KAON SERVER Demonstrator

Daniel Oberle, Steffen Staab, Rudi Studer, Raphael Volz

University of Karlsruhe,
Institute for Applied Informatics and Formal Descriptions Methods (AIFB)
D-76128 Karlsruhe
email: {`lastname`}`@aifb.uni-karlsruhe.de`

Identifier	Del 7
Class	Deliverable
Version	1.1
Date	07-15-2004
Status	Final
Distribution	Public
Lead Partner	AIFB

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Contents

Executive Summary

KAON SERVER can be considered as Application Server for the Semantic Web whose design and development are based on existing Application Servers. However, we apply and augment their underlying concepts for use in the Semantic Web and integrate semantic technology within the server itself.

This deliverable describes a Demonstrator (KAON SERVER release 1.0) from a user's point of view. A detailed discussion of the server's analysis, requirements, design and implementation is given in [12]. A detailed discussion of the contribution to the Middleware community is given in [10].

Part of this work has been done in cooperation with partners inside and outside WonderWeb. In particular, we are indebted to Marta Sabou, Vrije Universiteit Amsterdam, The Netherlands, as well as Debbie Richards, MacQuarie University Syndney, Australia, for their fruitful work on the ontology presented in section 4. Also Sean Bechhofer from the University of Manchester who maintains the OilEd ontology editor and invested great efforts for the KAON SERVER adaptation.

The new version 1.1 of this deliverable extends 1.0 by the following points:

- OilEd demonstrator (new Section 7)
- Semantic Web Service Connector (new Section 6)
- Association Management (Section 3.4)
- Interceptors (new section 3.3)
- Components update registry (section 3.2)
- Several bugfixes and general improvements

1 General Information

KAON SERVER can be considered as an Application Server for the Semantic Web (AS-SW) facilitating reuse of existing software modules, e.g. ontology stores, editors, and inference engines and, thus, the development and maintenance of comprehensive Semantic Web applications. In [12] we describe analysis, design and implementation in detail, [10] discusses the contribution for the Middleware community. The following subsections will document the KAON SERVER Demonstrator (release 1.0) from a user's point of view. In particular we will show how to install and start the server (section 2), how to deploy (3) and discover components (4) and how to work with client-side surrogates (5). Section 6 talks about the Semantic Web Services Connector that automatically generates OWL-S descriptions, Section 7 about the OilEd demo. Conclusion and future work are presented in section 8.

1.1 Relationship to KAON

KAON [5] is an open-source ontology management infrastructure targeted for semantics-driven business applications. It includes a comprehensive tool suite allowing easy ontology management and application. Important focus of KAON is on integrating traditional technologies for ontology management and application with those used typically in business applications, such as relational databases. KAON is developed by the Research Center for Information Technologies (FZI) and the Institute AIFB, both at the University of Karlsruhe. For a detailed technical description please confer to `http://kaon.semanticweb.org` as well as the KAON Developer's Guide [8].

```
<any dir>
   |- build
   |    |- kaon_build_root
   |    |       | - apionrdf
   |    |       | - ...
   |    |- kaon-ext_build_root
   |    |       | - kaonserver
   |    |       | - ...
  ...
   |- kaon
   |    |- 3rdparty
   |    |- apionrdf
   |    |- apiproxy
   |    |- ...
   |- kaon-ext
   |    |- 3rdparty
   |    |- kaonserver
   |    |- ...
```

KAON Extensions are a set of software modules that are optional to the KAON tool suite but rely on it. In contrast to KAON, the modules in KAON Extensions are disjoint, i.e. they do not rely on other modules within the project. However, they all rely at least on the kaonapi module from the KAON project. Hence, for development, the whole KAON project is required (please cf. [8]). Both projects should be checked out in parallel resulting a file-structure like follows:

1.2 Obtaining KAON SERVER

KAON SERVER is part of KAON Extensions and works with Sun's JDK 1.4.1_03 (available at `http://java.sun.com/j2se/`). Its source code, as well as released binaries can be obtained from `http://sourceforge.net/projects/kaon-ext`, including public CVS access.

The "kaonserver" module in KAON Extensions provides its own build.xml in the corresponding directory (it includes common.xml where some global constants are defined). The build file references some libraries from <any dir>/kaon/3rdparty and from <any dir>/build/ kaon_build_root/<module-dir>/lib. The remaining libraries are stored in <any dir>/kaon-ext/3rdparty.

A successful build results in a corresponding directory located at <any dir>/build/-kaon-ext_build_root/kaonserver. The module features a source and binary distribution zip-archive, javadocs, a copy of each required library and generated scripts. It is not required to have the KAON project checked out when working with the binary distribution of the KAON SERVER.

1.3 Java Package Overview

The Java packages of the kaonserver module are organized akin to the conceptual architecture depicted in [12] (also cf. Figure 6), i.e. they are divided in components, management, connectors and client where the latter holds all the client-side surrogates. We will explain the packages below:

edu.unika.aifb.kaon.server Only holds the interface Constants which, as the name suggests, contains all the constants required throughout the project. If a class needs some constants, it just has to implement the interface. Also, there is the class StartServer that eventually becomes the start script.

edu.unika.aifb.kaon.server.client Contains all the client-side surrogates for components written so far (hence the package name "client"). All surrogates are labelled Remote-<original class name or component name>. See section 5 for a description of the surrogates.

edu.unika.aifb.kaon.connectors Holds all the code of connector MBeans.

edu.unika.aifb.kaon.interceptors All the code related to Interceptors: an interface of the same name, an AbstractInterceptor and prototypical AuditingInterceptor and AuthenticationInterceptor.

edu.unika.aifb.kaon.management Holds system components that belong to the Management Core, i.e. the Component Loader as well as Association Management.

The Registry is a KAON ontology store, situated in the components package. The Microkernel is implemented by JBoss JMX (Java Management Implementations) MBeanServer, located in javax.management in jboss-jmx.jar.

edu.unika.aifb.kaon.components Contains classes and corresponding MBean interfaces for all functional and proxy components written so far.

edu.unika.aifb.kaon.server.test Here one can find all the classes for test purposes. Note that those are not maintained and might be outdated. However, the developer might find ideas for his or her own client code.

2 Starting the server

In package edu.unika.aifb.kaon.server, the Java class StartServer bootstraps the server with all connectors and the management core's system components. In the build or binary release, there is a script automatically starting this class in <any dir>/build/kaon-ext_build_root/kaonserver/release/bin/startserver.bat. You are required to start it from the release/bin directory or a correct configuration of the KAON SERVER is not ensured. Note that this script includes the invocation of the RMI registry which is required for the RMI connector. If you don't use the script but start the class Prototype manually, you have to start the RMI registry with a proper classpath set to KAON SERVER's classes. The following enumeration lists what happens during start-up:

1. Creation of the Management Core
   - Creation of the Kernel
   - Deployment of the Registry
   - Deployment of the Component Loader
   - Deployment of the Association Management
2. Creation of the Connectors
   - Deployment of the HTTP Adaptor GUI
   - Deployment of the RMI Connector
   - Deployment of the SOAP Connector
   - Deployment of the Semantic Web Services Connector
3. Descriptions
   An ontological description of every system component (situated in /resources/-descriptions) is included in the registry (cf. 4).
4. Paths
   Paths are being set for the component loader's hot deployment directory and some other configurations. It is important to have the working directory set to /release/bin when calling the startserver.bat script.

After a successful start-up, the user may interact manually with the server by working with the management console (`http://localhost:8082`) or view the contents of the registry[1]. For the latter, KAON OIModeller [8] is required (cf. also section 4). Component deployment and component discovery are discussed in the following sections.

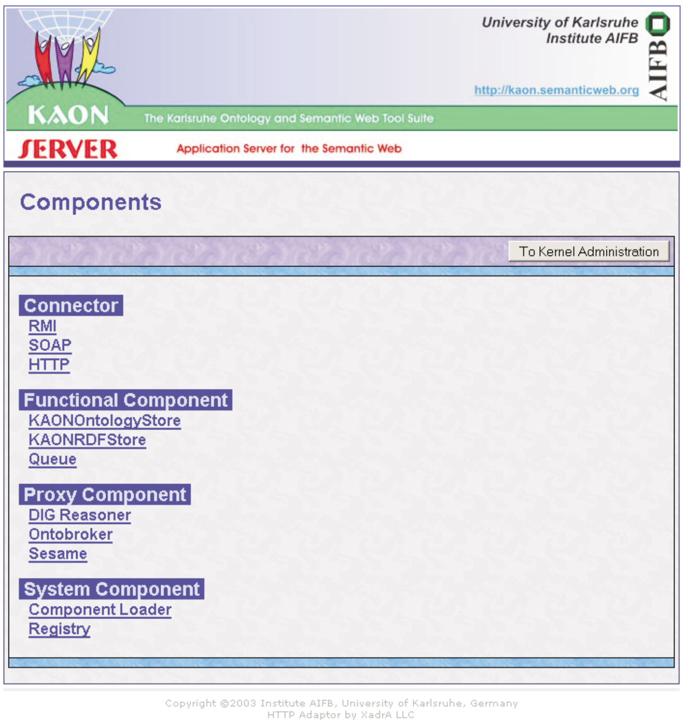

Figure 1: Screenshot of HTTP Connector.

3 Deploying components

There are two ways of deploying a component to the server's Microkernel. Typically, a developer realizes deployment by explicit Java code in his or her client. Another option is to manually deploy a component by copying a description file into a specified hot deployment directory. Both possibilities are supported by the component loader and discussed in the following subsections.

3.1 Deployment in a client

Normally, a JMX developer would hard-code deployment in his or her code. Like shown in the example below, a new javax.management.ObjectName has to be created and given as argument to registerMBean together with the actual MBean. The MBean in our case is a Sesame RDF store. Note that this method of deployment does whether enter the MBean in the registry nor apply the association management.

[1]Note that the console is an evaluation version only, i.e. functionality is restricted.

```
ObjectName name =
 new ObjectName("Functional Component:name=SesameStore");
server.registerMBean(sesame, name);
```

Thus, we developed the component loader system component for convenience. A client application has to create a surrogate for it in a first step. After that, the deploy() method takes an ontological description of the component which is automatically inserted into the registry. Also, associations to other components are detected and automatically applied by the Association Management System Component (cf. 3.4).

```
Properties props = new Properties();
props.put(CONNECTION, RMI);
props.put(RMI_HOST,"localhost");
props.put(RMI_PORT,"1099");
props.put(RMI_NAME,"/jmx/RMIConnector");
RemoteComponentLoader rcl
 = new RemoteComponentLoader(props);
rcl.deploy("file:///SesameComponent.kaon");
```

The exemplary description `file:///SesameComponent.kaon` may look like below and conforms to the registry's ontology (cf. 4). In essence, a component's description is made up of instances.

```
<softwaremodule:SoftwareModule rdf:ID="Sesame">
    <softwaremodule:presents rdf:resource="#SesameProfile"/>
    <softwaremodule:implements rdf:resource="#SesameComponent"/>
</softwaremodule:SoftwareModule>


<semanticwebprofile:SesameStore rdf:ID="SesameProfile">
    <softwaremodule:presentedBy rdf:resource="#Sesame"/>
    <semanticwebprofile:queryLanguage>SeRQL</semanticwebprofile:queryLanguage>
    <semanticwebprofile:supportsTransactions>No</semanticwebprofile:supportsTransactions>
    <semanticwebprofile:persistent>No</semanticwebprofile:persistent>
    <semanticwebprofile:persistent>None</semanticwebprofile:persistent>
</semanticwebprofile:SesameStore>

<implementation:ProxyComponent rdf:ID="SesameComponent">
    <softwaremodule:implementedBy rdf:resource="#Sesame"/>
    <implementation:hasCodeDetails rdf:resource="#SesameCodeDetails"/>
</implementation:ProxyComponent>

<implementation:CodeDetails rdf:ID="SesameCodeDetails">
    <implementation:name>Proxy Component:name=Sesame</implementation:name>
    <implementation:code>edu.unika.aifb.kaon.server.components.SesameComponent</implementation:code>
    <implementation:version>1.0</implementation:version>
</implementation:CodeDetails>
```

3.2 Manual (Hot) Deployment

The component loader offers the possibility to define a directory which is periodically being scanned for new description files. This directory can be set also via the management console. Files with extensions ".kaon", ".xml" and ".rdf" are regarded. Such files have to contain a valid KAON description according to the management ontology and at least one instance of the Component concept (or subconcepts thereof) like depicted in the subsection above. Several component descriptions in a file are possible. [2]

As soon as such a file is copied in the hot deployment directory, it is provided as argument to the component loader's deploy method that in turn registers and starts it and also enters the description in the registry and apply the association management. When a description file is moved or deleted the respective component(s) are undeployed. The hot deployment directory is set to kaonserver/resources/hotdeploy as default.

For convenience, we came up with several copy-by-example descriptions for all functional components available so far. All of them located in kaonserver/resources/descriptions. They may just be moved to kaonserver/resources/hotdeploy for instant deployment.

In the new version of KAON SERVER, components update some information automatically in the registry. This was necessary to keep information replica consistent both in form of component class members and attributes in the registry. To give an example: if the user opens a new model in RDFComponent (a KAON RDF store) then the corresponding attribute in the registry is automatically updated.

3.3 Defining Interceptors

Interceptors are software entities that monitor a request and modify them. Typically, aspects orthogonal to any application are realized by interceptors, e.g. auditing, security or transactions. They are a means to realize aspect oriented programming in the context of JMX. The package `edu.unika.aifb.kaon.server.interceptors` defines an interface `Interceptor`, an `AbstractInterceptor` as well as a prototypical `AuditingInterceptor` and `AuthenticationInterceptor`.

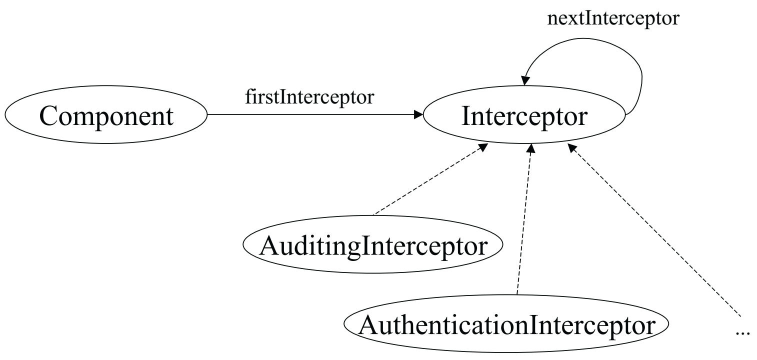

Figure 2: Conceptual model for defining interceptors

Figure 2 shows how the interceptors are conceptually represented in the ontology (cf. also Section 4). Every *Component* may be deployed with an arbitrary sequence of *Interceptor*s or specializations thereof. The example below depicts a concrete example where a KAONRDFStoreComponent is deployed with 3 interceptors.

[2]Instead of the directory, component descriptions can also be loaded from a URL. A user can invoke the component loader's deploy method in the management console and provide the URL as string argument.

```
...

<implementation:FunctionalComponent
rdf:ID="KAONRDFStoreComponent">
    <softwaremodule:implementedBy rdf:resource="#KAONRDFStore"/>
    <implementation:hasCodeDetails rdf:resource="#KAONRDFStoreCodeDetails"/>
    <implementation:firstInterceptor rdf:resource="#Interceptor1"/>
</implementation:FunctionalComponent>

...

<implementation:AuditingInterceptor rdf:ID="Interceptor1">
    <implementation:logFile>c:\test.log</implementation:logFile>
    <implementation:nextInterceptor rdf:resource="#Interceptor2"/>
</implementation:AuditingInterceptor>

<implementation:SecurityInterceptor rdf:ID="Interceptor2">
    <implementation:nextInterceptor rdf:resource="#Interceptor3"/>
</implementation:SecurityInterceptor>

<implementation:AuditingInterceptor rdf:ID="Interceptor3">
    <implementation:logFile>c:\test2.log</implementation:logFile>
</implementation:AuditingInterceptor>
```

The Component Loader detects such descriptions and constructs a proxy around the actual MBean by means of `java.lang.reflection` where the `Interceptor` implementations play the role of invocation handlers containing the MBean. The preferred syntax for components' names with interceptors is "Functional Component:name=Sesame, interceptor=auditing".

The `AuthenticationInterceptor` is kept very simple just to give a proof of concept. The provider deploys a component with such an interceptor along user and password. Surrogates are expected to transmit user and password with every invocation as last two arguments (currently only RemoteSesame supports this). If authentication is valid the interceptor removes user and password from the argument list and calls the actual method.

This approach is suboptimal as getAttribute and setAttribute methods cannot be extended by user and password in the arglist. Also the management console is actually not allowed to perform invocations in this way. Another drawback of this approach is that user and password are written in the registry. The deployer specifies both in the deployment descriptor which is entered in the registry.

3.4 Association Management

Descriptions of components may feature ontological associations between components, e.g. dependsOn, receivingEventsFrom, preventUnloading etc. The Association Management System Component is there to put such associations into action. It mainly cooperates with the Component Loader and the Registry. Regarding dependencies it plays a similar role to org.jboss.system.ServiceController in the JBoss Application Server. However, Association Management subsumes its functionality as it is there to manage also other kinds of associations and applies reasoning with the Registry.

The protocol for dependencies takes another approach than in JBoss. JBoss allows dependencies only between "Services" - we here allow dependencies between any kind of component. Hence, the protocol is looser. We do not define a lifecycle for MBeans like in JBoss. Instead, a component always is deployed, even though it is dependent on (not yet deployed) components. Only unloading of components is forbidden when they are in the dependency graph.

Protocol for deployment:

1. Component Loader (CL) enters description in registry
2. CL calls operationalizeAssociations at AssociationManagement (AM)
3. AM queries registry for associations (dependsOn, etc.)
4. AM reacts to those, e.g. by remembering the component name

Protocol for undeployment:

1. User/CL attempts to undeploy a component
2. before CL deletes contents from registry and undeploys it asks AM
3. Are there dependencies on the particular component?
4. If yes, exception is thrown; if no, undeployment is allowed

In v1.00, KAON SERVER is only able to operationalize “dependsOn”. One may define the association in a description file as follows. The example below expresses that a KAON ontology store relies on a KAON RDF store. Providing this description to the Component Loader does not require any more actions from the client/developer.

```
...
<implementation:FunctionalComponent rdf:ID="KAONComponent">
    <softwaremodule:implementedBy rdf:resource="#KAONOntologyStore"/>
    <implementation:hasCodeDetails rdf:resource="#KAONOntologyStoreCodeDetails"/>
    <implementation:dependsOn rdf:resource="file:/c:/descriptions/KAONRDFComponent.kaon#KAONRDFComponent"/>
</implementation:FunctionalComponent>
...
```

The drawback of this approach is that the identifying URI of the corresponding Component instance has to be known in advance. In most cases a client would query the registry at runtime for this URI and use the Association Management’s surrogate to apply the dependency.

```
Map parameters=new HashMap();
parameters.put(
 edu.unika.aifb.kaon.server.client.RemoteKAONConnection.SERVER_URI,
 "mbean://RMI@localhost:1099/jmx/RMIConnector/System%20Component?name=Registry"
 );
RemoteRegistry m_registry = new RemoteRegistry(parameters);
RemoteAssociationManagement m_am = new RemoteAssociationManagement(parameters);

m_am.addDependency(
 m_registry.getComponentURL("Functional Component:name=KAONOntoStore"),
 m_registry.getComponentURL("Functional Component:name=KAONRDFStore")
 );

...

m_am.releaseAssociations("Functional Component:name=KAONOntoStore");
```

Calling releaseAssociations on the Association Management will allow undeployment of the depending components again. If the user tries to undeploy a depending component by moving a description file out of the hotdeploy directory, the file is moved. However, the component remains deployed, the descriptions remains in the registry and the association remains. One has to invoke releaseAssociation on Association Management, unregister on RemoteMBeanServer and deleteDescription on Registry.

4 Discovering components

The registry, which is a simple ontology store, and its ontology play a central role in KAON SERVER. We are using the apionrdf implementation of the KAON API, i.e. the main memory transient version, as ontology store. Components can be described according to the management ontology and those descriptions can be given as argument to the Component Loader which in turn enters them in the registry. A client may discover a component it is in need of by querying the registry. All of that functionality is described in this section.

4.1 The ontology

KAON SERVER uses a management ontology as detailed in [9, 11, 10]. It takes a similar design to OWL-S [2] but has been adapted to describe software modules instead of web services. Figure 3 shows the ontology design in contrast to OWL-S.

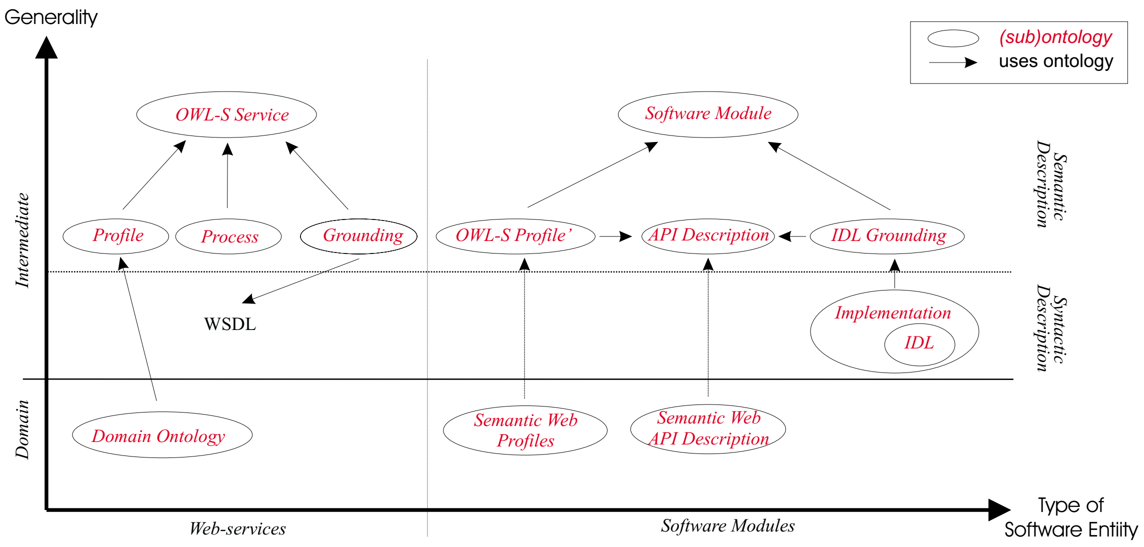

Figure 3: KAON SERVER's ontology

Each of the sub-ontologies is modelled in the KAON language and shipped with the distribution. The global logical namespace always starts with `http://kaon.semanticweb.org/kaon/server`. The logical URI of the registry is `http://kaon.semanticweb.org/kaon/server/registry` which is basically an empty KAON OIModel that includes the `http://kaon.semanticweb.org/kaon/server/registryscheme`. The latter includes all the subontologies listed below. This inclusion structure allows us to include component descriptions at run-time more easily. PhysicalURIs are usually logicalURI + ".kaon".

Software Module `/resources/ontology/softwaremodule.kaon`

Profile `/resources/ontology/profile.kaon`

API Description `/resources/ontology/apidescription.kaon`

IDL Grounding `/resources/ontology/idlgrounding.kaon`

Implementation `/resources/ontology/implementation.kaon`

IDL `/resources/ontology/idl.kaon`

Semantic Web Profile `/resources/ontology/semanticwebprofile.kaon`

Semantic Web API Description `/resources/ontology/semanticwebapidescription.kaon`

4.2 Component descriptions

The component loader (cf. 3) offers the possibility to deploy a component by means of an ontological description. Such a description basically features one instance of the Software Module concept. Technically, this description is stored in a KAON file that includes the registry. Below, we list an exemplary description for a Description Logic reasoner:

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

<!DOCTYPE rdf:RDF [
    <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <!ENTITY registry 'http://kaon.semanticweb.org/kaon/server/registryscheme#'>
    <!ENTITY softwaremodule 'http://kaon.semanticweb.org/kaon/server/softwaremodule#'>
    <!ENTITY profile 'http://kaon.semanticweb.org/kaon/server/profile#'>
    <!ENTITY apidescription 'http://kaon.semanticweb.org/kaon/server/apidescription#'>
    <!ENTITY idlgrounding 'http://kaon.semanticweb.org/kaon/server/idlgrounding#'>
    <!ENTITY idl 'http://kaon.semanticweb.org/kaon/server/idl#'>
    <!ENTITY implementation 'http://kaon.semanticweb.org/kaon/server/implementation#'>
    <!ENTITY semanticwebprofile 'http://kaon.semanticweb.org/kaon/server/semanticwebprofile#'>
    <!ENTITY semanticwebapidescription 'http://kaon.semanticweb.org/kaon/server/semanticwebapidescription#'>
]>

<?include-rdf
logicalURI="http://kaon.semanticweb.org/kaon/server/registryscheme"
physicalURI="file:/C:/Dev/KAON-ext/kaonserver/resources/ontologies/registryscheme.kaon"?>

<rdf:RDF
    xmlns="&registry;"
    xmlns:rdf="&rdf;"
    xmlns:softwaremodule="&softwaremodule;"
    xmlns:profile="&profile;"
    xmlns:apidescription="&apidescription;"
    xmlns:idlgrounding="&idlgrounding;"
    xmlns:idl="&idl;"
    xmlns:implementation="&implementation;"
    xmlns:semanticwebprofile="&semanticwebprofile;"
    xmlns:semanticwebapidescription="&semanticwebapidescription;"
>
```

The header of each description always starts with the definition and abbreviation of the required sub-ontologies' namespaces (as listed above) and the inclusion of registry-scheme.

```
softwaremodule:SoftwareModule rdf:ID="DIGReasoner">
    <softwaremodule:presents rdf:resource="#DIGReasonerProfile"/>
    <softwaremodule:implements rdf:resource="#DIGReasonerComponent"/>
</softwaremodule:SoftwareModule>


<semanticwebprofile:DIGReasoner rdf:ID="DIGReasonerProfile">
    <softwaremodule:presentedBy rdf:resource="#DIGReasoner"/>
    <semanticwebprofile:language>SHIQ</semanticwebprofile:language>
    <semanticwebprofile:actualReasoner>FaCT</semanticwebprofile:actualReasoner>
    <semanticwebprofile:holdsOntology>none</semanticwebprofile:holdsOntology>
    <semanticwebprofile:url>http://xyz</semanticwebprofile:url>
</semanticwebprofile:DIGReasoner>

<implementation:ProxyComponent rdf:ID="DIGReasonerComponent">
    <softwaremodule:implementedBy rdf:resource="#DIGReasoner"/>
    <implementation:hasCodeDetails rdf:resource="#DIGReasonerCodeDetails"/>
</implementation:ProxyComponent>
```

Central to each description is an instance of SoftwareModule which presents a Profile (in this case a specialization thereof located in the Semantic Web Profiles sub-ontology) and implements a SoftwareModuleImplementation (which is again specialized in this example). It is important to include the inverse properties presentedBy and implementedBy for querying.

```
<implementation:CodeDetails rdf:ID="DIGReasonerCodeDetails">
    <implementation:name>Proxy Component:name=DIG Reasoner</implementation:name>
    <implementation:code>edu.unika.aifb.kaon.server.components.DIGComponent</implementation:code>
    <implementation:version>1.0</implementation:version>
</implementation:CodeDetails>

</rdf:RDF>
```

The CodeDetails belong to the SoftwareModuleImplementation and are basically a conceptualization of the JMX MLET tags. Every MBean is attributed by —code— (the class name), —object— (a filename that contains a serialization of the MBean), —archive— (a list of .jar files), —codebase—, —name— (the MBean's ID), —version— and —arglist— (parameters for the MBean's constructor). All of them became attributes of CodeDetails with —archive— pointing to an instance of the newly introduced Archive concept. The latter features a transitive association requiresArchive that helps the component loader in computing all the necessary libraries.

4.3 Automatic client-side discovery

The registry's surrogate (RemoteRegistry) basically wraps a RemoteKAONConnection (cf. section 5.1.1) to the main-memory based KAON API component deployed to the kernel and offers some convenience methods for interaction.

Typically a client is interested in components of a certain type, e.g. ones that conform to the DIG[3] interface. The getComponentIDs() method provides this functionality.

[3]Description Logics Implementation group. Currently, FaCT and Racer conform to this interface.

For convenience, only the name of the desired concept (as defined in the Semantic Web Profiles sub-ontology) has to be given as argument. After retrieving the component IDs in a Collection, the client might choose which one to use by having a closer look at their profiles. A component ID is fed into getProfileInfo() for this purpose. The client might display them to the user or choose automatically according to some criteria. A surrogate can now be constructed by providing it the component ID. Below we give an example for discovering DIG Reasoners[4].

```
//Create Surrogate RemoteRegistry
Map parameters=new HashMap(); parameters.put(SERVER_URI,
 "mbean://RMI@localhost:1099/jmx/RMIConnector/
 System%20Component?name=Registry");
RemoteRegistry registry = new RemoteRegistry(parameters);

//Get Component IDs of all deployed DIGReasoners
Collection ids = registry.getComponentIDs("DIGReasoner");
if (ids.isEmpty()) {
    System.out.println("No DIGReasoners deployed");
} else {
    Iterator test = ids.iterator();
    while (test.hasNext()) {
        String componentid = (String)test.next();
        //something like "Proxy Component:name=Fact"
        //Listing properties for componentid for user to choose
        //get profile properties of component
        //(representationLanguage, ontology etc.)
        HashMap props = registry.getProfileInfo(componentid);
        for (Iterator temp=props.entrySet().iterator();temp.hasNext();) {
            Map.Entry me = (Map.Entry) temp.next();
            System.out.println("\t\tProperty:"+(String)me.getKey()+",
            Value:"+(String)me.getValue());
        }
    }
//choose component by graphical user interface or automatically
String choice = ...

//construct surrogate
RemoteReasoner proxy = new RemoteReasoner(server,choice);
```

4.4 Viewing the registry's contents

One can use KAON's OIModeler graphical user interface to browse the registry's current contents (see Figure 4). In the "Open OIModel" dialog choose "Other" and enter as physicalURI (the query part is the UTF-8 encoded originalphysicalURI, see section 5.1.1):

[4]cf. edu.unika.aifb.kaon.server.test.DiscoveryTest

`mbean://localhost:1099?file%3A%2F%2F<path>%2F\resources%2F\ontologies%2Fregistry.kaon`

Additionally, the connection parameters SERVER_URI and KAON_CONNECTION must be provided as discussed in section 5.1.1. Note that kaonserver-client.jar has to be in OIModeller's class path.

Every description file that is provided to the component loader's deploy method will be included by `/resources/ontologies/registry.kaon` at runtime. It will be excluded when the component is undeployed.

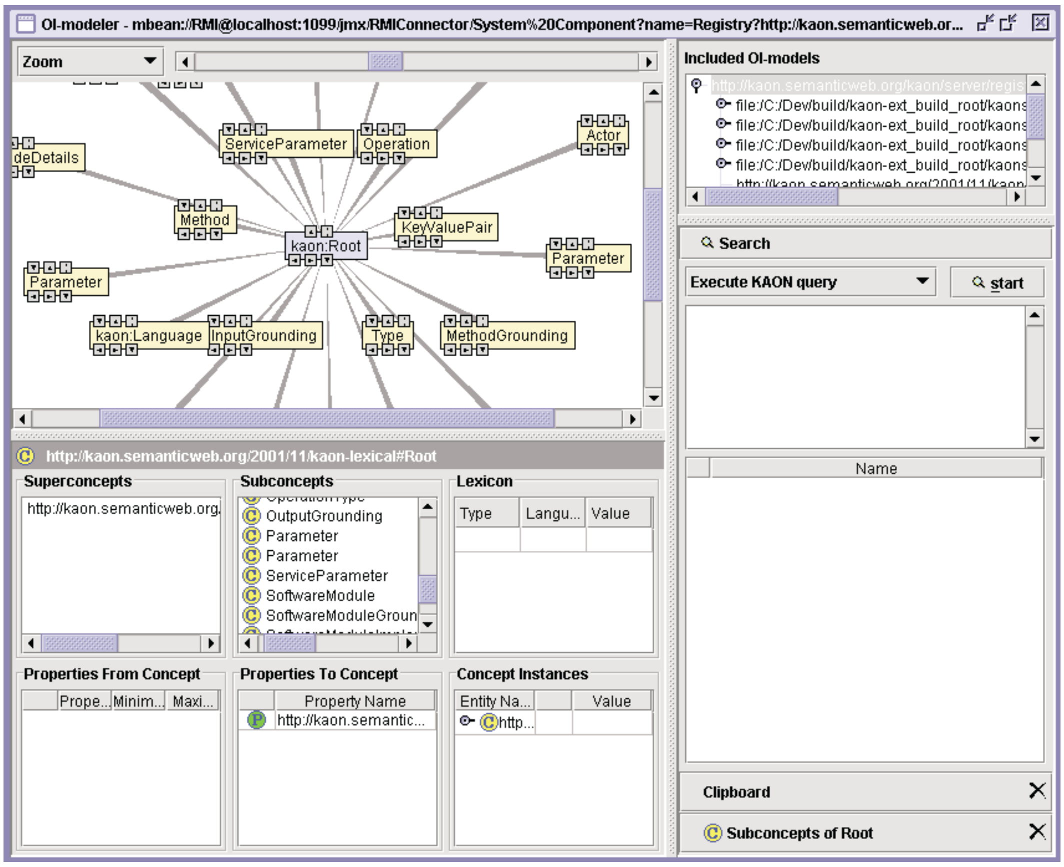

Figure 4: Browsing the registry

5 Working with client-side surrogates

As introduced in [12], surrogates are client-side objects that reveal the API of particular components residing within the KAON SERVER and relay communication to them (similar to stubs in CORBA). The idea of a surrogate is to relieve the developer of tunneling all method-calls to an MBean via MBeanServer.invoke(). Instead, the developer should be put in a position equal to working with the software module directly. All surrogates are located in edu.unika.aifb.kaon.server.client, they are labelled Remote<original class name or component name>.

Surrogates work with connector components which have to be provided to the constructor in the form of java.util.properties. The following table gives an overview of possible connectors and their properties (all of them defined in edu.unika.aifb.kaon.-server.Constants). At the moment, there are three connectors: a local connector, a SOAP and a RMI connector. Besides the connection parameters, every surrogate has to be provided the name of an MBean. This name is typically discovered by querying the registry (cf. section 4).

CONNECTION	Parameters	Possible Values
LOCAL	No other parameters required	-
SOAP	SOAP_HOST	Hostname of SOAP connector, default:localhost
	SOAP_PORT	Portnumber of SOAP connector, i.e. KAON SERVER's host, default:8085
	SOAP_NAME	Name of the WSDL-description, e.g. soapconnector
	SOAP_PATH	Path of the WSDL-description, e.g. jmx
RMI	RMI_HOST	Hostname of RMI connector, i.e. KAON SERVER's host, default:localhost
	RMI_NAME	Name of the RMI connector, default:jmx/RMIConnector
	RMI_PORT	Portnumber of RMI host, default: 1099

We already gave an example of constructing a surrogate using the RMI connector in section 3. Additionally, we will demonstrate how to construct the properties required for the surrogate “RemoteClient” which relays communication to the Ontobroker [3] proxy component [13] by using a SOAP connector. Its constructor takes the property list as well as the name of the MBean as argument. After instantiation the client is able to work with Ontobroker by using the surrogate object.

```
Properties props = new Properties();
props.put(CONNECTION,SOAP);
props.put(TYPE, MGMT);
props.put(SOAP_HOST, "localhost");
props.put(SOAP_PORT, "8085");
props.put(SOAP_PATH, "jmx");
props.put(SOAP_NAME, "soapconnector");
RemoteClient ontobroker=
 new RemoteClient(props,"Proxy Component:name=Ontobroker");
```

Note, that there also is a surrogate for the MBeanServer itself to interact directly with the kernel. Typically, all the other surrogates, like RemoteClient above, use RemoteMBeanServer to translate their calls to the MBeanServer's invoke method.

5.1 KAON Surrogates

KAON API and KAON RDF API implementations have been made deployable and surrogates have been developed, too. The surrogates themselves implement the KAON API and KAON RDF API respectively and relay communications to KAON API and KAON RDF API implementations deployed to the KAON SERVER. Thus, there is an additional indirection with KAON SERVER in between. Relevant prefix for the API implementations' physicalURIs is "mbean://"

5.1.1 RemoteKAONConnection

The RemoteKAONConnection implements a KAONConnection for KAON SERVER. On the server side, the actual KAONConnection is deployed as an MBean. PhysicalURIs for this kind of connection take the following syntax:

mbean://HOST?originalphysicalURI

The scheme *mbean* is important for RemoteKAONConnection to determine whether it is capable of handling this physical URI. Host is only used as a dummy and does not carry information. The parameter for the remote connection are encoded in the $SERVER_URI$. The $originalphysicalURI$ is the $physicalURI$ of the KAONConnection wrapped by the MBean (file, http, jboss, direct etc.). Note, that the $originalphysicalURI$ has to be UTF-8 encoded. This is achieved by the following code

```
new URI("mbean://<host>[:<port>]" +
 URLEncoder.encode(<original-physical-URI> ,"UTF-8"));
```

RemoteKAONConnection needs a SERVER_URI as additional parameter that holds the information needed to connect to the MBeanServer. It takes the following syntax:

mbean://CONNECTION@HOST:PORT/PATH/JMX-DOMAIN?JMX-ATTR-LIST

where

CONNECTION is either LOCAL, RMI or SOAP

HOST is either localhost,RMI_HOST or SOAP_HOST

PORT is either 0,RMI_PORT or SOAP_PORT

PATH is either RMI_NAME or SOAP_NAME

JMX-DOMAIN is the domain of the MBean

JMX-ATTR-LIST are the attributes of the MBean's JMX name

For example, a typical $SERVER_URI$ with SOAP access would look like the following mbean://SOAP@localhost:8085/jmx/soapconnector/example?Functional%20Component-=KAONComponent1. In addition to the $SERVER\_URI$, the $KAON\_CONNECTION$ parameter must be set to $edu.unika.aifb.kaon.server.client.RemoteKAONConnection$.

5.1.2 RemoteRDFFactory

An implementation of the RDF factory for remote models residing in the KAON SERVER. Relevant prefix for this kind of physicalURI is "mbean". All information required to address the remote MBeanServer is encoded in the physicalURI:

mbean://CONNECTION@HOST:PORT/PATH/JMX-DOMAIN?JMX-ATTR-LIST

where

CONNECTION is either LOCAL, RMI or SOAP

HOST is either localhost,RMI_HOST or SOAP_HOST

PORT is either 0,RMI_PORT or SOAP_PORT

PATH is either RMI_NAME or SOAP_NAME

JMX-DOMAIN is the domain of the MBean

JMX-ATTR-LIST are the attributes of the MBean's JMX name

For example, a typical physical URI with SOAP access would look like the following mbean://SOAP@localhost:8085/jmx/soapconnector/example?Functional%20Component-=RDFComponent1

Within RemoteRDFFactory's methods, this URI would be parsed into a valid JMX-name "Functional Component:name=RDFComponent1" to address the MBean within the server. All the other information is required to instantiate the SOAPConnector. Before using this surrogate, the factory should be registered with the RDFManager:

```
RDFManager.registerFactory( "edu.unika.aifb.kaon.
server.client.RemoteRDFFactory");
```

5.2 Other surrogates

Within the current KAON SERVER distribution there are several other surrogates that mostly coincide with the adaptation of existing software modules. Most of them are described in other WonderWeb deliverables:

RemoteRDFGeneric This surrogate features the most basic methods that are common to RDF stores. It can be used in conjunction with the KAON RDF API Component and the Sesame Component.

RemoteSesame Surrogate for the Sesame RDF store [14]. In version 1.0 this surrogate has been updated with regards to security issues. It is now expected that the provider deploys his SesameComponent to the KAON SERVER with user, password and server URL already set. Therefore, the surrogate and thus the client do not have to deal with the credentials except an Authentication Interceptors is explicitly deployed. It also the only surrogate that is able to work with the prototypical Authentication interceptor so far.

RemoteReasoner Surrogate for DIG Reasoners [1].

RemoteClient Surrogate for Ontobroker [13].

RemoteRegistry Surrogate for the registry (cf. 4).

RemoteComponentLoader Surrogate for the component loader (cf. 3)

RemoteAssociationManagement Surrogate for the Association Management (cf. 3.4)

RemoteMBeanServer Surrogate for the kernel, i.e. the MBeanServer.

6 The Semantic Web Service Connector

The Semantic Web Service Connector (SWSConnector) provides a flexible mechanism to access the methods of any deployed MBean by the SOAP protocol along corresponding WSDL and OWL-S descriptions.

The connector uses the GLUE[5] Web Services engine to handle SOAP requests/ responses and the automatic generation of the corresponding WSDL descriptions. While generating the WSDL with a given interface requires online two lines of code, routing incoming SOAP requests to the actual MBean necessitates an indirection. This is due to the fact that every request has to be routed through the MBeanServer. Hence, we had to come up with a reflection proxy that reveals the MBean's interface and does the corresponding routing. OWL-S is finally generated by using the OWL-S API[6] and additionally information stemming from the registry.

[5] `http://www.webmethods.com`
[6] `http://www.mindswap.org/2004/owl-s/api/`

6.1 Architecture

Figure 5 shows the steps that lead to the publishing of a WSDL and OWL-S description. The SWSConnector and its GLUE SOAP engine are able to publish several MBeans at once. Also, the connector applies the Association Management, i.e. when an MBean is published as Semantic Web Service it must not be undeployed.

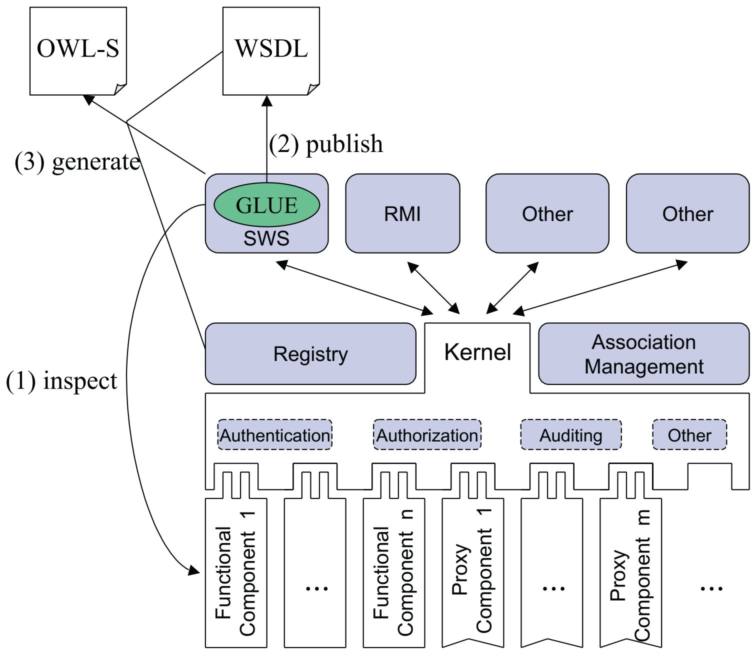

Figure 5: Process of publishing.

The client or user first provides an MBean ID to the publish method together with an URL path identifier, e.g. “Proxy Component:name=Sesame” and “sws”. In step (1) the connector inspects the specified MBean and provides an invocation handler to its SOAP engine. The invocation handler translates every incoming request into respective calls to the MBeanServer which are routed to the specified MBean. The WSDL description is automatically derived by using Java reflection and published (2). Next, the OWL-S API is used to generate a simple OWL-S description derived from the WSDL description. That comprises the grounding in particular, but also basic profile and process information (3). Additional information is taken from the MBean’s description in the registry. Although that could enrich the OWL-S description significantly, we encountered several difficulties that are due to ontology mapping (see next subsection).

The whole process results in `http://localhost:8090/sws/sesame` as the Web Service’s URI with `http://localhost:8090/sws/sesame.wsdl` and `http://localhost:8090/sws/sesame.owls` be ing the URIs of the WSDL and OWL-S description, respectively.

6.2 Ontology Mapping

First of all, every method of an MBean becomes a separate service with profile, process model (with one AtomicProcess that represents the method) and grounding. Regarding the mapping from the registry's information to the OWL-S description, the SWS Connector behaves as follows:

- "serviceName" in registry is copied
- "textDescription" is copied, if not existing, default description is generated
- "Actor" instance with corresponding attributes ("webURL", "title", "fax", "email", "phone", "physicalAddress") is copied and instantiated only once but referenced by each service.
- "serviceParameters" (i.e. "representationLanguage", "model", "serverURL" etc.) are copied. A new instance is created and referenced by all services with "sParameter" pointing the original parameter's value and "serviceParameterName" getting the name of the property as value. Thus, the subproperty-relationship is lost.

Further information could be leveraged. However, it would require cumbersome mapping of ontological information at run-time. As long as there is no ontology mapping language (such as XSLT for XML) the mapping would have to be hard-coded. In our opinion, this is a clear use-case for applying a foundation ontology. Both ontologies would have to be aligned to a common "roof" to facilitate the reuse of component and service descriptions. Below we list information that is lost due to that problem:

- Semantic Web Profile: In KAON SERVER's ontology we subclass Profile to "RDFStore", "OntologyStore", "Reasoner" etc. with specified ServiceParameters. Such information could be used in the OWL-S profile also.
- Semantic Web API Description: components' APIs along their methods and parameters are described in the registry. We could reuse such information in the process model to further specify processes, as well as their inputs and outputs.

7 OilEd Demonstrator

The OilEd demonstration shows how KAON SERVER can be used to facilitate the development of Semantic Web applications. OilEd is both a DAML+OIL and OWL ontology editor. Instead of loading a saving an ontology from a file, this version connects to the KAON SERVER, queries the registry for available RDF stores, lets the user choose one and loads/saves the contents from/into that store. Currently KAON RDF stores and Sesame are supported, others are possible. In a similar fashion, OilEd makes use of deployed reasoners rather than instantiating a local reasoner for every classification. Thus, we increase flexibility and reuse towards a programming in the large (aka megaprogramming) which is typically required for Semantic Web applications. Any store or reasoner can be deployed or undeployed at runtime.

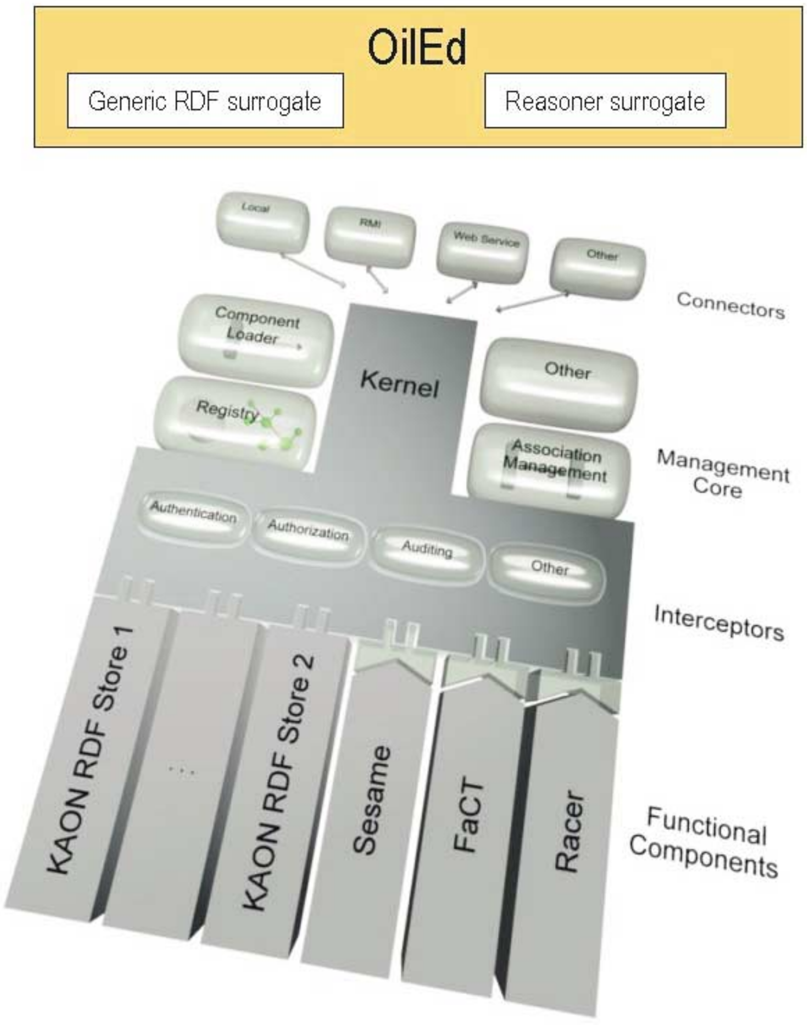

Figure 6: OilEd as a client acting upon KAON SERVER.

Like depicted in Figure 6, OilEd works with both a generic RDF surrogate and a generic reasoner surrogate. The first is able to relay communication to KAON RDF stores and Sesame. The latter is able to talk to FaCT and Racer. At run time, OilEd queries the registry for deployed RDF store and reasoners, respectively. Subsumption reasoning is automatically applied and the result (in the form of component IDs) returned. The user may choose one of them and may view their according metadata.

Note that the loading and saving only works with OWL-DL ontologies, i.e. RDF serializations thereof. The ontologies must validate according to the OWL Validator as it is used within OilEd itself. When loading and saving across different stores one might encounter problems which are due to the different XML/RDF parsers used in OilEd and in the stores. On the one hand, this clearly hinders interoperability. On the other hand, such a situation could be overcome by writing a flexible interceptor deployed on top of every RDF store. It would monitor source and destination of XML/RDF streams and could reparse and reserialize to circumvent such problems. The following configuration has been tested:

1. Start KAON SERVER

2. Deploy stores and reasoners
   This is done by moving the description files (shipped with the OilEd distribution in /descriptions) into the hotdeploy directory of KAON SERVER. Others are possible, of course.

3. Assumptions

   - KAONRDFComponent.kaon
     The model and value attributes should be the file-URL for test.owl which is in the ontologies directory of OilEd. Other OWL-DL ontologies that can be loaded into KAON RDF stores and OilEd are possible.

   - SesameComponent.kaon
     Assumes that Sesame is running @ localhost:8080/sesame, and that it accesses mem-rdf-db. Currently, the OilEd code does not allow to chose repositories. Chosen will be the one that is specified here with hardwired testuser:opensesame.

   - FaCT.kaon
     The description file assumes that FaCT runs on http://localhost:8080/dig. Tomcat should be started before the proxy component is deployed. The URL can be changed, of course.

   - Racer.kaon
     Like FaCT but on port 8081.

4. Start OilEd 3.5.7d and load from repository (loading from KAONRDFStore has been tested successfully)

5. Edit your ontology
   Classify the ontology This is done by clicking on ”DIG”. OilEd queries the KAON SERVER for available DIG reasoners and lets the user chose.

6. Save your ontology (Sesame has been tested successfully, saving back into KAON RDF stores yields a parser error)

8 Conclusion

This deliverable presented the KAON SERVER Demonstrator from a user’s point of view. We discuss analysis, requirements, design and implementation in [12] as well as the contribution to the Middleware community in [10]. The largest part of the design has been implemented along several adaptations of existing software [1, 13, 14].

In the more distant future several research questions and tasks arise. First of all, we plan to align KAON SERVER’s ontology to the DOLCE foundational ontology [6]. The current ontology is based on OWL-S [9] and suffers conceptual ambiguity, lacks concise axiomatization, is designed too loosely and has a narrow scope. We might be able to overcome those shortcomings by such an alignment. Also, the mapping between component and service descriptions will be facilitated by that as discussed in section 6. First steps in this direction have already been undertaken and the results are promising [4, 7].

Second, we plan to include a trust management system. As soon as the whole Semantic Web layer cake has been established, trust will probably be an important aspect of Semantic Web applications. Thus, the KAON SERVER should support a developer in reoccurring tasks in these aspects.

Finally, we consider research regarding the flexible handling of different types of Semantic Web software entities, such as web services, peers or agents. On the one hand this comprises easy integration of existing ones into the server, thus lifting the responsibility of handling several different protocols off a developer. On the other hand, deployed components ought to be offered also by web service, peer and agent protocols. This task includes translating semantic descriptions accordingly.

Acknowledgements We would like to thank Marta Sabou, Vrije Universiteit Amsterdam, and Debbie Richards, Macquarie University Sydney, for the fruitful cooperation. Their work helped a great deal in designing the management ontology and component descriptions. Also Sean Bechhofer from the University of Manchester who maintains the OilEd ontology editor and invested great efforts for the KAON SERVER adaptation.

References

[1] Sean Bechhofer, *Fact and oiled clients*, WonderWeb Deliverable D10, Aug 2003, http://wonderweb.semanticweb.org.

[2] Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew V. McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia P. Sycara, *DAML-S: Web service description for the Semantic Web*, The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings (Ian Horrocks and James A. Hendler, eds.), Lecture Notes in Computer Science, vol. 2342, Springer, 2002, pp. 348–363.

[3] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer, *Ontobroker: Ontology based access to distributed and semi-structured information*, Database Semantics - Semantic Issues in Multimedia Systems, IFIP Conference Proceedings, vol. 138, Kluwer, 1998, pp. 351–369.

[4] Aldo Gangemi, Peter Mika, Marta Sabou, and Daniel Oberle, *An ontology of services and service descriptions*, Tech. report, Laboratory for Applied Ontology (ISTC-CNR), Viale Marx, 15, 00137 Roma, 2003.

[5] Alexander Maedche, Boris Motik, and Ljiljana Stojanovic, *Managing multiple and distributed ontologies in the semantic web*, VLDB Journal **12** (2003), no. 4, 286–302.

[6] Claudio Masolo, Stefano Borgo, Aldo Gangemi, Nicola Guarino, Alessandro Oltramari, and Luc Schneider, *The wonderweb library of foundational ontologies*, WonderWeb Deliverable D15, Aug 2002, http://wonderweb.semanticweb.org.

[7] Peter Mika, Daniel Oberle, Aldo Gangemi, and Marta Sabou, *Foundations for service ontologies: Aligning owl-s to dolce*, The Thirteenth International World Wide Web Conference Proceedings, ACM, MAY 2004, pp. 563–572.

[8] Boris Motik, *KAON — the Karlsruhe Ontology and Semantic Web framework — Developer's guide for KAON 1.2.5*, http://kaon.semanticweb.org, Feb 2003.

[9] D. Oberle, M. Sabou, D. Richards, and R. Volz, *An ontology for semantic middleware: extending DAML-S beyond web-services*, On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops, Lecture Notes in Computer Science, vol. 2889, Springer, 2003, pp. 28–29.

[10] Daniel Oberle, Andreas Eberhart, Steffen Staab, and Raphael Volz, *Developing and managing software components in an ontology-based application server*, Middleware 2004, ACM/IFIP/USENIX 5th International Middleware Conference, Toronto, Ontario, Canada, LNCS, Springer, 2004.

[11] Daniel Oberle, Marta Sabou, and Debbie Richards, *An ontology for semantic middleware: extending DAML-S beyond web-services*, Tech. Report 426, University of Karlsruhe, Institute AIFB, 76128 Karlsruhe, Germany, 2003.

[12] Daniel Oberle, Steffen Staab, Rudi Studer, and Raphael Volz, *Supporting application development in the semantic web*, ACM Transactions on Internet Technology (TOIT) **4** (2004), no. 4.

[13] Raphael Volz, Daniel Oberle, Steffen Staab, and Rudi Studer, *Ontobroker and ontoedit adaptation*, WonderWeb Deliverable D9, Jul 2003, http://wonderweb.semanticweb.org.

[14] ______, *Triple client*, WonderWeb Deliverable D8, Jun 2003, http://wonderweb.semanticweb.org.