 you can see what happens when the first call to a service is received and the `WSAMIServlet` is initialized with its `init` method.

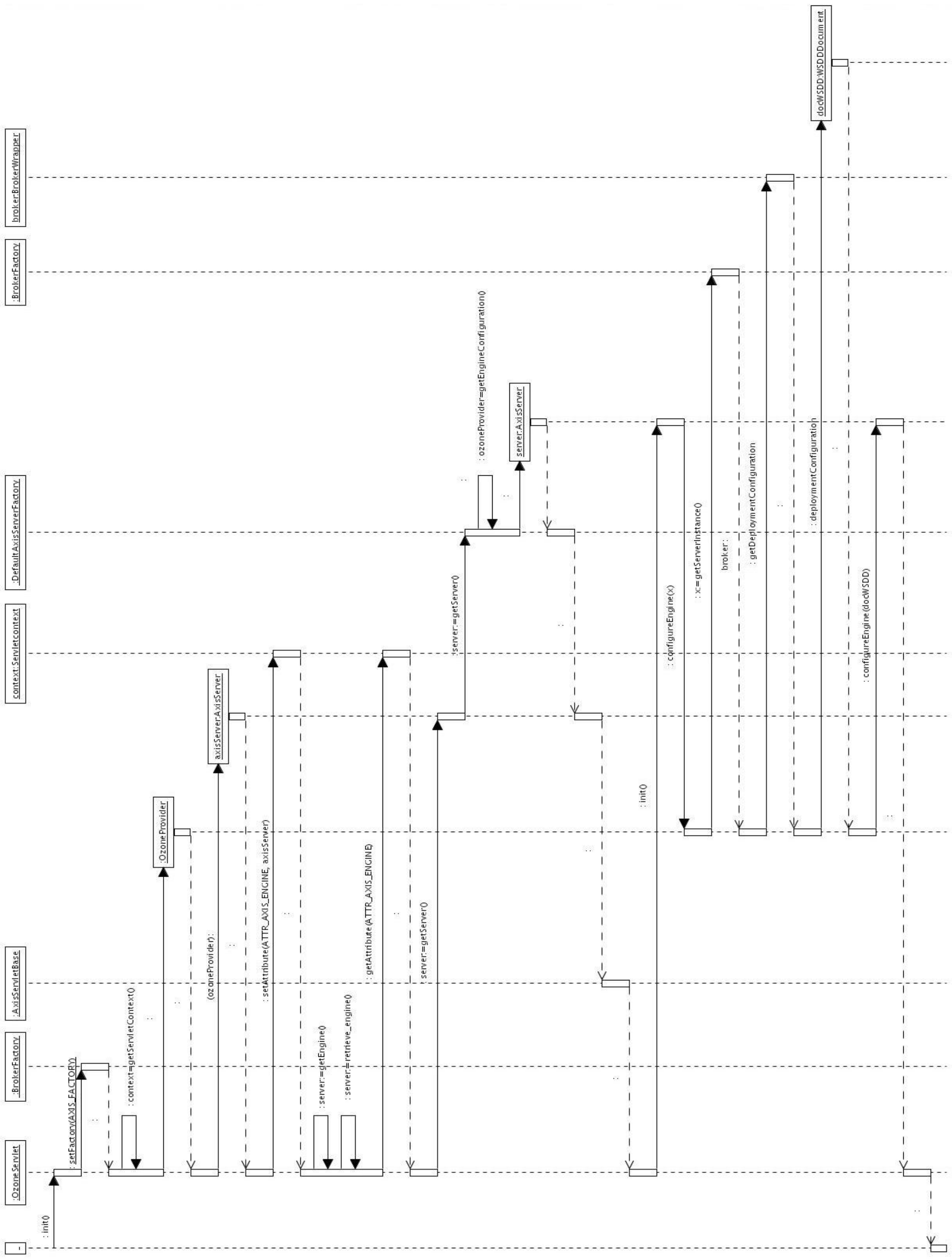

Figure 10 Making a service call on Core Broker (Axis based) - initialization sequence diagram

In a first phase, an instance of `AxisServer` is created and it is associated to an instance of `WSAMIProvider`. The instance of `AxisServer` is then associated to an attribute of the `ServletContext` of `WSAMIServlet` to be retrieved afterwards.

Then `WSAMIServlet` makes use of the `getEngine` method (inherited from its base class `AxisServeltBase`) to get a reference to the instance of `AxisServer` previously set in `servletContext.`

The following phase consists in the initialization of `AxisServer` with its `init` method: the method `WSAMIProvider.configureEngine` is called to create a `WSDDDocument` from `DeploymentConfiguration` stored in CoreBroker (see Figure 4 and Figure 5).

In Figure 11 you can see what happens after the system initialization and at each following call: as an `AxisServer` has already been associated to `WSAMIServlet`, it is now just retrieved and initialized.

The `init` will make the server ask the `WSAMIProvider` to configure the engine and then, it will finally delegate the responsibility for service execution to the SOAP server by making a call to the `doPost` method of the superclass `AxisServlet`.

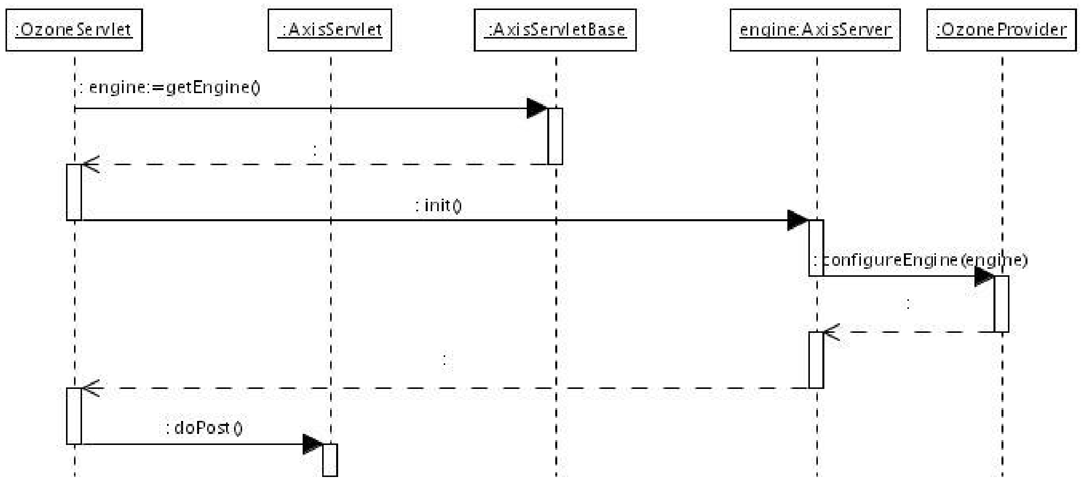

Figure 11 Core Making a service call on Core Broker (Axis based) – sequence diagram

3.4. Customization

The WSAMI abstract interface specification is extended with the definition of the QoS properties (for example: security, resource-saving) required from a service. In the service WSAMI definition, these properties are defined by the `QoSCriterion` element inside `ServiceQoS` and `ConnectorQoS` elements (see WSAMI XSD definition in 7.1).

Our approach in service QoS implementation is based on the use of a pair of content customizers at both ends, as presented in [STEI]. A Customizer decomposes into a local and a remote service. Any message exchanged between the client and the service goes

through the Customizer.
In the WSAMI specification, the `QoSCriterion` part specifies the specific QoS criteria that are enforced by the embedding customizer. The `Local` and `Remote` parts specify the abstract interfaces of the customizer services.
Given the above WSAMI QoS elements (`QoSCriterion`, `Local` and `Remote`), a service instance matches a requested service if they both specify the same document URI for the service's abstract interface and if the a service with the URI of the Local Customizer service's abstract interface is available in the environment of the client and the a service with the URI of the Remote Customizer service's abstract interface is available in the environment of the service (if `Colocation` is set to true, they must be available respectively, on the client and on the service nodes).

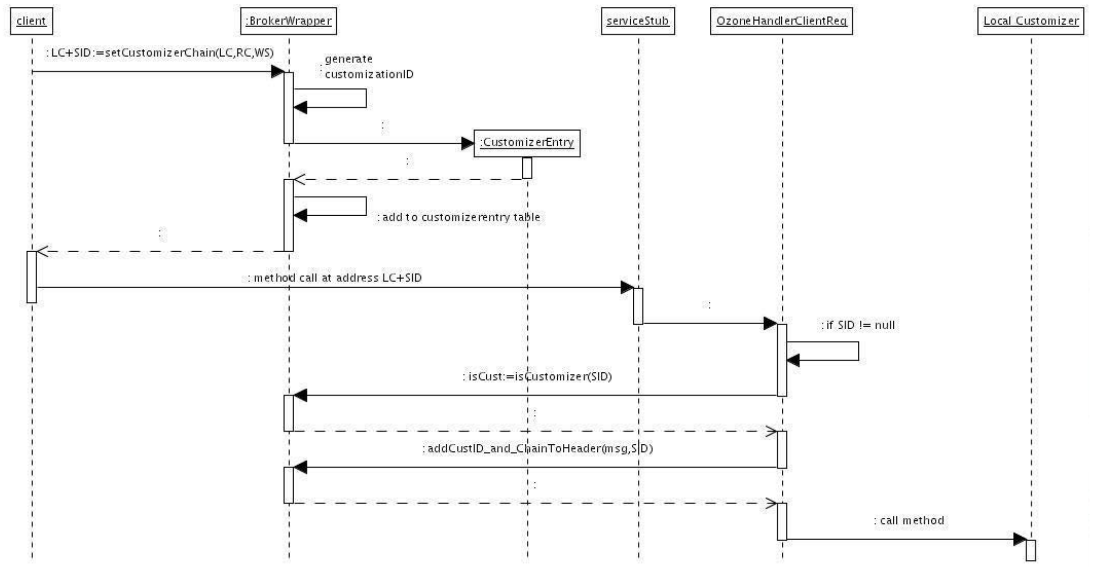
client
:BrokerWrapper
serviceStub
OzoneHandlerClientReq
Local Customizer
: LC+SID:=setCustomizerChain(LC,RC,WS)
generate customizationID
:CustomizerEntry
: add to customizerentry table
: method call at address LC+SID
: if SID != null
: isCust:=isCustomizer(SID)
: addCustID_and_ChainToHeader(msg,SID)
: call method

Figure 12 Customization sequence diagram on client side

In Figure 12 we present the details of the interaction between the client application and the Core Broker to enable the customization in the interaction between a client and the service. We take as example the interaction with the Axis based Core Broker.

The first step in client-service interaction customization is to retrieve the endpoint of the service and of the local and remote customizers with the method `NDService.getService` (not shown in figure) (for details see section 4.3).

Class diagram in Figure 4 details the Core Broker classes and methods used in the following of this section.

When the client has obtained these three values it can set the customization chain in Core Broker with the method `BrokerWrapper.setCustomizerChain` that has as input the three endpoints.

This method adds a new `CustomizerEntry` in the internal data structure of the

BrokerWrapper. This entry consists of the three values and an identifier customizationSessionID composed of a randomly generated number and of the IP address of the client node. This identifier will be used to identify this interaction in the distributed system (client, local customizer, remote customizer).

The address returned by the method setCustomizerChain is the address of the local customizer followed by the customizationSessionID.

This address is used to build a stub of the service (the interface and therefore also the stub of service and local customizer are the same, so the access to the local customizer instead of the service is transparent for the client).

The stub will use the SOAP implementation running under the Core Broker to make a call to the customizer (see section 3.3).

On the client side, the standard way the Axis SOAP engine makes calls is to initialize the engine configuration with the file client-config.wsdd that is provided in the Axis SOAP plug-in. The customization-related content of this file is the following:

```
<globalConfiguration>
  <requestFlow>
    <handler name="wsami" type="java:CoreBroker.Axis.WSAMIHandlerClientReq"/>
  </requestFlow>
</globalConfiguration>
```

In this file the normal execution flow on the client side is modified with the use of handler mechanism (for detail see Axis documentation at http://ws.apache.org/axis/java/user-guide.html and http://ws.apache.org/axis/java/architecture-guide.html).

With this configuration for clients, the underlying SOAP implementation will modify the normal SOAP message path and before sending the message to the local customizer will call the method invoke of the handler class CoreBroker.Axis.WSAMIHandlerClientReq provided by the Axis plug-in.

This method will verify the destination address of the message and if it is in format address + customizationSessionID, it means that the customization has been set up: if customizationSessionID is identified with an entry CustomizerEntry of the local Core Broker, the method BrokerWrapper.addCustID_and_ChainToHeader is called to modify the SOAP message header adding the address of remote customizer, the address of the service and the customizationSessionID.

Afterwards, the modified message is finally sent to the local customizer.

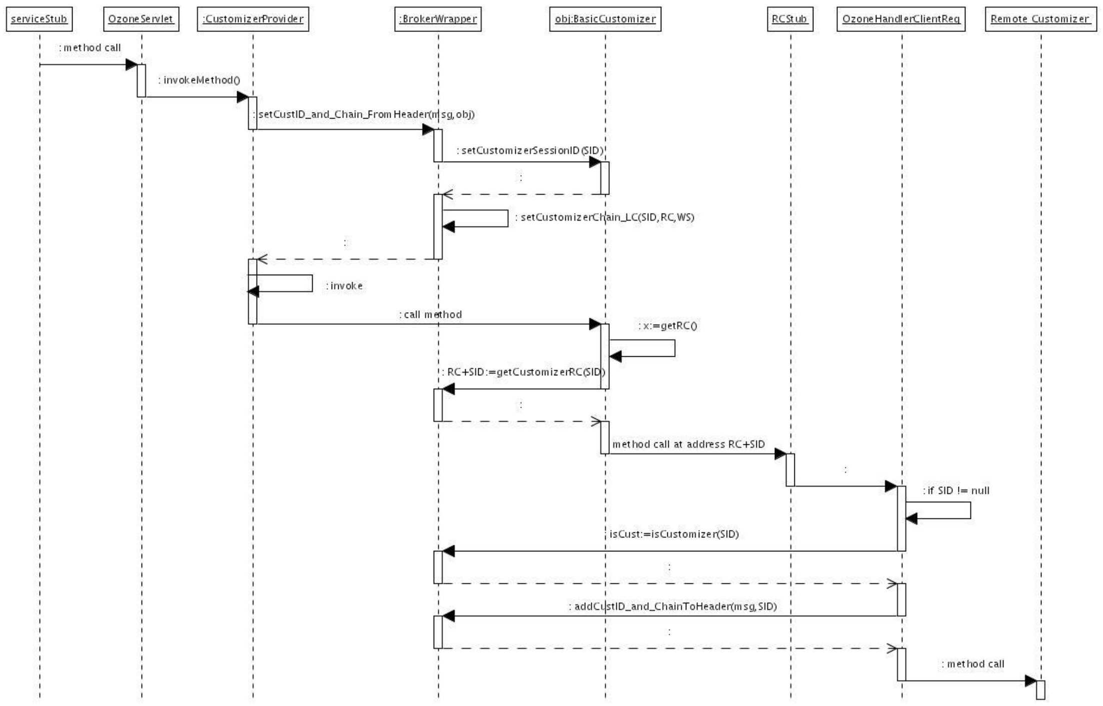
serviceStub
OzoneServlet
:CustomizerProvider
:BrokerWrapper
obj:BasicCustomizer
RCStub
OzoneHandlerClientReq
Remote Customizer
: method call
: invokeMethod()
: setCustID_and_Chain_From Header(msg,obj)
: setCustomizerSessionID(SID)
: setCustomizerChain_LC(SID,RC,WS)
: invoke
: call method
: x:=getRC()
: RC+SID:=getCustomizerRC(SID)
method call at address RC+SID
: if SID != null
isCust:=isCustomizer(SID)
: addCustID_and_ChainToHeader(msg,SID)
: method call

Figure 13 Customization sequence diagram on Local customizer

In Figure 13 we detail the reception of this modified message on the server side where the local customizer is running.
On the Core Broker, the local customizer has been deployed as a regular web service, with the algorithm described in section 3.1 but the WSDD deployment document generated respectively by `Csoap.wsdl.generator` (for CSoap) and `CoreBroker.Axis.WSDL2WSDD` (for Axis) will be modified to be adapted to the customization process.
In the case of Axis WSDD, the line:

```
<service name="localCustomizer" provider="java:RPC">
```

will be modified into:

```
<service name="localCustomizer"
            provider="java:CoreBroker.Axis.CustomizerProvider">
```

In this way the normal service method execution will be modified and instead of using the standard RPC provider for service execution of the SOAP implementation, the provider `CoreBroker.Axis.CustomizerProvider` (see Figure 9) of the SOAP plug-in will be invoked.

This class will make a call to `BrokerWrapper.setCustID_and_Chain_FromHeader` that retrieves the address of remote customizer, the address of the service and the `customizationSessionID` from the SOAP message header.

These values are then used to configure the Core Broker itself with the method

BrokerWrapper.setCustomizerChain_LC that adds a CustomizerEntry to the CustomizationTable (see Figure 4).

The customizationID is used to configure the local customizer by calling its method setCustomizerSessionID that copy this value into the Local Customizer object. This customizationID identifies the interaction to which the local customizer takes part in.

The Local Customizer object must extend the class CoreBroker.BasicCustomizer (see Figure 4) and therefore inherits the methods setCustomizerSessionID, getRC and getWS.

CoreBroker.Axis.CustomizerProvider finally returns the control to its super class org.apache.axis.providers.java.RPCProvider to manage the service call with the default behavior.

The local customizer method will be executed and when this method implementation requires making a call to the remote customizer, its address can be retrieve using the method getRC of the local customizer that makes use of the customizationID previously set with setCustomizerSessionID to ask the Core Broker the address corresponding to this ID in its CustomizerTable data structure.

When the Remote Customizer method call is invoked, the same process explained for the call to Local Customizer is started and the method invoke of the handler class CoreBroker.Axis.WSAMIHandlerClientReq is called: the destination address of the message is address + customizationSessionID, customizationSessionID is used to retrieve a CustomizerEntry in the local Core Broker and the method BrokerWrapper.addCustID_and_ChainToHeader is called to modify the SOAP message header adding the address of the service and the customizationSessionID. Finally, the modified message is finally sent to the remote customizer.

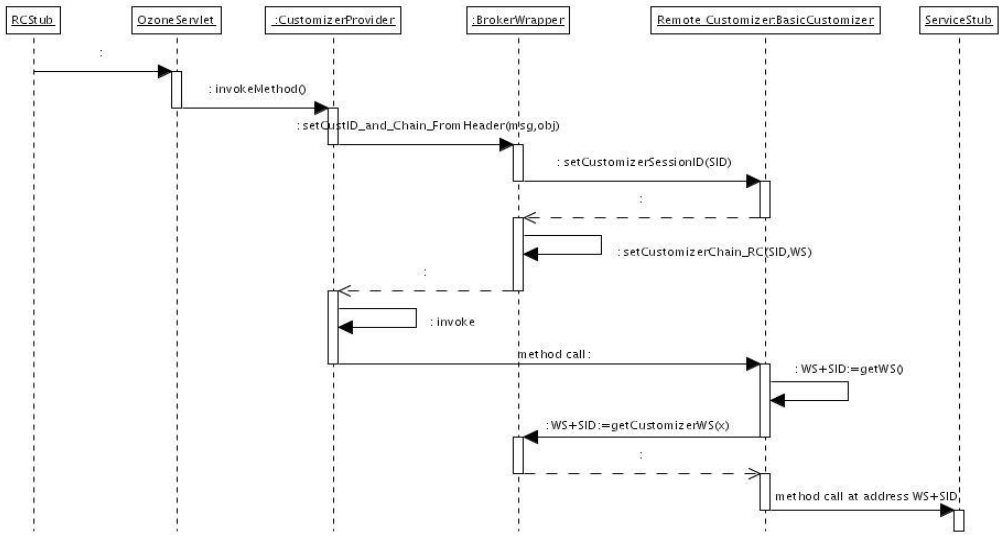
RCStub
OzoneServlet
:CustomizerProvider
:BrokerWrapper
Remote Customizer:BasicCustomizer
ServiceStub
: invokeMethod()
: setCustID_and_Chain_FromHeader(msg,obj)
: setCustomizerSessionID(SID)
: setCustomizerChain_RC(SID,WS)
: invoke
method call :
: WS+SID:=getWS()
: WS+SID:=getCustomizerWS(x)
method call at address WS+SID

Figure 14 Customization sequence diagram on Remote customizer

In Figure 14 we detail the reception of this modified message on the server side where the remote customizer is running. In the figure you can see that the sequence is very similar to the one described above for the local customizer.

For the remote customizer the rules for the deployment on the Core Broker and for the inheritance from the `CoreBroker.BasicCustomizer` are the same as for the local customizer.

The only difference is that the methods that deals with the address endpoints of local and remote customizers and service (`BrokerWrapper.setCustID_and_Chain_FromHeader` and `BrokerWrapper.setCustomizerChain_RC`) will not have to handle the address of the local customizer because it is useless at this stage of the interaction.

3.5. Tools

3.5.1. Platform-dependent Tools

The implementation of the WSAMI Middleware requires some platform-dependent features (e.g., getting information about the network cards and the power plugging and handling the routing table) that are not provided by the standard Java class library.

To solve this problem we provide some tools, based on Java Native Interface (JNI), that can retrieve all the required platform-dependent information. The JNI allows Java applications to get access to functionalities implemented in already existent library or application written in another programming language (for example C or C++).

The utilities described in this section are based on JNI and libraries written in C language (either provided by operating system or available as free software and distributed under the terms of the GNU Library General Public License) and are provided both for Linux and for Windows operating systems.

Power Plugging

In the service discovery algorithm described in section 4.3 we make use of power-plugging information of a node: we want to know if it is power plugged or not. To have access to this information we make use of the Advanced Power Management (APM) library.

The APM library is a set of user-level programs to control the Advanced Power Management system found in all modern laptop computers and most modern desktops. For the Linux version of the WSAMI Middleware this feature is implemented using the package available at the address http://www.worldvisions.ca/~apenwarr/apmd/.

For the Windows version of the WSAMI Middleware this feature is based on operating system calls.

Network Interfaces

In the service discovery algorithm described in section 4.3 but also in different components like the deployment system (section 3.1), the registration of ND on SLP (section 4.2) and the UR service (section 5) we need information about the configuration of the network cards installed on the node.

This functionality is implemented only on Linux platform. The routine to get information about the network interfaces (interface name and IP address) is based on kernel system calls, while the routine to get information related to wireless configuration (if a network card is wireless or not and if it is, the wireless network name and if it is in ad-hoc or infrastructure mode) is based on the Wireless Tools for Linux that is available at the address http://www.hpl.hp.com/ personal/Jean_Tourrilhes/Linux/Tools.html.

Routing Table

In the service discovery algorithm on nodes connected to ad-hoc networks described in section 4.3 we need to get access and modify the routing table.

Only the Linux version of this feature is provided by the WSAMI Middleware, and it is implemented making use of the Linux `ip` command (see "`man ip`" for documentation and ftp://ftp.sunet.se/pub/Linux/ip-routing/ for source code).

The following modifications to the source code are required to let the library be properly integrated into the WSAMI Middleware:

1. In all `*.c` and `*.h` files the line `#include "utils.h"` must precede all other lines of type `#include "... .h"`
2. In `lib/utils.c` file the method `get_prefix_1` must be modified in the following way to avoid a segmentation fault:

   The lines:

   ```
   int err;
   unsigned plen;
   char *slash;
   ```

   must be modified into:

   ```
   int err;
   unsigned plen;
   char *slash;
   char *arg_tmp = strdup(arg);
   ```

   and the lines:

   ```
   slash = strchr(arg, '/');
   if (slash)
         *slash = 0;
   ```

   must be modified into:

```
slash = strchr(arg_tmp, '/');
if (slash)
      *slash = 0;
```

3. In file `ip/ip.c` the `main` method must removed to avoid an error raised during the generation of the shared library used by the tool.
4. In file `ip/iproute.c` in method:

   ```
   int iproute_modify(int cmd, unsigned flags, int argc, char **argv)
   ```

   The `exit(1)` and `exit(2)` system calls must be replaced by system calls `return 1` and `return 2` to avoid the `ip` library killing the Web server (either Jetty or Jakarta Tomcat) on which the WSAMI Middleware is running.
5. In file `ip/iproute.c` in method:

   ```
   int iproute_modify(int cmd, unsigned flags, int argc, char **argv)
   ```

   To solve bug in library (socket is opened but never closed) and avoid "too many open files" error, add the line:

   ```
   rtnl_close(&rth);
   ```

   at the end of the method, just before the final line

   ```
   return 0;
   ```

3.5.2. WSDL2WSAMI & Deployer Tools

The `WSDL2WSAMI` and `Deployer` tools provide the functionalities required for the deployment of a service on the WSAMI Middleware. You can find a detailed description of the syntax and use of these tools in [USERGUIDE].

The `WSDL2WSAMI` tool is used to generate the deployment document and the WSAMI files associated to a service. You can see some examples of the WSAMI files for the Middleware-related services, the Naming & Discovery Service in section 4.4, the UR Service in section 5.1 and the Gateway Service in section 6.1. All these files are compliant with the XSD specification defined in section 7.1.

The functionalities offered by `Deployer` have already been described in section 3.1 about the deployment of a service and Figure 6 shows the actions executed by this tool to deploy a service on the local node. The deployment document sent to the `AdminServlet` is compatible with XSD in 7.3 and is produced from the deployment document generated by `WSDL2WSAMI` and compliant with XSD specification defined in section 7.2.

In Figure 15 you can see how these tools have been implemented: a common class `CLBaseClass` provides the basic functionalities to implement a command line application with a variable number of options and parameters. The `WSDL2WSAMI` and `Deployer` classes extend the superclass `CLBaseClass` and exploit the functionalities offered in order to implement the tools described above.

The class `CLBaseClass` implements methods to parse the command line options (`parseArgs`, `help` and `run`) and to handle their values (`getArrayStringOption`, `getStringOption`, `getBooleanOption`) and defines some abstract methods that must be implemented by subclasses to set the application specific options and parameters (`setNparams`, `setParamDescription`, `setValues` and `addOptions`). The data structure `options` consists of the list of available options and, after the parsing, the values associated to the options.

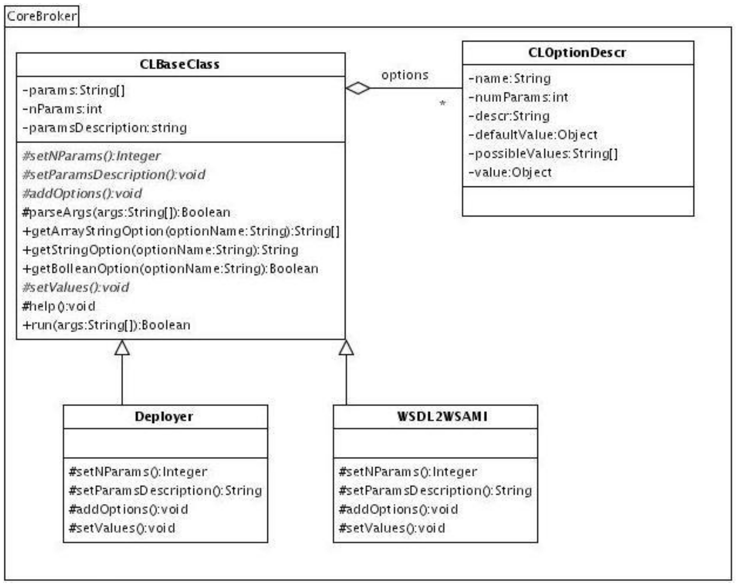

Figure 15 Tools class diagram

In Figure 16 you can see how the constructor of `CLBaseClass` and the `run` method are implemented: the former initializes the options list by adding the `help` option and then makes a call to the abstract methods `setNparams` and `setParamDescription` to define the number of parameters required by the command and their associated description in the help message.

The latter, the `run` method, makes a call to the abstract method `addOptions` to define the list of available options for the command implemented by the specific subclass of `CLBaseClass` and then calls the `parseArgs` method that parses the command line taking into account the available options as defined by `CLOptionDescr` in `options` data structure and if the parsing is correct, the field `CLOptionDescr.value` of each option is filled with the option value.

If the parsing is not correct, the `run` method returns false to notify the failure of the parsing, otherwise the `setValues` is invoked to allow the application to use the values of the options to initialize class member variables to be used later and the method returns a true value.

If the returned valued is true, the application can continue to execute the command related operations, that is, creating WSAMI files for `WSDL2WSAMI` and sending deployment document for `Deployer`.

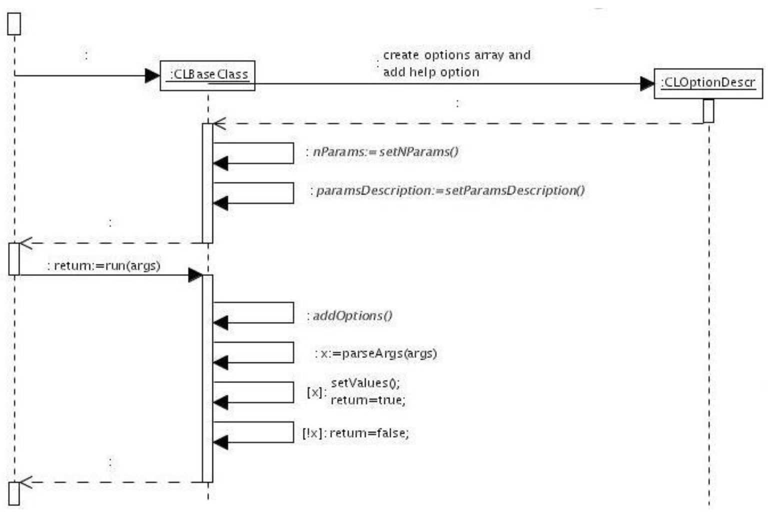

Figure 16 Tools sequence diagram

4. Naming & Discovery Service

The role of the Naming & Discovery Service (that will be simply referred to as ND service in the following) is the localization of services available on the network. ND is designed as a distributed naming service with an instance of this service deployed on the Core Broker of every WSAMI-compliant device. We define a protocol for lookup and discovery of services that allows the ND services to mutually exchange information about services' concretes bindings (i.e., the endpoint address to access a service) and abstract interfaces.
To discover ND services running in the local area, ND is based on Service Location Protocol (for implementation details see section 4.2)

Practically, service requests are issued to the local ND service, providing as input the URI of the corresponding WSAMI abstract interface (method `getService`) or the URL of the service instance (method `getService_ByInstanceURL`). The ND service then seeks a matching service instance with respect to service instances that may be reached from the terminal. Seeking a matching instance may further lead to customize the connector, depending on the value of the `QoSConnector` element of the service's abstract interface and on the value of the `QoSConnector`, `Local` and `Remote` elements of the service's customization document (for implementation details see section 4.3).

The ND service manages two service repositories: one for the local service instances (section 4.1) and one for remote services (section 4.3).

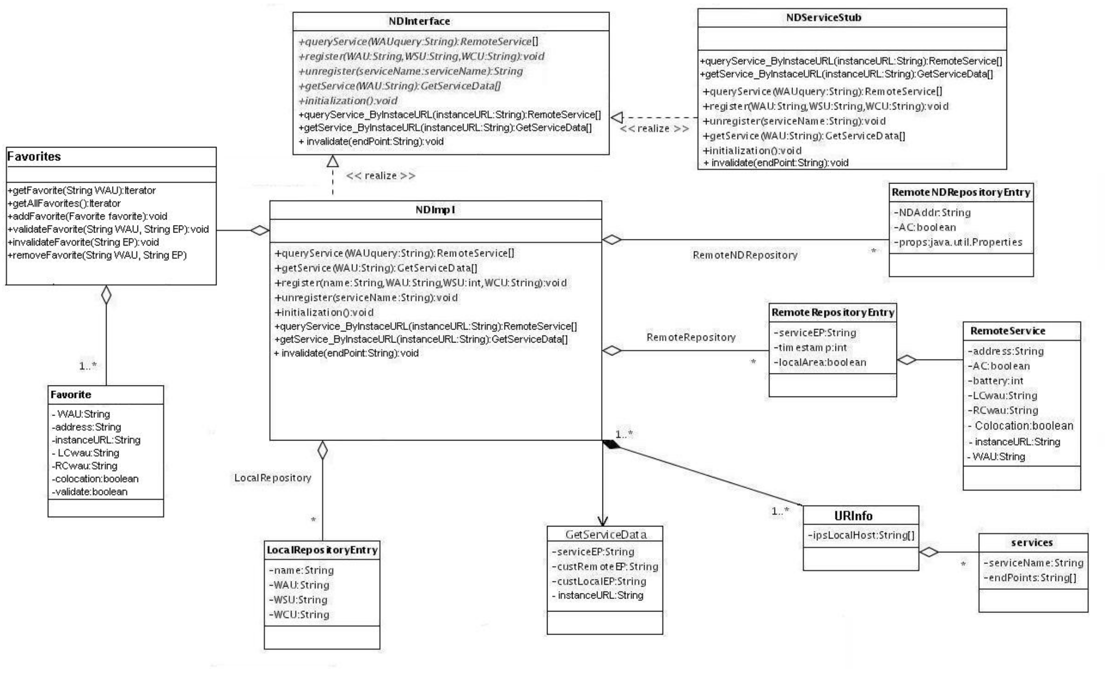
NDInterface
+queryService(WAUquery:String):RemoteService[]
+register(WAU:String,WSU:String,WCU:String):void
+unregister(serviceName:serviceName):String
+getService(WAU:String):GetServiceData[]
+initialization():void
+queryService_ByInstaceURL(instanceURL:String):RemoteService[]
+getService_ByInstaceURL(instanceURL:String):GetServiceData[]
+ invalidate(endPoint:String):void
NDServiceStub
+queryService_ByInstaceURL(instanceURL:String):RemoteService[]
+getService_ByInstaceURL(instanceURL:String):GetServiceData[]
+queryService(WAUquery:String):RemoteService[]
+register(WAU:String,WSU:String,WCU:String):void
+unregister(serviceName:String):void
+getService(WAU:String):GetServiceData[]
+initialization():void
+ invalidate(endPoint:String):void
<< realize >>
<< realize >>
Favorites
+getFavorite(String WAU):Iterator
+getAllFavorites():Iterator
+addFavorite(Favorite favorite):void
+validateFavorite(String WAU, String EP):void
+invalidateFavorite(String EP):void
+removeFavorite(String WAU, String EP)
NDImpl
+queryService(WAUquery:String):RemoteService[]
+getService(WAU:String):GetServiceData[]
+register(name:String,WAU:String,WSU:int,WCU:String):void
+unregister(serviceName:String):void
+initialization():void
+queryService_ByInstaceURL(instanceURL:String):RemoteService[]
+getService_ByInstaceURL(instanceURL:String):GetServiceData[]
+ invalidate(endPoint:String):void
RemoteNDRepositoryEntry
-NDAddr:String
-AC:boolean
-props:java.util.Properties
RemoteNDRepository
*
RemoteRepositoryEntry
-serviceEP:String
-timestamp:int
-localArea:boolean
RemoteRepository
*
RemoteService
-address:String
-AC:boolean
-battery:int
-LCwau:String
-RCwau:String
- Colocation:boolean
- instanceURL:String
- WAU:String
1..*
Favorite
- WAU:String
-address:String
-instanceURL:String
- LCwau:String
-RCwau:String
-colocation:boolean
-validate:boolean
LocalRepository
*
LocalRepositoryEntry
-name:String
-WAU:String
-WSU:String
-WCU:String
GetServiceData
-serviceEP:String
-custRemoteEP:String
-custLocalEP:String
- instanceURL:String
1..*
1..*
URInfo
-ipsLocalHost:String[]
*
services
-serviceName:String
-endPoints:String[]

Figure 17 Naming and Discovery service class diagram

The ND service is deployed at startup at the address http://localhost:8080/services/ND (see WSDL concrete file in section 4.4) where localhost is the IP address of the node. The deployment document used is the following:

```
<deploy
      xmlns="http://www-rocq.inria.fr/arles/wsami"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami deploy.xsd" >
  <service name="ND">
    <wad name="NDAbstract.wsami"/>
    <wsd name="NDService.wsami"/>
    <wau name="http://localhost:8080/wsami/NDAbstract.wsami"/>
    <implementation>
      <class name="NDService.NDImpl"/>
      <type name="javaclass"/>
      <package name="NDService"/>
    </implementation>
    <scope name="Application" />
  </service>
</deploy>
```

As you can see in the document, the ND service WAU identifier is http://localhost:8080/wsami/NDAbstract.wsami.

4.1. Local repository

The local repository is a data structure storing information about the locally deployed services. In Figure 17 you can see how the repository is implemented.

A `Hashmap` of `LocalRepositoryEntry` where the three fields `WAU`, `WSU` and `WCU` are respectively the URI of the WSAMI Abstract, Service and Customizer documents and the `name` is the same name used in Core Broker to identify the deployed service.

This data structure is updated by methods `ND.register` and `ND.unRegister`.

The other information about the service (address endpoint and content of the WSAMI documents) can be retrieved from the Core Broker using the `name` of the service.

4.2. SLP and ND

As you can see in Figure 1, the ND Service is based on Service Location Protocol (SLP). In our implementation the Service Location Protocol (SLP) used is OpenSLP version 1.0.11 (http://www.openslp.org).

SLP is used to discover ND Services running in the local area (ad-hoc or infrastructure-based):

- On every host, an SLP server runs under the Service Agent (SA) mode.
- The ND service registers itself on the SLP server with the same address as it has been deployed on the Core Broker during the system startup (for example: http://128.93.8.102:8080/services/ND). For more details see the Registration algorithm below.
- The ND service makes use of SLP client library to multicast requests to all the SLP servers in the local area to collect the addresses of all the reachable ND services.

  For more details see the Discovery algorithm below.

Discovery algorithm

The ND discovery algorithm is implemented as a thread repeated periodically to retrieve all the local area instances of ND services. Each discovery operation is based on classes provided by the OpenSLP Java library.

The `SLPLocator` class is used to execute the search for ND services. The `ServiceType` class defines the type of service the `SLPLocator` must search for. In our case the name of the service to be specified is `"service:ND"` (that is the same name used by ND to register itself on the local instance of SLP server, for example it can be `"service:ND:http://128.93.8.102:8080/services/ND"`) and the attribute to be specified is `"(&(ac=true))"` or `"(&(ac=false))"` depending on the type of nodes that you what to find (power plugged or not).

The ND addresses and relative properties (for example if the power is plugged or not) collected by SLP are used to update the `RemoteNDRepositoryEntry` entries (each one identifying a ND service) in the `Hashtable RemoteNDRepository` (see Figure 17).

On gateway nodes (node connected both on ad-hoc networks and infrastructure-based networks) the Gateway Service (see section 6) is deployed at system start-up after the deployment of the ND Service. On these nodes the lookup algorithm is slightly modified because the used SLP Java library does not support multicast of SLP requests on multiple network interfaces.

To solve this problem we make use of the library version OpenSLP 1.1.5 (that is not provided with Java interface) through the Java Native Interface. This version of SLP supports multicast requests on all the network interfaces of the node. This library does not support multithread, so instead of implementing the search for ND with "`ac=true`" and "`ac=false`" with two concurrent threads, we use a single thread that does a serial execution of the two queries.

Registration algorithm

The mobility of devices allows them to move from one infrastructure-based network to another one or to switch from an ad hoc to an infrastructure-based network and vice versa. You can further plug (or unplug) a network card in the host. All this operation can modify the IP address associated to the host or associate multiple IP addresses to the host.

To solve this problem related to the dynamicity of IP addresses and maintain the system in a consistent state, the SLP server and ND service require to be reconfigured taking into account the evolving network configuration of the host.

The running instance of SLP server is tied to a fixed IP address (one or more) and does not support the dynamic modification of this address(es). Therefore, the system reconfiguration must be handled by the ND that is in charge of identifying the need for reconfiguration and start a new instance of SLP server adapted to the new network configuration.

We detail now the registration algorithm that is implemented as a thread started by ND when it is deployed (see Figure 18 for state diagram representation of the algorithm).

When the thread is started, the first action is to get the IP addresses of the node and write them into the `slp.conf` file.
Then it sends a `WAITING_SLP_STARTING` event and sets the state to `SLP_NOT_RUNNING`. Now the thread must wait for a running instance of SLP before taking any further actions. In the description of this algorithm the actions of sending and receiving an event are implemented with a file shared between ND Service and the SLP server script in which they write(send) or read(receive) the code corresponding to the event (this file is located in the `WsamiBinDir` directory described in system startup section 3.2)
Afterwards, a loop is started and these actions are periodically repeated:

- get information about power (see section 3.5)
- receive an event:

- if state is `SLP_NOT_RUNNING` and the event is `SLP_STARTED`:
  - register the ND service on SLP server with the latest IP addresses written in `slp.conf` and the detected power information
  - send event `ND_REGISTRED`
  - set state to `SLP_RUNNING`
- if state is `SLP_RUNNING`
  - IP addresses are retrieved and compared to old ones (those written in `slp.conf`)
  - if IP addresses have changed:
    - write IP addresses in `slp.conf`
    - send event `WAIT_SLP_STARTING`
    - set state to `SLP_NOT_RUNNING`
    - if UR registration is enabled: update the address endpoints of the services registered on the UR with the method `updateAllAddresses` (see section 5)
  - if IP addresses have not changed:
    - get information about power
    - if power information have changed:
      - unregister the ND service from SLP server
      - register ND service with new power information on SLP server

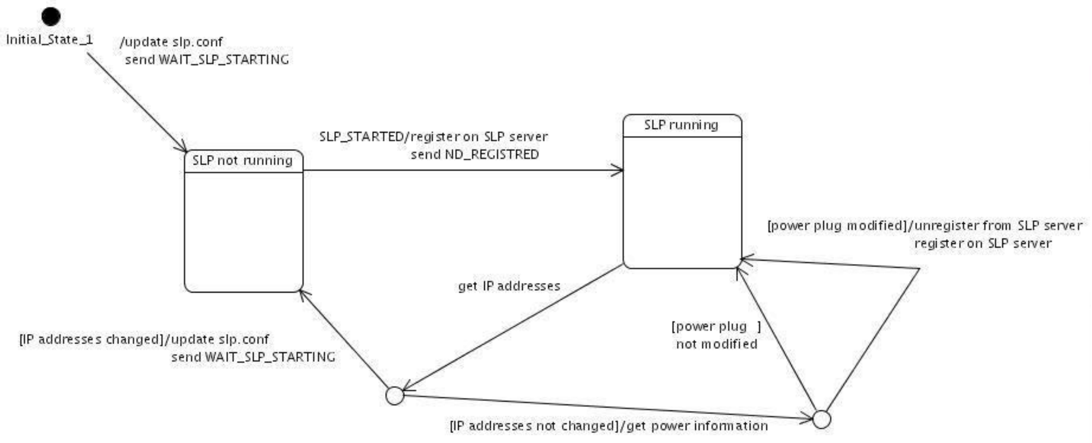

Figure 18 ND registration algorithm state diagram

The SLP server is handled by the SLPMgr shell script that must be executed before the system startup described in section 3.2. The operations executed by this script can be summarized with the state diagram in Figure 19.

When the script starts, it sends an `INITIALIZATION_SYSTEM` event and then these actions are periodically repeated:

- receive an event:
    - if the event is `WAITING_SLP_START`:
        - stop the running instance of SLP
        - start a new instance of SLP server with the configuration file `slp.conf` written by ND service
        - send event `SLP_STARTED`

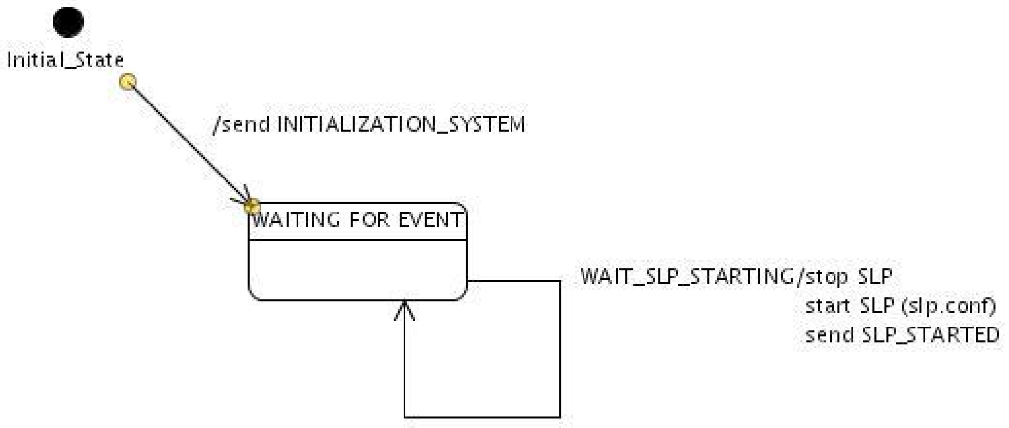

Figure 19 SLP server Mgr state diagram

4.3. GetServices and Remote Repository

The repository of remote services is implemented as a `Hashmap` of `RemoteRepositoryEntry` (see Figure 17) each entry containing the information about a service retrieved by the algorithm described in this section.

When an application wants to access a service, it must obtain its address endpoint. This information can be provided either by the `ND.getService` or the `ND.getService_ByInstanceURL` method. The input value for the former method is the WAU identifier while the input value for the latter is the URL of the service instance of the searched service and the output for both methods is the list of services found with respect to the algorithm described in this section and corresponding to the specified identifier. You can see in `ND.GetServiceData` that for each service we define not only the endpoint, but also the endpoints of the local and remote customizers. This latter information will be retrieve only if the retrieved service is defined as customized.

When called, the `ND.getService` and `ND.getService_ByInstanceURL` methods looks for a service instance matching the required parameters inside the local repository (`LocalRepository` in Figure 17) and if not found, execute the algorithm described below.

If the network is in the infrastructure-based mode (see section 3.5.1), the following sequential steps are performed until an instance is found:

I1. The service instance is sought in the local area by first looking for eligible instances hosted by power-plugged nodes within the remote repository. This information is retrieved in the `RemoteRepositoryEntry` list of remote services, with the fields `localArea=true` and `RemoteService.AC=true`.

    If there is no such reachable instance, a request for a service instance is sent to the ND services in the local area that are running on power-plugged nodes. The ND services to be contacted can be identified using the `RemoteNDRepositoryEntry.AC=true` field. A call to `ND.queryService` method is sent to each ND service at the address contained in `RemoteNDRepositoryEntry.NDAddr`.

    Upon reception of the request (`queryService`), the remote ND service seeks matching instances within the local repository (see section 4.1) and returns the eligible instances -if any as an array of `ND.RemoteService` objects.

I2. The service instance is sought on the Internet by looking for eligible instances within the remote repository and local Web services directory –if any. The former information is retrieved in the `RemoteRepositoryEntry` list of remote services, with the field `localArea=false`. The latter information is retrieved in the local Web services directory stored in `Favorites` (in Figure 17 - see [DEVELGUIDE] and [USERGUIDE] for more information about how to manage the local Web services directory using a UDDI client).

I3. A reachable service instance hosted by a non power-plugged node in the local area is sought within the remote repository. As in I1., this information is retrieved in the `RemoteRepositoryEntry` list of remote services, with the fields `localArea=true` and `RemoteService.AC=false`.

    If none is found, the service is requested to the ND services running on the (non-power-plugged) wireless nodes in the local area. As in I1, the ND services to be contacted can be identified using the `RemoteNDRepositoryEntry.AC=false` field and the `ND.queryService` call is sent to address `RemoteNDRepositoryEntry. NDAddr`.

I4. Ultimately, the service request is sent via the Internet, to the universal repository that is provided by the user upon ND initialization. The ND makes use of the method `getServices` of UR service to get the endpoints of the services matching the WAU identifier across the network. The universal repository service and its relationships and interactions with ND service are detailed in section 5.

The retrieved service can have customization information associated to it. If so, the same algorithm from I1 to I4 is applied to the service's local and remote customizer abstract interfaces until an instance is found. If `colocation` attribute is set to `true` for the customizers (see `RemoteService` class in Figure 17), the local customizer will match only if its address is the same as the node where the client is running and the remote customizer will match only if its address is the same as the node where the service is running.

If the network is in the ad hoc mode (see section 3.5.1), the service is requested to nodes in the local area, according to the following steps, until an instance is found:

A1. A reachable service instance hosted by a power-plugged node is sought in a way similar to Step I1.

A2. The same process as above is applied for non power-plugged nodes in a way similar to Step I3.

A3. An instance of a Gateway service (see section 6) that relays messages for access to the Internet is sought, according to the above Steps A1 and A2. If an instance is found, the routing table is configured (see section 3.5.1) according to the information obtained from the Gateway service in order to use the node where this service is running as default address to be connected to the Internet and the target service is sought within the Internet according to the above Steps I2 and I4.

The retrieved service can have customization information associated to it. If so, the same algorithm from A1 to A3 is applied to the service's local and remote customizer abstract interfaces until an instance is found. If `colocation` attribute is set to `true` for the customizers (see `RemoteService` class in Figure 17), the local customizer will match only if its address is the same as the node where the client is running and the remote customizer will match only if its address is the same as the node where the service is running.

If one or more eligible instances are retrieved, the remote repository (`hashmap` of `RemoteRepositoryEntry`) is updated accordingly. And, all the retrieved instances are returned by the `getService` method in an array of `GetServiceData` classes.

The method `ND.invalidate` can be used by a client application when a service instance it is interacting with is no longer available (fro example if an exception is raised while making a call to the service). The client can invalidate the address endpoint of this service in the ND's Remote Services Repository to prevent the `ND.getService` and `ND.getService_ByInstanceURL` methods from returning this address as valid service endpoint.

4.4. WSAMI and WSDL files

NDAbstract.wsami

```
<?xml version="1.0" encoding="UTF-8"?>

<wsami
        xmlns="http://www-rocq.inria.fr/arles/wsami"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami WSAMI.xsd">
```

```
 <Definition name="ND" targetNamespace="http://localhost:8181/">
  <Abstract name="ND">
   <Interface hrefSchema="http://localhost:8080/wsami/NDInterface.wsdl"/>
  </Abstract>
 </Definition>
</wsami>
```

NDService.wsami

```
<?xml version="1.0" encoding="UTF-8"?>

<wsami
        xmlns="http://www-rocq.inria.fr/arles/wsami"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami WSAMI.xsd">

 <Definition name="ND" targetNamespace="http://localhost:8181/">
  <Service name="ND">
   <Abstract hrefSchema="http://localhost:8080/wsami/NDAbstract.wsami"/>
   <Concrete hrefSchema="http://localhost:8080/wsami/NDConcrete.wsdl"/>
  </Service>
 </Definition>
</wsami>
```

NDInterface.wsdl

```
<?xml version="1.0" ?>

<definitions name="urn:NDService"
             targetNamespace="urn:NDService"
             xmlns:tns="urn:NDService"
             xmlns:typens="urn:NDService"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- type defs -->
  <types>
    <xsd:schema targetNamespace="urn:NDService">

     <xsd:complexType name="RemoteService">
       <xsd:sequence>
        <xsd:element name="WAU" type="xsd:string"/>
        <xsd:element name="address" type="xsd:string"/>
        <xsd:element name="instanceURL" type="xsd:string"/>
        <xsd:element name="AC" type="xsd:boolean"/>
        <xsd:element name="battery" type="xsd:int"/>
        <xsd:element name="LCwau" type="xsd:string"/>
        <xsd:element name="RCwau" type="xsd:string"/>
        <xsd:element name="colocation" type="xsd:boolean"/>
       </xsd:sequence>
     </xsd:complexType>

     <xsd:complexType name="ArrayOfRemoteService">
       <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
          <xsd:sequence>
            <xsd:element name="strings" minOccurs="0" maxOccurs="unbounded"
```

```
                 type="typens:RemoteService"/>
       </xsd:sequence>
     </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="RemoteServices">
    <xsd:sequence>
     <xsd:element name="list" type="typens:ArrayOfRemoteService"/>
    </xsd:sequence>
  </xsd:complexType>


  <xsd:complexType name="GetServiceData">
    <xsd:sequence>
     <xsd:element name="serviceEP" type="xsd:string"/>
     <xsd:element name="custLocalEP" type="xsd:string"/>
     <xsd:element name="custRemoteEP" type="xsd:string"/>
     <xsd:element name="instanceURL" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ArrayOfGetServiceData">
    <xsd:complexContent>
     <xsd:restriction base="soapenc:Array">
       <xsd:sequence>
         <xsd:element name="strings" minOccurs="0" maxOccurs="unbounded"
                 type="typens:GetServiceData"/>
       </xsd:sequence>
     </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

 </xsd:schema>
</types>



<!-- message declns -->
<message name="initializationRequest"/>
<message name="stopRequest"/>

<message name="RegisterRequest">
  <part name="name" type="xsd:string"/>
  <part name="WAU" type="xsd:string"/>
  <part name="WSU" type="xsd:string"/>
  <part name="WCU" type="xsd:string"/>
</message>

<message name="UnRegisterRequest">
  <part name="WAU" type="xsd:string"/>
</message>
<message name="UnRegisterResponse">
  <part name="Error" type="xsd:string"/>
</message>

<message name="GetService_ByInstanceURLRequest">
  <part name="instanceURL" type="xsd:string"/>
</message>
<message name="GetService_ByInstanceURLResponse">
```

```
    <part name="serviceEP" type="typens:ArrayOfGetServiceData"/>
  </message>

  <message name="GetServiceRequest">
    <part name="WAU" type="xsd:string"/>
  </message>
  <message name="GetServiceResponse">
    <part name="serviceEP" type="typens:ArrayOfGetServiceData"/>
  </message>

  <message name="QueryService">
    <part name="WAUquery" type="xsd:string"/>
  </message>
  <message name="ReplyService">
    <part name="EP" type="typens:RemoteServices" />
  </message>

  <message name="QueryService_ByInstanceURL">
    <part name="instanceURLquery" type="xsd:string"/>
  </message>
  <message name="ReplyService_ByInstanceURL">
    <part name="EP" type="typens:RemoteServices" />
  </message>

  <message name="invalidateRequest">
    <part name="serviceEP" type="xsd:string"/>
  </message>
 <!-- port type declns -->
  <portType name="NDPortType">
    <operation name="initialization">
      <input message="tns:initializationRequest"/>
    </operation>
    <operation name="stop">
      <input message="tns:stopRequest"/>
    </operation>

    <operation name="Register">
      <input message="tns:RegisterRequest"/>
    </operation>

    <operation name="UnRegister">
      <input message="tns:UnRegisterRequest"/>
      <output message="tns:UnRegisterResponse"/>
    </operation>
    <operation name="getService">
      <input message="tns:GetServiceRequest"/>
      <output message="tns:GetServiceResponse"/>
    </operation>
    <operation name="getService_ByInstanceURL">
      <input message="tns:GetService_ByInstanceURLRequest"/>
      <output message="tns:GetService_ByInstanceURLResponse"/>
    </operation>
   <operation name="QueryService">
      <input message="tns:QueryService"/>
      <output message="tns:ReplyService"/>
    </operation>
   <operation name="QueryService_ByInstanceURL">
      <input message="tns:QueryService_ByInstanceURL"/>
      <output message="tns:ReplyService_ByInstanceURL"/>
    </operation>
    <operation name="Invalidate">
      <input message="tns:invalidateRequest"/>
    </operation>

  </portType>


  <!-- binding declns -->
  <binding name="NDBinding" type="tns:NDPortType">

    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http" />

    <operation name="initialization">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>


    <operation name="stop">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>


    <operation name="Register">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>
    <operation name="UnRegister">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
     <output>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
    <operation name="getService">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>

    <operation name="getService_ByInstanceURL">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>

    <operation name="QueryService">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>

    <operation name="QueryService_ByInstanceURL">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>

    <operation name="Invalidate">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:NDService"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>
  </binding>
```

```
</definitions>
```

<u>NDConcrete.wsdl</u>

```
<definitions name="urn:NDService"
             targetNamespace="urn:NDService"
             xmlns:tns="urn:NDService"
             xmlns:typens="urn:NDService"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

 <import namespace="urn:NDService" location="NDInterface.wsdl" />
 <!-- service decln -->
  <service name="NDServiceX">
    <port name="ND" binding="tns:NDBinding">
      <soap:address location="http://localhost:8080/services/ND"/>
    </port>
  </service>
</definitions>
```

5. Universal Repository (UR) Service

The Universal Repository service is a centralized repository where references to services distributed across the network are registered to be retrieved by ND services when the service discovery process (see section 4.3) in the local area leads to find an empty list of services.

If an address of a universal repository is provided by the user in the configuration file upon WSAMI Middleware startup (see section 3.2), the UR service will be used by the local ND service to make available local running services and to let local application discover remote services that could not be discovered without UR.

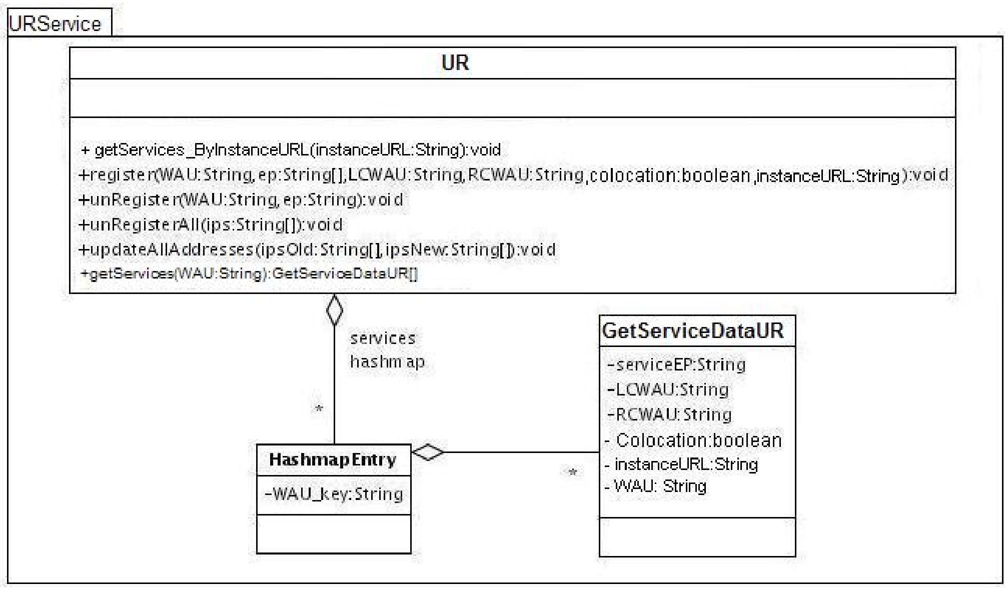
URService
UR
+ getServices_ByInstanceURL(instanceURL:String):void
+register(WAU:String,ep:String[],LCWAU:String,RCWAU:String,colocation:boolean,instanceURL:String):void
+unRegister(WAU:String,ep:String):void
+unRegisterAll(ips:String[]):void
+updateAllAddresses(ipsOld:String[],ipsNew:String[]):void
+getServices(WAU:String):GetServiceDataUR[]
services
hashmap
*
HashmapEntry
-WAU_key:String
*
GetServiceDataUR
-serviceEP:String
-LCWAU:String
-RCWAU:String
- Colocation:boolean
- instanceURL:String
- WAU: String

Figure 20 UR service class diagram

In the UR service class diagram in Figure 20 you can see that the service provides functionalities for service registration and unregistration. These methods are called by ND service that is in charge of registering (unregistering) on the UR each locally deployed (undeployed) service. The registration is done with `register` method requiring the WAU identifier of the service, all the endpoints associated (one different service endpoint is associated to the service for each corresponding IP addresses of the node), the WAU identifiers of customizers (if service customization is enabled), the colocation flag and the URL of the service instance.

The universal repository can be queried by ND services through the method `getServices` returning an array of `GetServiceDataUR` each entry storing information (WAU, endpoint and URL instance of the service, Local and Remote customizer WAU and collocation flag) about a service matching the WAU identifier. The service search can also be executed by service instance URL with `getServices_ByInstanceURL` method that returns the same array data structure as `getServices`.

If the node's IP address(es) change (see discussion about dynamic IP address change in Registration algorithm in section 4.2) the endpoints associated to the local services registered on the UR service must be updated so that they can continue to be reachable by remote nodes.

To keep the state of the UR service repository consistent with the system, when the IP address of a node change, the ND service calls the method `updateAllAddresses` with the old and the new node's addresses, and when the WSAMI Middleware shuts down, the ND calls the method `unRegisterAll` using as input all the node's IP addresses used to register services on the UR service. All the services associated to these addresses will be unregistered.

5.1. WSAMI and WSDL files

URAbstract.wsami

```
<?xml version="1.0" encoding="UTF-8"?>
<wsami xmlns="http://www-rocq.inria.fr/arles/wsami"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami WSAMI.xsd">
   <Definition name="UR" targetNamespace="http://localhost:8080/wsami/">
     <Abstract name="UR">
       <Interface hrefSchema="http://localhost:8080/wsami/URInterface.wsdl"/>
     </Abstract>
   </Definition>
</wsami>
```

URService.wsami

```
<?xml version="1.0" encoding="UTF-8"?>
<wsami xmlns="http://www-rocq.inria.fr/arles/wsami"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami WSAMI.xsd">
   <Definition name="UR" targetNamespace="http://localhost:8080/wsami/">
     <Service name="UR">
        <Abstract hrefSchema="http://localhost:8080/wsami/URAbstract.wsami"/>
        <Concrete hrefSchema="http://localhost:8080/wsami/URConcrete.wsdl"/>
     </Service>
   </Definition>
</wsami>
```

URInterface.wsdl

```
<?xml version="1.0" ?>

<definitions name="urn:URService"
             targetNamespace="urn:URService"
             xmlns:tns="urn:URService"
             xmlns:typens="urn:URService"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">


<!-- type defs -->
  <types>
    <xsd:schema targetNamespace="urn:URService">

     <xsd:complexType name="GetServiceDataUR">
       <xsd:sequence>
        <xsd:element name="WAU" type="xsd:string"/>
        <xsd:element name="serviceEP" type="xsd:string"/>
        <xsd:element name="instanceURL" type="xsd:string"/>
        <xsd:element name="LCWAU" type="xsd:string"/>
        <xsd:element name="RCWAU" type="xsd:string"/>
        <xsd:element name="colocation" type="xsd:boolean"/>
       </xsd:sequence>
     </xsd:complexType>

     <xsd:complexType name="ArrayOfGetServiceDataUR">
       <xsd:complexContent>
```

```
        <xsd:restriction base="soapenc:Array">
          <xsd:sequence>
            <xsd:element name="strings" minOccurs="0" maxOccurs="unbounded"
                    type="typens:GetServiceDataUR"/>
          </xsd:sequence>
        </xsd:restriction>
       </xsd:complexContent>
     </xsd:complexType>

     <xsd:complexType name="ArrayOfString">
       <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
          <xsd:sequence>
            <xsd:element name="strings" minOccurs="0" maxOccurs="unbounded"
                    type="xsd:string"/>
          </xsd:sequence>
        </xsd:restriction>
       </xsd:complexContent>
     </xsd:complexType>


    </xsd:schema>
  </types>


<!-- message declns -->

  <message name="registerRequest">
    <part name="WAU" type="xsd:string"/>
    <part name="serviceEP" type="typens:ArrayOfString"/>
    <part name="instanceURL" type="xsd:string"/>
    <part name="LCWAU" type="xsd:string"/>
    <part name="RCWAU" type="xsd:string"/>
    <part name="colocation" type="xsd:boolean"/>
  </message>
  <!-- message name="registerResponse"/-->

  <message name="unRegisterRequest">
    <part name="WAU" type="xsd:string"/>
    <part name="serviceEP" type="xsd:string"/>
  </message>
  <!-- message name="unRegisterResponse"/-->

  <message name="unRegisterAllRequest">
    <part name="ips" type="typens:ArrayOfString"/>
  </message>

  <message name="updateAllAddressesRequest">
    <part name="ipsOld" type="typens:ArrayOfString"/>
    <part name="ipsNew" type="typens:ArrayOfString"/>
  </message>

  <message name="getServicesRequest">
    <part name="WAU" type="xsd:string"/>
  </message>
  <message name="getServicesResponse">
    <part name="services" type="typens:ArrayOfGetServiceDataUR"/>
  </message>

  <message name="getServicesRequest_ByInstanceURL">
```

```
    <part name="instanceURL" type="xsd:string"/>
  </message>
  <message name="getServicesResponse_ByInstanceURL">
    <part name="services" type="typens:ArrayOfGetServiceDataUR"/>
  </message>


<!-- port type declns -->

  <portType name="URInterface">
    <operation name="register">
      <input message="tns:registerRequest"/>
      <!--output message="tns:registerResponse"/-->
    </operation>

    <operation name="unRegister">
      <input message="tns:unRegisterRequest"/>
      <!--output message="tns:unRegisterResponse"/ -->
    </operation>

    <operation name="unRegisterAll">
      <input message="tns:unRegisterAllRequest"/>
    </operation>

    <operation name="updateAllAddresses">
      <input message="tns:updateAllAddressesRequest"/>
    </operation>

    <operation name="getServices">
      <input message="tns:getServicesRequest"/>
      <output message="tns:getServicesResponse"/>
    </operation>

    <operation name="getServices_ByInstanceURL">
      <input message="tns:getServicesRequest_ByInstanceURL"/>
      <output message="tns:getServicesResponse_ByInstanceURL"/>
    </operation>

  </portType>


<!-- binding declns -->

  <binding name="URSOAPBinding" type="tns:URInterface">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="register">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.UR.Service"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>

    <operation name="unRegister">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.UR.Service"
                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>

    <operation name="unRegisterAll">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.UR.Service"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>

    <operation name="updateAllAddresses">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.UR.Service"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
    </operation>

    <operation name="getServices">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.UR.Service"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="htto://Service.org"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>

    <operation name="getServices_ByInstanceURL">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.UR.Service"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="htto://Service.org"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
</definitions>
```

<u>URConcrete.wsdl</u>

```
<definitions name="urn:URService"
             targetNamespace="urn:URService"
             xmlns:tns="urn:URService"
             xmlns:typens="urn:URService"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
<import namespace="urn:URService" location="URInterface.wsdl" />
 <!-- service decln -->
  <service name="UR_Service">
    <port name="UR" binding="tns:URSOAPBinding">
      <soap:address location="http://localhost:8080/services/UR"/>
    </port>
  </service>
</definitions>
```

6. Gateway service

The gateway service can be deployed on multi-homed hosts to enable the routing from an ad-hoc wireless network to an infrastructure-based network (either wireless or wired).
To enable the routing on the host, the WSAMI Middleware requires the deployment of the Gateway service and the right routing configuration (for details about the Gateway Service configuration see [USERGUIDE]).

The ND service makes use of the Gateway service in the service discovery algorithm for wireless ad-hoc nodes (described in section 4.3). In step A3 of this algorithm, if a service is not found in the local ad-hoc wireless network, the ND looks for an instance of Gateway service in the local area and if found, its address will be set in the routing table as default gateway to allow the node to access services running outside the local network.

As you can see in the WSDL service abstract interface in the following section, the only method provided by the service is `GetGWIPAddr`. This method retrieves network card information using the functionalities offered by the tools described in section 3.5.1 and returns the IP address of the card connected to the infrastructure-based network.
The ND of the ad-hoc node will add this address to the routing table using the tools described in section 3.5.1.

This service is available only for the Linux version of the WSAMI Middleware.

6.1. WSAMI and WSDL files

GWAbstract.wsami

```
<?xml version="1.0" encoding="UTF-8"?>
<wsami
        xmlns="http://www-rocq.inria.fr/arles/wsami"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami WSAMI.xsd">
 <Definition name="GW" targetNamespace="http://localhost:8080/wsami/">
  <Abstract name="GW">
   <Interface hrefSchema="http://localhost:8080/wsami/GWInterface.wsdl"/>
  </Abstract>
 </Definition>
</wsami>
```

GWService.wsami

```
<?xml version="1.0" encoding="UTF-8"?>
<wsami
        xmlns="http://www-rocq.inria.fr/arles/wsami"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www-rocq.inria.fr/arles/wsami WSAMI.xsd">

 <Definition name="GW" targetNamespace="http://localhost:8080/wsami/">
  <Service name="GW">
   <Abstract hrefSchema="http://localhost:8080/wsami/GWAbstract.wsami"/>
   <Concrete hrefSchema="http://localhost:8080/wsami/GWConcrete.wsdl"/>
  </Service>
 </Definition>
</wsami>
```

GWInterface.wsdl

```
<?xml version="1.0" ?>

<definitions name="urn:GWService"
             targetNamespace="urn:GWService"
             xmlns:tns="urn:GWService"
             xmlns:typens="urn:GWService"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

<!-- message declns -->
  <message name="GetGWIPAddrRequest"/>
  <message name="GetGWIPAddrResponse">
   <part name="addr" type="xsd:string"/>
  </message>

<!-- port type declns -->
  <portType name="GWInterface">
    <operation name="GetGWIPAddr">
      <input message="tns:GetGWIPAddrRequest"/>
      <output message="tns:GetGWIPAddrResponse"/>
    </operation>
  </portType>

<!-- binding declns -->
  <binding name="GWSOAPBinding" type="tns:GWInterface">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetGWIPAddr">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
                   namespace="http://www.GW.Service"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="htto://Service.org"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
```

```
</definitions>
```

GWConcrete.wsdl

```
<definitions name="urn:GWService"
             targetNamespace="urn:GWService"
             xmlns:tns="urn:GWService"
             xmlns:typens="urn:GWService"
             xmlns:xsd="http://www.w3.org/1999/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
<import namespace="urn:GWService" location="GWInterface.wsdl" />
 <!-- service decln -->
  <service name="GW_Service">
    <port name="GW" binding="tns:GWSOAPBinding">
      <soap:address location="http://localhost:8080/services/GW"/>
    </port>
  </service>
</definitions>
```

7. XSD defintions

7.1. WSAMI.xsd

```
<xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www-rocq.inria.fr/arles/wsami"
      targetNamespace="http://www-rocq.inria.fr/arles/wsami"
      elementFormDefault="qualified" attributeFormDefault="unqualified" >


<xsd:element name="wsami">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Definition" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="Definition">
  <xsd:complexType >
    <xsd:choice minOccurs="1" maxOccurs="1" >
       <xsd:element ref="Abstract" />
       <xsd:element ref="Service" />
       <xsd:element ref="Customizer"/>
    </xsd:choice >
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="targetNamespace" type="xsd:anyURI"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="Abstract">
  <xsd:complexType >
    <xsd:sequence>
       <xsd:element name="Interface"  type="XMLDocumentType"/>
       <xsd:element name="Conversation" type="XMLDocumentType"  minOccurs="0"/>
```

```
        <xsd:element ref="ServiceQoS" minOccurs="0" />
        <xsd:element ref="ConnectorQoS" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="XMLDocumentType">
    <xsd:attribute name="hrefSchema" type="xsd:anyURI"/>
  </xsd:complexType>

  <xsd:element name="Service">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Abstract" type="XMLDocumentType"/>
        <xsd:element name="Concrete" type="XMLDocumentType"/>
        <xsd:element ref="Required" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Required">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="ReqService" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element  name="ReqService">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Abstract" type="XMLDocumentType"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="optional" type="xsd:boolean" default="false"/>
      <xsd:attribute name="instance" type="xsd:boolean" default="false"/>
      <xsd:attribute name="multiple" type="xsd:boolean" default="false"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Customizer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="QoSCriterion" maxOccurs="unbounded"/>
        <xsd:element name="Local" type="XMLDocumentType"/>
        <xsd:element name="Remote" type="XMLDocumentType"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="Colocation" type="xsd:boolean" default="false"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="ServiceQoS">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="QoSCriterion" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
</xsd:element>

<xsd:element name="ConnectorQoS">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="QoSCriterion" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="QoSCriterion">
  <xsd:complexType>
    <xsd:attribute name="name" type="QoSType"/>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="QoSType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="transactionalService"/>
    <xsd:enumeration value="security"/>
    <xsd:enumeration value="saveBandwidth"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

7.2. deploy.xsd

```
<xsd:schema
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns="http://www-rocq.inria.fr/arles/wsami"
       targetNamespace="http://www-rocq.inria.fr/arles/wsami"
       elementFormDefault="qualified" attributeFormDefault="unqualified" >


<xsd:element name="deploy">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="service" maxOccurs="unbounded" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="service">
  <xsd:complexType>
    <xsd:sequence>
       <xsd:element name="wad" type="wsamifile" />
       <xsd:element name="wsd" type="wsamifile" />
       <xsd:element name="wcd" type="wsamifile" minOccurs="0"/>
       <xsd:element name="wau" type="wsamiURI" />
       <xsd:element ref="implementation"/>
       <xsd:element name="scope" type="scopeType" />
       <xsd:element name="instance_URL" type="instanceURLType"  minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="customizer" type="customizerType" />
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
```

```
<xsd:complexType name="wsamifile">
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="wsamiURI">
  <xsd:attribute name="name" type="xsd:anyURI"/>
</xsd:complexType>

<xsd:element name="implementation">
  <xsd:complexType >
    <xsd:sequence>
       <xsd:element name="class"  type="wsamifile"/>
       <xsd:element name="type" >
         <xsd:complexType>
           <xsd:attribute name="name" type="implementationType" />
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="package" type="wsamifile" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="implementationType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="javaclass"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="customizerType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="remote"/>
    <xsd:enumeration value="local"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="scopeType">
  <xsd:attribute name="name" type="scopeEnum"/>
</xsd:complexType>
<xsd:simpleType name="scopeEnum">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Application"/>
    <xsd:enumeration value="Session"/>
    <xsd:enumeration value="Request"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="instanceURLType">
  <xsd:attribute name="name" type="xsd:anyURI" />
</xsd:complexType>

<!-- undeployment XML schema -->

<xsd:element name="undeploy">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="service" maxOccurs="unbounded" >
        <xsd:complexType>
          <xsd:sequence>
             <xsd:element name="wau" type="wsamiURI" />
          </xsd:sequence>
          <xsd:attribute name="name" type="xsd:string" />
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

7.3. deploy_install.xsd

```
<xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www-rocq.inria.fr/arles/wsami"
      targetNamespace=http://www-rocq.inria.fr/arles/wsami
      elementFormDefault="qualified" attributeFormDefault="unqualified">

<xsd:element name="deploy">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="service" maxOccurs="unbounded" />
         </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name="service">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="wad" type="wsamifile" />
         <xsd:element name="wsd" type="wsamifile" />
         <xsd:element name="wcd" type="wsamifile" minOccurs="0" />
         <xsd:element name="wau" type="wsamiURI" />
         <xsd:element ref="implementation" />
         <xsd:element name="scope" type="scopeType" />
        <xsd:element name="instance_URL" type="instanceURLType"minOccurs="0"/>
         <xsd:element ref="install" />
      </xsd:sequence>
      <xsd:attribute name="customizer" type="customizerType" />
      <xsd:attribute name="name" type="xsd:string" />
   </xsd:complexType>
</xsd:element>

<xsd:complexType name="wsamifile">
      <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="wsamiURI">
      <xsd:attribute name="name" type="xsd:anyURI" />
</xsd:complexType>
<xsd:element name="implementation">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="class" type="wsamifile" />
                  <xsd:element name="type">
                    <xsd:complexType>
                     <xsd:attribute name="name" type="implementationType" />
                    </xsd:complexType>
                  </xsd:element>
                        <xsd:element name="package" type="wsamifile" />
                  </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:simpleType name="implementationType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="javaclass" />
            </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="customizerType">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="remote" />
                  <xsd:enumeration value="local" />
            </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="scopeType">
      <xsd:attribute name="name" type="scopeEnum" />
</xsd:complexType>
<xsd:simpleType name="scopeEnum">
            <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="Application" />
                  <xsd:enumeration value="Session" />
                  <xsd:enumeration value="Request" />
            </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="instanceURLType">
      <xsd:attribute name="name" type="xsd:anyURI" />
</xsd:complexType>

<!-- undeployment XML schema -->
<xsd:element name="undeploy">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="service" maxOccurs="unbounded">
                     <xsd:complexType>
                        <xsd:sequence>
                              <xsd:element name="wau" type="wsamiURI" />
                        </xsd:sequence>
                     </xsd:complexType>
                  </xsd:element>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<!-- install -->
<xsd:element name="install">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="file" maxOccurs="unbounded">
                     <xsd:complexType>
                        <xsd:attribute name="type" type="fileType" />
                        <xsd:attribute name="name" type="xsd:string" />
                        <xsd:attribute name="location" type="xsd:string" />
                     </xsd:complexType>
                  </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" />
      </xsd:complexType>
</xsd:element>

<xsd:simpleType name="fileType">
```

```
             <xsd:restriction base="xsd:string">
                   <xsd:enumeration value="descr" />
                   <xsd:enumeration value="implClass" />
                   <xsd:enumeration value="implJar" />
             </xsd:restriction>
</xsd:simpleType>
<!-- install -->

</xsd:schema>
```

7.4. install.xsd

```
<xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www-rocq.inria.fr/arles/wsami"
      targetNamespace="http://www-rocq.inria.fr/arles/wsami"
      elementFormDefault="qualified" attributeFormDefault="unqualified" >


<xsd:element name="install">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="file" maxOccurs="unbounded" >
        <xsd:complexType>
           <xsd:attribute name="type" type="fileType" />
           <xsd:attribute name="name" type="xsd:string" />
           <xsd:attribute name="location" type="xsd:anyURI" />
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="fileType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="descr"/>
    <xsd:enumeration value="implClass"/>
    <xsd:enumeration value="implJar"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Bibliography

USERGUIDE

WSAMI Middleware User's guide

DEVELGUIDE

WSAMI Middleware Developer's guide

CSOAP-GUIDE

CSoap User Guide

CSOAP-ARCH

CSoap Architecture

STEI

J. Steinberg and J. Pasquale. A Web middleware architecture for dynamic customization of content for wireless clients. In Proceedings of the WWW'02 Conference, 2002