# An agent-based system for Grid services provision and selection

G U S T A F   N I M A R

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LECS-2004-64

# An agent-based system for Grid services provision and selection

GUSTAF NIMAR

Examiner
Assoc. Prof. Vladimir Vlassov
(IMIT/KTH)

Master of Science Thesis
Stockholm, Sweden 2004

IMIT/LECS-2004-64

## Abstract

During these last years we have seen a dramatically increase of services and products accessible over the Internet. In addition to this, the number of service requestors has increased along with the general public's interest in using the Internet as a marketplace. Considering these two facts it's becoming impossible to continue this progress unless we find ways to bring these two parts together.

*An agent-based system for Grid services provision and selection* is an agent-based architecture for provision, selection and (in the future) composition of Grid services, with respect to a user's requirement. The idea is to organize the agent architecture as a marketplace where service providers and requestors can meet and negotiate about services. A user specifies its requirement to an agent, who starts to negotiate with the agents provisioning services.

The main delivery of this project is a prototype implementing the architecture in Java. We will use Grid services based on the Open Grid Services Architecture (OGSA) and the agent architecture will be implemented using an existing agent software platform. The delivered prototype of system architecture was realized using Globus Toolkit 3 [23], i.e. a well-known implementation of OGSA, as well as the JADE agent platform [38]. This prototype was used in the evaluation tests of the proposed agent architecture.

## Sammanfattning

Under de senaste åren har de blivit en dramatisk ökning av antalet produkter och tjänster som görs tillgängliga över Internet. Dessutom bör det tilläggas att antalet användare av tjänsterna har ökat i samband med allmänhetens växande intresse av att använda Internet som en marknadsplats. Om man reflekterar över dessa två fakta är de lätt att se att detta är en ohållbar utveckling om vi inte hittar nya lösningar för att föra de två parterna samman.

*An agent-based system for Grid services provision and selection,* är en agent-baserad arkitektur för att erbjuda, välja och (i framtiden även) komponera Grid service-baserade tjänster, utifrån en användares krav. Den huvudsakliga idén är att organisera arkitekturen som en marknadsplats där användare som erbjuder tjänster kan förhandla om dessa med dem som är i behov av dem. En användare som söker efter en tjänst beskriver den för sin agent som därefter börjar förhandla med agenter som erbjuder tjänster.

Den huvudsakliga utdelningen av det här projektet kommer att vara en prototyp av arkitekturen i Java. Vi kommer att använda oss av Grid Services, vilket är tjänster baserade på Open Grid Services Architecture (OGSA). Vidare så kommer vår arkitektur att förverkligas på en existerande mjukvaru-platform för agenter. Den levererade prototypen använde sig av Globus Toolkit 3 [23], som är en välkänd implementering av OGSA, likväl som agent-plattformen JADE [38]. Prototypen användes vid utvärderingen av den föreslagna agent-arkitekturen.

# TABLE OF CONTENTS

**1 INTRODUCTION ....... 1**

1.1 MOTIVATION ....... 1
1.2 PROJECT GOALS ....... 1
1.3 RELATED WORK ....... 2
1.4 STRUCTURE OF THESIS ....... 2

**2 RELATED TECHNOLOGIES ....... 3**

2.1 GRIDS ....... 3
2.2 WEB SERVICES ....... 3
*2.2.1 WSDL ....... 4*
*2.2.2 SOAP ....... 8*
*2.2.3 SOAP Message ....... 8*
*2.2.4 SOAP Transports ....... 9*
*2.2.5 UDDI ....... 9*
2.3 GRID SERVICES ....... 10
*2.3.1 Stateful Web services ....... 10*
*2.3.2 WSDL Extensions and Conventions ....... 13*
*2.3.3 Globus Toolkit 3 ....... 14*
2.4 AGENTS ....... 15
*2.4.1 Intelligent Agents ....... 16*
*2.4.2 Multi-Agent Systems ....... 16*
*2.4.3 Market structure ....... 16*
*2.4.4 Agent Communication Languages ....... 16*
*2.4.5 Agent content language ....... 17*
*2.4.6 Multi-Agents Toolkits ....... 19*
*2.4.7 Java ....... 20*
2.5 ONTOLOGY ....... 21
*2.5.1 OWL ....... 21*
*2.5.2 OWL-S ....... 21*
*2.5.3 Tools ....... 22*
2.6 COMPOSITION ....... 23
*2.6.1 Linear Logic ....... 23*

**3 ANALYSIS AND DESIGN ....... 25**

3.1 TERMINOLOGY ....... 25
3.2 SCENARIOS OF SYSTEM USAGE ....... 25
*3.2.1 Service requestor ....... 25*
*3.2.2 Service provider ....... 26*
*3.2.3 Detailed Scenario ....... 26*
3.3 USE CASES ....... 26
*3.3.1 Provision of service ....... 27*
*3.3.2 Selection of service ....... 28*
3.4 INTERACTION ....... 28
*3.4.1 Provision of service ....... 29*
*3.4.2 Selection of service ....... 29*
3.5 AGENT-TO-AGENT INTERACTION ....... 30
*3.5.1 Service Provision Agent Protocols ....... 30*
*3.5.2 Service Selection Agent Protocols ....... 31*
3.6 MESSAGE CONTENT ....... 31
3.7 MATCHING ALGORITHM ....... 33
3.8 INFORMATION FLOW ....... 34
3.9 COMPOSITION ....... 37
*3.9.1 Overview of the proposed design ....... 37*
*3.9.2 LL representation of services and proof intuition ....... 38*

**4 IMPLEMENTATION ....... 39**

4.1 DEVELOPMENT PLATFORM ....... 39

4.2 AGENTS ....39
*4.2.1 ServiceProvisionAgent ....39*
*4.2.2 ServiceSelectionAgent ....40*
4.3 CONTENT ....42
*4.3.1 lang ....42*
*4.3.2 owls ....42*
*4.3.3 wsdl ....43*
4.4 GRID ....43
4.5 MATCHER ....43
4.6 STORAGE ....43
4.7 GRID SERVICE EXTENSION ....44
4.8 USER MANUAL ....45

**5 VALIDATION ....46**

5.1 GENERATING A GRID SERVICE ONTOLOGY DESCRIPTION ....46
5.2 ASSIGNMENT OF A REGISTRY ....47
5.3 REALIZING A DETAILED SCENARIO OF SYSTEM USAGE ....48

**6 EVALUATION ....51**

6.1 TEST-BED PLATFORM ....51
6.2 PERFORMANCE BASED ON TIME MEASUREMENTS ....51
*6.2.1 Provision of service ....52*
*6.2.2 Selection of service ....53*
*6.2.3 Matching algorithm ....55*
6.3 MEMORY USAGE ....57
*6.3.1 Agents ....57*
*6.3.2 Service description ....58*
6.4 EVALUATION SUMMARY ....58

**7 CONCLUSIONS ....59**

**8 FUTURE WORK ....61**

**9 REFERENCES ....62**

**A. ABBREVIATIONS ....65**

**B. PROTOTYPE MANUAL ....66**

**C. GENERATED SERVICE DESCRIPTIONS IN OWL-S ....67**

**D. OWL-S VALIDATION RESULTS ....75**

**E. EXECUTION PRINTOUTS OF VALIDATION OF DETAILED SCENARIO ....76**

**F. PROTOTYPE DOCUMENTATION ....79**

# TABLE OF FIGURES

Figure 2.1 Invocation of a Web Service....4
Figure 2.2 An example types element....5
Figure 2.3 An example of WSDL Message elements....6
Figure 2.4 An example WSDL portType element. ....6
Figure 2.5 An example WSDL binding element using SOAP on top of HTTP. ....7
Figure 2.6 An example WSDL service element including a port element....7
Figure 2.7 An example SOAP message. ....8
Figure 2.8 Different ways to accessing a Grid service....11
Figure 2.9 An example description of a Service Data Element....13
Figure 2.10 An example gwsdl portType element. ....13
Figure 2.11 Globus Toolkit 3 Core architecture [23]....15
Figure 2.12 A FIPA ACL inform message....17
Figure 2.13 FIPA-request protocol [26]....17
Figure 2.14 A simple RDF document....18
Figure 2.15 An example FIPA RDF Action [31]....19
Figure 2.16 Structure of an Agora agent [32]....20
Figure 3.1 Use cases of the system. Both selection and provision of service are divided into several minor Use cases. ....27
Figure 3.2 Collaboration diagram describing the interaction between different parts of the system used to provision services....29
Figure 3.3 Collaboration diagram describing the interaction between different parts of the system used to select services ....30
Figure 3.4 The Protocols used by the Service Provision Agent. The register message is a platform specific method and is not to be mistaken for FIPA ACL message. ....31
Figure 3.5 The Protocols used by the Service Selection Agent. The search and result messages are platform specific methods and are not to be mistaken for FIPA ACL messages. ....31
Figure 3.6 A FIPA RDF Action requesting a search at SPA1....32
Figure 3.7 A response containing the result of the FIPA RDF Action message. ....32
Figure 3.8 The service matching algorithm. ....33
Figure 3.9 The information flow in the system....36
Figure 3.10 The design of the service composer....38
Figure 4.1 Method for assigning a new service. ....40
Figure 4.2 Method for search for a service....42
Figure 4.3 The Service Data Type OwlsDataType....44
Figure 4.4 Importing the OWL-S Service Data type....44
Figure 4.5 Instatiating a OWL-S Service Data element....45
Figure 4.6 The initialization method, optional for GT3 Grid services....45
Figure 5.1 Validation of Grid service description generation....47
Figure 5.2 Validation of the interaction with the registry. ....48
Figure 5.3 Prinout of a SPA extracing service from a VORegistry....48
Figure 5.4 The validation of the detailed scenario of system usage....49
Figure 6.1 The time consumed by different parts of the system when provisioning a service. ....53
Figure 6.2 The time consumed by different part of the system when selecting a service. ....55
Figure 6.3 The scalability of the matching algorithm. ....56
Figure 6.4 The time consumed by the different matching passes....57

# TABLE OF TABLES

TABLE 2.1 THE PREDEFINED PORTTYPES IN OGSI. ..........14
TABLE 2.2 CONNECTIVES AND OPERATORS IN LINEAR LOGIC. ..........24
TABLE 3.1 EXPLANATION OF THE MOST IMPORTANT TERMS USED IN THE DOCUMENT. ..........25
TABLE 3.2 THE FIPA COMMUNICATIVE ACTS USED IN THE PROJECT. ..........30
TABLE 4.1 BEHAVIOURS OF THE SERVICE PROVISION AGENT. ..........40
TABLE 4.2 BEHAVIOURS OF THE SERVICE SELECTION AGENT. ..........41
TABLE 5.1 THE METHODS IMPLEMENTED IN THE SAMPLE GRID SERVICE. ..........46
TABLE 5.2 THE SPAS AND THEIR SERVICES IN THE DETAILED SCENARIO OF SYSTEM USAGE. ..........49
TABLE 6.1 THE TIME CONSUMED BY DIFFERENT PARTS OF THE SYSTEM WHEN PROVISIONING A SERVICE. ......52
TABLE 6.2 THE TIME CONSUMED BY DIFFERENT PART OF THE SYSTEM WHEN SELECTING A SERVICE. ..........54
TABLE 6.3 MEMORY USAGE. ..........58

# 1 Introduction

This report is the first step in a research towards an architecture based on *Agent-Enabled Logic-Based Web Services Selection and Composition* [1] at IMIT at the Royal Institute of Technology (KTH). The report is also written with respect to a Master Thesis in Distributed and Parallel systems.

The main focus of this report is to examine provision and selection of a special type of Web services (in our case Grid services), in an agent-based system. This includes developing an architecture prototype defining and implementing the functionalities mentioned above. We will also briefly look into how the system could be extended to include composition of services. The evaluation of the architecture will be based on the prototype and will hopefully be of interest for the latter steps in the research.

We'll start out with a motivation for the area of research. Then there will be a more detailed description of the overall goals of the project, followed by some related work. Finally we'll look at background technologies including the concepts Grids, Web services, Grid services, Multi-Agent Systems as well as intelligent agent.

## 1.1 Motivation

Let's say that you are planning to take a trip matched together by several shorter distances and means of conveyance. You have heard of a traveling agency that is providing complete traveling packages, e.g. traveling from Hjo to Kista can result in taking a bus to Skövde, a train to Stockholm and finally taking the subway to Kista. We assume that all the major traveling businesses provide a Web service-based interface for querying routes and time, as well as for booking tickets.

The idea here is for the agency to customize a service for the traveler by composing the services provided by the different businesses. Perhaps you are only concerned with the traveling time, or with the overall price. These are things considered by the agency when composing your journey.

This example is only one possible usage, and there are many more. The demand for new ways to bring providers and requestors of services together is increasing rapidly with the number of actors on the market.

## 1.2 Project Goals

As mentioned earlier this Master Thesis is the first step in a bigger research towards a novel solution for active Web Services selection and composition [1]. The goal of the overall research is to develop a logic-based technique for composition and negotiation of Web services, in an agent-based architecture.

In the first step of the project, i.e. this Master thesis, we will look at selection and composition of services in Grids, i.e. Grid services. In order to be able to select a service one must provide it first. Therefore Grid service provision will also be a part of the project. The aim of the project is to design an architecture for Grid service provision, selection and finally (if time allows) composition. Using this design a system prototype is to be implemented, covering the functionality of provision and selection (not composition), based on an agent cooperative model. The prototype will be the base for our evaluation of the architecture.

## 1.3 Related work

Selection and composition of Web services has been addressed in many research systems lately. Comparing these systems, one can easily see that there is no universal solution to the problem, or at least not yet.

Perhaps the most important feature of Web services is its platform and programming languages independence. This independence is possible due to techniques like Universal Description, Discovery, and Integration (UDDI) [2], Web Services Description Language (WSDL) [3] and Simple Object Access Protocol (SOAP) [4] making it possible to discover, bind and invoke Web services across a network. These techniques are also the foundation of Grid services.

There have been several works on combining agents with Grid services. MyGrid is an e-science project with the goal to provide technique for biologists and bioinformaticians to run experiments by composing workflows [5]. MyGrid make use of several agent techniques such as Agents, Agent Communication Language and negotiation when reaching agreements. Another system of interest is MAGGIS [6], i.e. a Multi-Agent system architecture for monitoring of Grid services. DAMLJessKB [7] is a software with the intent to read, interpret and allowing for querying of DAML+OIL[8] documents. As the name reveals it uses the Java Expert System Shell (Jess) [9], i.e. is a rule engine for Java. The main benefit with the DAMLJessKB is that it allows for reasoning about supplied DAML+OIL documents. This feature has been used in the DAML-S Matcher [10], which is an agent matching DAML-S [11] documents, i.e. documents for describing Web services.

The goal of the overall project is to develop a technique for selection and composition of Web services based on logic with high expressive power (such as Linear Logic [12]). There has been some research in this area and especially interesting are technologies based on the same type of logic ([13] and [14]).

## 1.4 Structure of thesis

The reminder of this thesis is structured as follows: Chapter 2 covers the technologies related to the thesis; Chapter 3 presents the analysis and the proposed design of the system; Chapter 4 describes the implementation of the proposed design; after the implementation has been covered the thesis concentrates on validation and evaluation, of the implemented prototype, separately in Chapter 5 and 6; the conclusions drawn from the thesis is found in Chapter 7; and finally Chapter 8 includes future work.

# 2 Related technologies

The following chapter will describe technologies related to the project and it will be used as the knowledge base for the rest of the document. The first sections will cover Grids and Web services which will lead us into Grid services. Then we'll look at Agents and finally a section about ontologies. The reason, for covering ontologies in this chapter, is that it will be used to increase our expressiveness in the agent-to-agent negation. This will hopefully result in a well-formed communication and ease a future extension of the system.

## 2.1 Grids

The idea to share data and computing resources across a network is rather old – but never the less a hot topic. Grid computing is a quite new approach that has increased its popularity during the last decade. A Grid represents resources (computers, servers and data storages) connected together as a large virtual computer. The aim of Grid computing systems is to present a large set of resources, provided by heterogeneous systems, in a uniform way.

A Grid computing system (such as Open Grid Service Architecture (OGSA) [15]) is defined by an open set of standards and protocols making it possible for communication between heterogeneous systems. The Grid is also said to be transparent, i.e. it keeps the complexity hidden from the user, for whom the system appears in a coherent way.

Like peer-to-peer (P2P) Grid computing supports sharing of files. But in contrast to P2P systems Grid computing allows many-to-many sharing, and is extended not only to support files but also other resources. Perhaps Grid computing has more in common with Clusters, at least both of them support sharing of computing resources. The difference here lies in the fact that Clusters are both geographically and platform dependent.

The Grid system architecture we'll be using in this project is OGSA, which is based on Web services. Therefore it's crucial at least to get the idea of Web services to understand OGSA's Grid services.

## 2.2 Web Services

Web services are a distributed computing paradigm for creating applications based on the client/server model. What makes Web services special is the fact that it's using simple Internet-based standards, making it possible for interoperable machine-to-machine interaction in a platform- and language-independent manner over a network. This should be studied in comparison with other technologies such as CORBA [16], and Java RMI [17] that are bound to highly dependent clients and servers.

Web services themselves are just serving as a software interface describing a set of operations, which are accessible over the network using XML [18] messages. This Description is written in a machine-processable Description language (e.g. WSDL) that allows other systems to use this description to interact with the Web service. Web services also define discovery methods used to locate relevant service providers.

Web services are deployed on the network by a *service provider*, i.e. a person or organization providing the service. Then there's usually a *service broker* that helps a provider and a requestor of a service to find each other. The most common ways are to build the broker as an index or a registry over published services. The *service requestor* can use a broker to find the requested service and then it uses the description language to bind (negotiate settings before accessing the service) to the service.

Figure 2.1 describes the steps (in chronological order) taken when invoking a Web service. In the first step the service provider registers its services at the broker. After the services have been registered service requestors can search the broker for suitable services. Once a suitable service has been found the service requestor can use the information provided by the broker to receive additional information needed to invoke the service. This information is provided by a WSDL document (described in detail below). Using the WSDL document the service requestor can specify its messages in order to invoke the operation of interest.

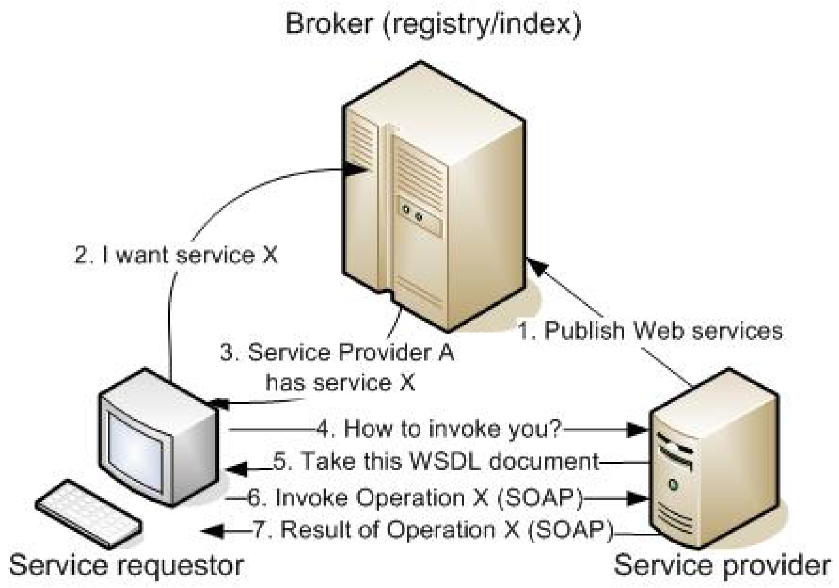

Broker (registry/index)
2. I want service X
1. Publish Web services
3. Service Provider A has service X
4. How to invoke you?
5. Take this WSDL document
6. Invoke Operation X (SOAP)
7. Result of Operation X (SOAP)
Service requestor
Service provider

**Figure 2.1 Invocation of a Web Service.**

In contrast to Grid services, Web services are stateless and thereby cannot remember values of operations carried out.

### 2.2.1 WSDL

To be able to use a service one must know how to interact with or take advantage of the service. Web Services Description Language (WSDL) [3] is an XML-based Language used for this purpose, i.e. describing how to interact with a Web service. Due to the fact that WSDL is based on XML makes it independent of programming languages as well as development environments. A WSDL document describes the different operations that can be carried out, how to invoke these operations, and the expected result. It also defines supported protocols, e.g. SOAP.

A WSDL document is structured as a logical tree (of elements) where the root is a definitions element holding six other elements for describing a service. These elements can be categorized into three different groups depending on the information they are holding, namely: Service Interface, Service Binding, and Service Implementation. These elements will be explained further below using an example Web Service called *RoutePrice service*. The service provides price lookup of a given route.

#### 2.2.1.1 Service Interface

The first group contains the Types, Message and PortTypes elements. The characteristics of this group are that it contains information, independent of platforms, protocols or programming languages, about supported operations, and the data being exchanged.

**The types element**

The types element contains different data type definitions that are used to describe the messages being exchanged. WSDL prefers if the types are described in XML Schema Definition (XSD) to keep high interoperability and platform independence. XSD specifies how to formally describe elements in an XML document. Figure 2.2 holds an example types element defining two types, i.e. *GetRoutePriceRequest* and *RoutePrice*.

```
<types>
   <schema
    targetNamespace=http://travelbusiness.com/routeprice.xsd
    xmlns="http://www.w3.org/2000/10/XMLSchema">

       <element name="GetRoutePriceRequest">
          <complexType>
              <all>
                  <element name=″route" type="string"/>
              </all> ServiceSelectionAgent
          </complexType>
       </element>

       <element name="RoutePrice">
          <complexType>
              <all>
                  <element name="price" type="float"/>
              </all>
          </complexType>
       </element>
   </schema>
</types>
```

**Figure 2.2 An example types element.**

**The message element**

The message element represents an abstract definition of the data being exchanged between providers and requestors. Every message consists of one or more logical parts, one for each parameter of a Web service operation. Each part associates with a concrete type defined in the Types element. Every operation has at least one of the two messages input and output, where the former describe input parameters and the latter the return data of an operation. In Figure 2.3 both input and output messages are defined. The messages only consist of a single logical body part.

```
    <message name="GetRoutePriceInput">
        <part name="body" element="xsd1:GetRoutePriceRequest"/>
    </message>

    <message name="GetPriceOutput">
        <part name="body" element="xsd1:RoutePrice"/>
    </message>
```

**Figure 2.3 An example of WSDL Message elements.**

**The portType element**

The portType element encloses a set of operations supported by the Web service. Each operation element includes its messages, in this example it has input and output messages. In Figure 2.4 the portType *RoutePricePortType* is defined. The portType includes a single operation called *GetRoutePrice.*

```
    <portType name="RoutePricePortType">
        <operation name="GetRoutePrice">
           <input message="tns:GetRoutePriceInput"/>
           <output message="tns:GetRoutePriceOutput"/>
        </operation>
    </portType>
```

**Figure 2.4 An example WSDL portType element.**

A WSDL portType (basically the end of a communication link) can support one of the four transmission primitives:

- **One-way.** The requestor sends a message to the provider. This will render an input element.
- **Request-response.** The requestor sends a message to the provider, who sends a correlated message back. This gives us both an input and output element specifying the format for the request and response.
- **Solicit-response.** Same as the Request-response primitive but here the provider sends the first messages and the requestor the correlated response.
- **Notification.** The provider sends a message to the requestor. Here we have an output element.

In addition to the input and output messages WSDL specifies a fault message that has the abstract format of an error message. The fault message is only available in the Request-response and Solicit-response primitives where a response message is expected.

#### 2.2.1.2 Service Bindings

The Service Binding is the second group and it only contains the Binding element. Here one can find supported protocols and the encoding of messages used.

**The binding element**

Given a protocol the binding element provides concrete details about a particular portType, i.e. protocol details for operations and the format of messages supported by the portType. Notice that there can be several bindings for a single portType (a portType can support more than one protocol). In Figure 2.5 there’s a binding element associated with the RoutePricePortType.

```
    <binding name="RoutePriceSoapBinding"
type="tns:RoutePricePortType">

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

        <operation name="GetRoutePrice">
           <soap:operation
            soapAction="http://travelbusiness.com/GetRoutePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>
```

**Figure 2.5 An example WSDL binding element using SOAP on top of HTTP.**

### 2.2.1.3 Service Implementation

The last group of elements holds implementation dependent details about how a client invokes the operations provided by the Web service. Figure 2.6 holds a service element defining a service called *RoutePriceService*. The service is associated with the binding element in Figure 2.5.

```
<service name="RoutePriceService">
     <documentation>Route price lookup service</documentation>

     <port name="RoutePricePort" binding="tns:RoutePriceBinding">

        <soap:address
        location="http://travelbusiness.com/travelquote"/>
      </port>
</service>
```

**Figure 2.6 An example WSDL service element including a port element.**

**The service element**

The service element encloses a set of related port elements and defines the name of the service. The related ports have the following relationship:

- There is no communication among the related ports.
- When several ports share a port type but binds different addresses, the ports are alternatives to the same service.
- The supported port types of some related ports can be used by a customer to determine which services to be used.

**The port element**

The port element basically specifies an endpoint, where the service requestor can bind or connect to, for accessing the service. This is done simply by assigning an address to a binding.

### 2.2.2 SOAP

A service requestor must use one of the supported protocols described in the binding element (in a WSDL document) to invoke the capabilities of a Web service. Simple Object Access Protocol (SOAP) [4] is the most frequently used protocol, almost becoming a de-facto standard when it comes to Web services. SOAP is based on XML and its specification contains

- the syntax for messages
- and a model for exchanging them,
- rules for the encoding of data in messages,
- instruction for transporting SOAP over HTTP,
- and finally it defines a model for performing Remote Procedure Calls (RPC).

We will look at some parts of the specification.

### 2.2.3 SOAP Message

Using the example in Figure 2.7, we'll go through the different elements building a SOAP message. The example is constructed as a request to an operation in the WDSL document found in Figures 2.2-2.6, and therefore the Header element is excluded.

```
<Env:Envelope
  xmlns:Env="http://www.w3.org/2003/05/soap-envelope/"
  Env:encodingStyle="http://www.w3.org/2003/05/soap-encoding/">
   <Env:Body>
       <m:GetRoutePrice xmlns:m="Some-URI">
           <symbol>BUS-402</symbol>
       </m:GetRoutePrice>
   </Enc:Body>
</Enc:Envelope>
```

**Figure 2.7 An example SOAP message.**

#### 2.2.3.1 Envelope

As the name reveals Envelope is the root element of a SOAP message. It includes the other elements, i.e. Header and Body. Envelope contains attributes for defining namespaces and properties for encoding of data in the message. The URI http://www.w3.org/2003/05/soap-envelope is the namespace of example in Figure 2.7. Depending on the namespace one can tell the SOAP version used. If the version isn't recognized a fault message is returned.

#### 2.2.3.2 Header

In addition to the data carried in a message there can be other vital parts as well, e.g. a message can be part of a series of messages in a business transaction. Because it's not feasible to define every possible extension to SOAP, the Header element was introduced. The purpose is to allow users to define extensions without modifying the payload or the overall structure of the message. The Header element is optional in a SOAP message (as we see in Figure 2.7), but in order of presence it must be defined as the first child element of an Envelope.

#### 2.2.3.3 Body

The Body element is the payload of an SOAP message and it holds the application-specific data, e.g. a query for the price of a route.

### 2.2.4 SOAP Transports

We need a way to send, or to transport, our SOAP messages from the sender to the receiver. There are no restrictions in the specification regarding the means of transport. Although most developers use the well-tested Hypertext Transfer Protocol (HTTP) [19] one might go for Carriers Pigeons as well.

#### 2.2.4.1 SOAP on top of HTTP

Hypertext Transfer Protocol (HTTP), the standard protocol of the web, is a good choice when it comes to carrying SOAP messages; mostly because of its wide acceptance. It's so well suited that rules for using the protocol is included in the SOAP specification.

In the specification HTTP Post is defined as the standard method for sending SOAP messages, and additionally the HTTP response for responding to one. The specified URI found in the HTTP header is the receiver of the message, e.g. a Web service.

### 2.2.5 UDDI

In similarity to surfing the web it's not always the case that address of the requested Web service is known in advanced, or one might not know any suited services. Having this problem while surfing the web most users would probably turn to an index service searching for the requested page. Universal Discovery Description and Integration (UDDI) [2] can be seen as an index service for Web services, where users can publish and discover Web services.

In a UDDI registry a business can register itself and its services. Each business will be represented by an XML document. A description of a business is divided into three categories: "white pages" holds information about the business such as the address and fax; "yellow pages" includes information categorizing services based on taxonomy (i.e. classification of services based on a few characteristics); and finally "green pages" holds the technical information about the services provided by the business.

#### 2.2.5.1 UDDI architecture

An UDDI registry consists of one or several UDDI nodes that together manage the data stored in the registry. The data in the registry is replicated among the UDDI nodes. The data is called simply UDDI data and is divided into four core types:

These are *businessEntity*, which describes a business or an organization providing Web services; *businessService* that describes a set of related Web services all provided by the same businessEntity; *bindingTemplate* holds the necessary information for invoking a service; and finally the *tModel* that provides a technical model consisting of reusable concepts such as transport, protocol and namespace.

The core data structures are assigned a unique key when the data is published. The key is used as an identifier in the system.

## 2.3 Grid services

As stated before, Grid services are based on Web services but they are conformed under a set of conventions defined by the Open Grid Services Infrastructure (OGSI) specification [20]. In similarity to Web services you can say that Grid services are also a WSDL-defined service. But Grid services have an extended syntax when it comes to WSDL and introduces new concepts such as service data, stateful instances, references, and notifications of services. Considering some of these concepts you can claim that Grid services have characteristics similar to distributed object-based systems [20]. But there are some characteristics that do differ, such as inheritance, services instance mobility, development approach, and hosting technology.

In order to clarify the differences between Web- and Grid services we will look at some of the major characteristics that are introduced by OGSI Version 1.0:

- Enabled stateful Web services
- Extended Web service interfaces to include Service Data
- Asynchronous notifications of state changes
- Service groups
- Extended the portType
- Lifecycle management (creation and destruction of Grid services)
- GSH and GSR (references to instances of services).

Some of these concepts important to this project will be covered in the text.

### 2.3.1 Stateful Web services

The first concept in the list is probably the most important one, i.e. the introduction of stateful Web services. As mentioned before regular Web services are stateless, i.e. values won’t be preserved from one invocation to another.

When interacting with a Web service it’s done directly towards the service. The service makes no difference regarding the requestor of the service, all are treated as equals. To interact with a Grid service one must use an instance of the service and each client using the same instance is treated equally (as seen in Figure 2.8). An instance of a service can be compared with an instance in Object Oriented programming. Each instance has its own state as well as a unique name. Furthermore, an instance is associated with one or more Grid Service Handles (GSHs), and one or more Grid Service References (GSRs), more about this later. The instance is named by its GSHs, i.e. in the form of URIs.

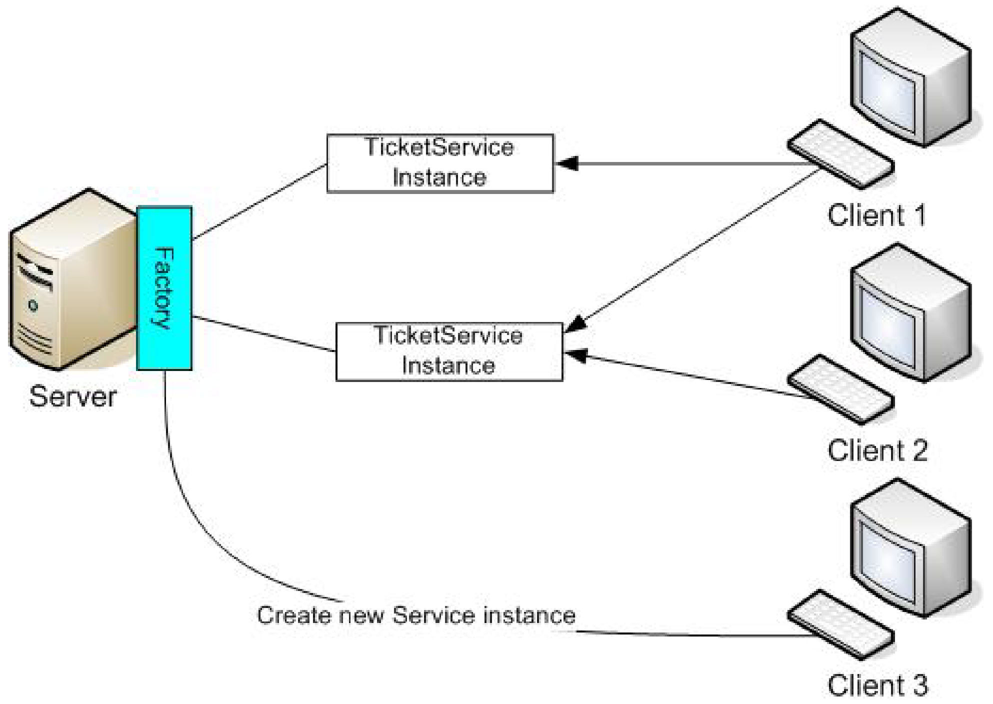


**Figure 2.8 Different ways to accessing a Grid service.**

### 2.3.1.1 Grid Service Handles and References

When a requestor needs the Grid service instance it uses the service's handle and reference. The Grid Service Handle contains a permanent pointer to the service instance (a GSH must indisputably for all time point the same Grid service instance) and therefore it doesn't hold any detailed information about how to access the service instance. A GSH is translated, using a HandleResolver, into a Grid Service Reference (GSR). A GSR provides the information needed to access the service (a GSR can have the format of a WSDL document or a CORBA IOR [16]). The GSR is considered valid as long as the associated Grid service instance exists, but notice that the GSR may become invalid even if the instance still exists (due to time constraints). In such cases the requestor should use the GSH to resolve a new GSR.

As mentioned above a HandleResolver is a Grid service that resolves a GSH into a GSR. When registering a service it's required that the service is registered with at least one HandleResolver, called its home HandleResolver. This home HandleResolver is found in the GSH. One problem is how to obtain the GSR of the HandleResolver. The solution is to make all the HandleResolvers support a bootstrapping operation and a common protocol (HTTP or HTTPS).

### 2.3.1.2 Creation and destruction of instances

To create a service instance a requestor needs to invoke the operation createService. createService is located on a Grid service with a portType that extends the Factory portType, or another portType defining methods for creating instances of Grid services.

Now when we have seen how to create an instance one might want to know how they are destroyed. Actually there is an instance destruction operation defined in the GridService portType (that must be extended by every Grid service). Another approach is to let a client create an instance valid for a specific period of time and when the time expires the instance is destroyed. When using this approach the client negotiates with the factory about the time constraints of the instance.

#### 2.3.1.3 Time in OGSI

When a service requestor negotiates the expiration time of a GSR, or when it determines if an instance has expired it needs to model time. OGSI uses the GMT global time standard. Furthermore a synchronization protocol (such as Network Time Protocol) is needed for clients and servers to synchronize with GMT global time.

#### 2.3.1.4 Service Data

We've been talking about how Grid services can be thought of as stateful Web services, but in order to complete this description we need to extend the interface to include operations on state data. OGSI introduces an approach called Service Data that provides requestors with methods on the state data. The Service Data is local for every instance of a Grid service and there is no restriction on the quantity of Service Data Elements. A Service Data Element can hold non-technical information not suited for WSDL, such as cost, frequency of updates etc.

To avoid creating operations for every Service Data Element some basic operations for manipulating the data are included in the mandatory GridService portType. Basic operations like query, update and notification of change of Service Data Elements.

Service Data Elements (SDEs) are included in the portType element that they are associated with. The values of a SDE are simply called Service Data Element values, or SDE values. These values can be specified statically in the portType or dynamically assigned during runtime.

In Figure 2.9 is an example description of a Service Data Element meant to be used in a Route Price service, i.e. a service for looking up prices of routes. The Service Data Element specifies three elements, where each one is holding a value. The first element *hits* is intended for storing the number of invocations of the Route Price Service instance. The *lastRoute* is supposed to hold the most recently resolved route. The final element *statistics* is meant to hold some additional statistic values.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="RoutePriceData"
 targetNamespace="http://travelbusiness.com/travelquote/RoutePriceSDE"
 xmlns:tns="http://travelbusiness.com/travelquote/RoutePriceSDE"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:types>
<schema
targetNamespace="http://travelbusiness.com/travelquote/RoutePriceSDE"
        attributeFormDefault="qualified"
        elementFormDefault="qualified"
        xmlns="http://www.w3.org/2001/XMLSchema">

        <complexType name="RoutePriceDataType">
                <sequence>
                        <element name="hits" type="int"/>
                        <element name="lastRoute" type="string"/>
                        <element name="statistics" type="string"/>
                </sequence>
        </complexType>
</schema>
</wsdl:types>
</wsdl:definitions>
```

**Figure 2.9 An example description of a Service Data Element.**

### 2.3.2 WSDL Extensions and Conventions

As we already know OGSI uses WSDL to describe the interfaces of its Grid services. Due to the fact that WSDL doesn't support extensions, and given the restriction that all Grid services must extend the GridService portType, OGSI defines an extension element to handle this in WSDL 1.1 (this will be supported in WSDL 1.2 [21]). This extension only concerns the portType element of a WSDL document and is defined in a separate namespace with the prefix gwsdl. Figure 2.10 shows an example portType element extending the obligatory GridService portType. The portType includes the operation *GetRoutePrice* and the RoutePriceData SDE (seen in Figure 2.9).

```
<gwsdl:portType name="RoutePricePortType" extends="ogsi:GridService">
        <operation name="GetRoutePrice">
                <input message="tns:GetRoutePriceInputMessage"/>
                <output message="tns:GetRoutePriceOutputMessage"/>
                <fault name="Fault" message="ogsi:FaultMessage"/>
        </operation>
        <sd:serviceData name="RoutePriceData"
        type="data:RoutePriceDataType" minOccurs="1" maxOccurs="1"
        mutability="mutable" modifiable="false" nillable="false">
        </sd:serviceData>
</gwsdl:portType>
```

**Figure 2.10 An example gwsdl portType element.**

Apart from the GridService there are several other predefined portTypes (such as HandleResolver and Factory) in OGSI.

#### 2.3.2.1 Grid services portTypes

There are several predefined portTypes in the OGSI specification covering some common distributed computing patterns. With one exception (NotificationSink) they all extend the mandatory GridService portType. Therefore when implementing a Grid service application it must extend one of the predefined portTypes listed in Table 2.1.

| portType name | Description |
|---|---|
| GridService | standard portType including the mandatory behaviour of the service model |
| HandleResolver | maps a GSH to a GSR |
| NotificationSource | includes notification subscription |
| NotificationSubscription | defines the relationship between a source and a sink |
| NotificationSink | defines an operation for delivering notification messages |
| Factory | defines the standard operation for creating grid service instances |
| ServiceGroup | allows clients to manage service groups |
| ServiceGroupRegistration | operations making it possible for Grid services to join and leave ServiceGroups |
| ServiceGroupEntry | defines the relationship between an instance of a Grid service and a ServiceGroup it's participating in |

**Table 2.1 The Predefined portTypes in OGSI.**

**GridService portType**

The GridService portType holds the interface of the core functionality required by a Grid Service. This includes operations for manipulating Service Data elements and for destroying Grid service instances. The portType also includes several predefined Service Data elements, e.g. factoryLocator, gridServiceHandle, gridServiceReference and terminationTime.

The GridService portType can be compared to the standard Object in Object Oriented programming (which is the superclass of all classes).

### 2.3.3 Globus Toolkit 3

The Globus Toolkit 3 (GT3) is a complete implementation of the Open Grid Service Infrastructure (OGSI) and it's seen by many as a de facto standard in Grid middleware [22]. Globus Toolkit isn't just an OGSI implementation; it includes a lot of other services, utilities, etc.

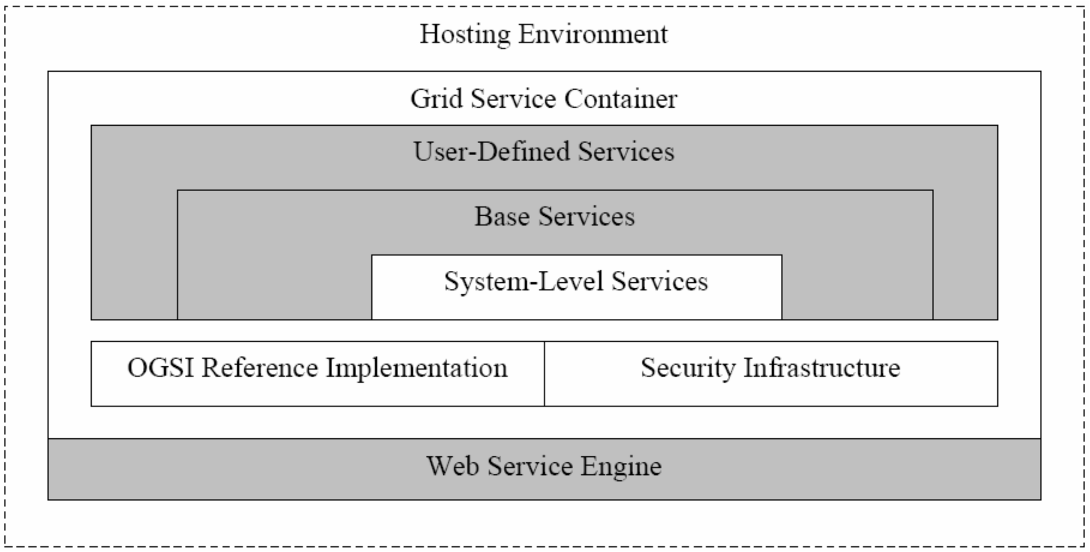


**Figure 2.11 Globus Toolkit 3 Core architecture [23].**

The white boxes in Figure 2.11 represent the *GT3 Core services*. Together they provide the essential parts for building and executing Grid services. The *OGSI Reference Implementation* provides implementation of the predefined portTypes listed in Table 2.1. These can be configured by the service provider to suit its own services. The *Security Infrastructure* implementation provides means for authentication and secure messaging. The parts seen so far are only the base for building Grid services and they aren't providing services during run time. The *System-Level Services* (such as logging-, management- and administration Grid services) on the other hand include services for maintaining Grid services. *GT3 Base Services* implement several services as job management, index services (allowing us to discover Grid services), and Reliable File Transfer

All the services described above must interact with the *Grid Service Container*, i.e. the OGSI run time environment. This container handles the maintaining of instances as well as incoming messages.

## 2.4 Agents

There are many definitions regarding the notion of an Agent. One thing that they usually have in common is ability of autonomous action. Other than that, the definitions usually have some differences.

In an attempt to clarify things we'll use the following definition of an Agent:

*An agent is an autonomous software program acting on behalf of a user, capable of interacting with the environment it's situated in, to achieve its goals.*

Using this definitions and the example given in the motivation of this project, one can describe a travel agent as: an autonomous software program, interacting with other agents and Web services in order to find the best traveling package with respect to the requirement and priorities of the traveler.

### 2.4.1 Intelligent Agents

The above example requires some intelligence of the agent to succeed in planning and ordering the best tickets for the trip. Considering this example we'll go through a list of suggested capabilities of an intelligent agent, by Wooldridge and Jennings (1995) [24].

- **Reactivity.** The tickets for a route in the overall plan taken by bus have sold out. The agents now rebuild its plan and the route will be taken by train instead.

- **Proactiveness.** In order to plan the trip the agent starts by contacting different traveling businesses, querying for price and timetables.

- **Social ability.** The agent must be able to interact with the traveler and other agents, e.g. to confirm orders or to negotiate price.

### 2.4.2 Multi-Agent Systems

Connecting intelligent agents together will give us a Multi-Agent System (MAS). These are some possible characteristics of MASs [25]:

- Agents has incomplete information or capability to solve problems on their own
- No global control of the system
- Decentralized data
- Asynchronous Computation

The interaction of agents can be self-interested or cooperative, where the latter will be the focus of this project. In a cooperative Multi-Agent System agents can share a common goal, or at least they can use each other's expertise to reach their own.

### 2.4.3 Market structure

The organizational structure of a MAS is concerned with the ways agents communicate and coordinate. The structure can take many different shapes but our interests in this project lie within a Market structure.

In a Market structure the control is distributed among the agents, that is competing for services or other resources, e.g. to buy tickets for a route. The valuation of services is mostly based on money but there can be other valuations as well. Among the default functionality of a Market structure one should find support for matchmaking, negotiation, communication and coordination. In addition to this the Market should have an open architecture and provide users means for exchanging the default functionality.

### 2.4.4 Agent Communication Languages

In order for agents to cooperate in a MAS, effective communication is required. One of the first Agent Communication Languages (ACLs) was Knowledge Query and Manipulation Language (KQML). KQML is a message-based language and can be thought of as an envelope format for messages, or as the outer language of a message. The thing that made KQML unique is the fact that it's based on speech acts, i.e. to treat messages as actions. Every message has a performative describing the intent of the message, e.g. *advertise* and *ask-one*. A KQML message has no restriction regarding the content of the message.

Due to some flaws in KQML the Foundation for Intelligent Physical Agents (FIPA) developed its own ACL similar to KQML. The FIPA ACL defines 20 different communicative acts (that corresponds to KQML performatives) along with their semantic interpretation. The two most important communicative acts are *inform* and *request*. The former is used by a sender to convince the receiver of the content and the latter is used by the sender to request an action to be carried out at the receiver. FIPA also has specifications covering message protocols and agent platforms.

```
(inform
        :sender         BusCompanyAgent
        :receiver       TravelingAgencyAgent
        :content        (price  trip 100)
        :language       sl
        :ontology       travel
)
```

**Figure 2.12 A FIPA ACL inform message.**

In Figure 2.12 there is an example of an FIPA ACL *inform* message and in Figure 2.13 there is an FIPA-*request* protocol.

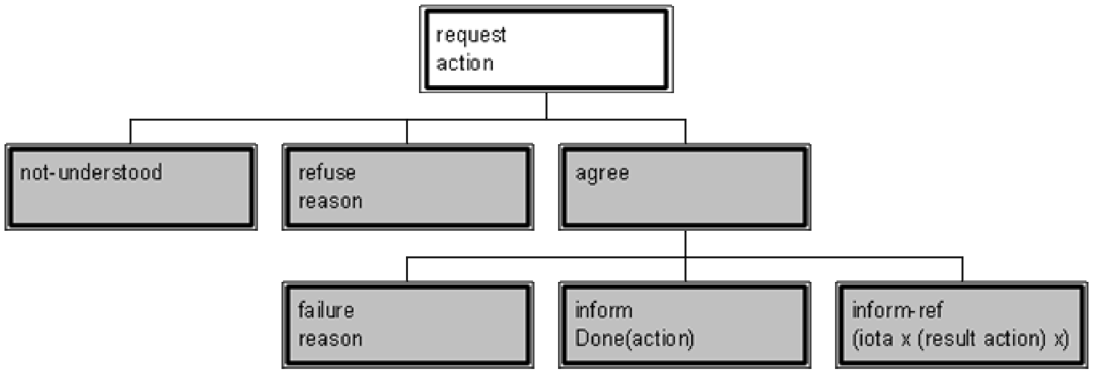


**Figure 2.13 FIPA-request protocol [26].**

## 2.4.5 Agent content language

The Agent Communication Language (ACL) is also known as the outer language and basically is a carrier of messages. An ACL can carry any type of message and it has no restriction regarding the language of the content. E.g. the content can be express in Semantic Language (SL) [27], Knowledge Interchange Format (KIF)[28] or in Resource Description Framework (RDF). Due to the fact that this project is closely connected to Grid services it becomes natural to have a content language expressed in XML. FIPA RDF is a content language fulfilling our needs regarding functionality and RDF is also recommended to be expressed in XML [29].

### 2.4.5.1 FIPA RDF Content Language

Before describing FIPA RDF Content Language it's necessary to at least have an idea about what the Resource Description Framework (RDF) is. As the name reveals RDF is a framework used for describing and exchanging metadata, i.e. information about information. RDF is basically about describing resources — a resource can be anything, but it must be represented by an URI.

RDF uses a triple called a statement when describing resources. A statement consists of three resources, i.e. a subject, an object and a predicate. The subject is associated with the object using the predicate. The subject could e.g. be *mailto:nimar@kth.se*, the predicate *http://www.it.kth.se/~it00_gni/Author* and the object *http://www.it.kth.se/~it00_gni/Masterthesis*. This could be translated into "Nimar is the author of the Master thesis". In Figure 2.14 is the resulting RDF document presented in XML. This example is closely related to an example given by [30].

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:home="http://www.it.kth.se/~it00_gni/>
  <rdf:Description
about="http://www.it.kth.se/~it00_gni/masterthesis">
    <home:Author rdf:resource="mailto:nimar@kth.se"/>
  </rdf:Description>
</rdf:RDF>
```

**Figure 2.14 A simple RDF document.**

FIPA RDF Content Language extends RDF to support basic functionality for expressing Objects, Propositions and Actions. An Object represents an identifiable entity in the domain; a Proposition is an extension of the RDF statement to include an additional truth value; and an Action expresses an act to be carried out by an object. This basic extension is called *fipa-rdf0*. FIPA defines several extensions but those won't be of interest in the scope of this project. The most important extension in our case is the ability to model Actions. In Figure 2.15 is an example action called *JohnAction1*. One can see that a namespace called *fipa* is included in the document and that every resource declared in this namespace is a part of the *fipa-rdf0* extension. An Action has three properties: an act represents the action to be carried out; an actor represents the entity to carry out the action; and finally an argument (an optional property) that can work as an input to execute the act.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:fipa="http://www.fipa.org/schemas/fipa-rdf0#">

  <fipa:Action rdf:ID="JohnAction1">
    <fipa:actor>John</fipa:actor>
    <fipa:act>open</fipa:act>
    <fipa:argument>
      <rdf:bag>
        <rdf:li>door1</rdf:li>
        <rdf:li>door2</rdf:li>
      </rdf:bag>
    </fipa:argument>
  </fipa:Action>
</rdf:RDF>
```

**Figure 2.15 An example FIPA RDF Action [31].**

### 2.4.6 Multi-Agents Toolkits

There is no universal definition of what Multi-Agent Toolkit is, or what it's supposed to include. Let's say that it's a software platform giving agent developers a higher abstraction, and allowing them to implement agents with the desired features.

One of the initial goals of this project was to develop two prototypes on different Multi-Agents Toolkits. For that reason we looked closer into Java Agent Development Environment (JADE) and Agora in order to examine their capabilities.

#### 2.4.6.1 JADE

Java Agent Development Environment (JADE) is a fully Java-based Open Source middleware for the development of multi-agent applications. According to the developers of JADE (TILAB [38]) it's the most frequently used Agent platform. The middleware is said to comply with the FIPA specification and it gives developers a higher level of abstraction. JADE also includes tools for debugging and deploying.

The agent platform can be distributed among heterogeneous machines, as long as JAVA run time environment is available. A platform includes a set of active containers, i.e. a running instance of the JADE environment and each container can hold several agents. Every platform must have a single special container, called a main-container. The main-container is the first container to be started and it must always be active because all the other containers (in a platform) are connected to it.

Agents use message passing to communicate in JADE (the platform keeps a private FIFO queue for every agent in the system). The messages follow the FIPA ACL specification. Agents can fetch their messages by using polling, blocking, timeout or pattern matching. The full FIPA communication model is implemented in JADE.

The easiest way to implement functionality in JADE is to use the predefined behaviors, e.g. OneShotBehaviour which is only executed once. The behaviors are organized as a FIFO queue where the scheduler runs the first behavior. The executing behavior must release the control in order for others to run.

### 2.4.6.2 Agora

Agora is a software architecture supporting implementation of software agents and agent-based marketplaces [32]. A central concept of Agora is an Agora node, i.e. a cooperative node allowing agents to communicate, coordinate and to negotiate. In other words one can say that an Agora node is a meeting place for agents allowing for cooperative work.

When initiating a new Agora node there are some default agents created on the fly. The *Agora Manager* is a default agent providing general managing and matchmaking functions. The *Coordinator* allows for coordination between agents and finally the *Negotiator* provides functionality for conflict resolution.

In addition to Agora nodes and the default agents, Agora also has Registered Agents. Basically the default agents are Registered agents with predefined functionality. When talking about market places a Registered agent can be either a seller or a buyer. These Agents can communicate in a peer-to-peer manner or through the Agora Manger. The messages are carried in a FIPA ACL and are sent via the Message Proxy as seen in Figure 2.16. The agent can communicate with a user as well, using the Log system. Each agent maintains a Knowledge Base containing a Prolog-based representation of its rules, messages and facts.

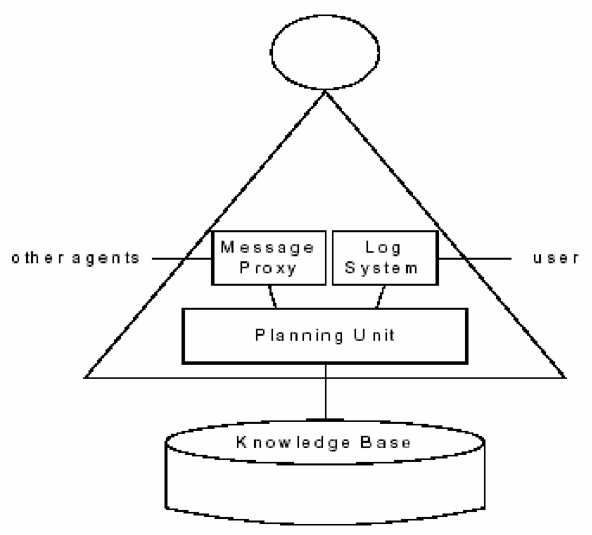


**Figure 2.16 Structure of an Agora agent [32].**

## 2.4.7 Java

Java is the high-level Object Oriented programming language central to this project, mostly because of its nice features when it comes to software agent developing. Java is an interpreted language where a program is compiled into an intermediate language, called Bytecode. The Bytecode is then interpreted at the target machine during runtime. Therefore Java is platform independent and it only requires a runtime library to be able to run applications. Furthermore Java supports a security mechanism called sandboxing, i.e. running untrusted applications in a secure way, by giving it limited access to system resources.

The main objection against Java is probably an insufficient performance, especially when it comes to thread handling. This is addressed in JADE by assigning only one thread of control to each agent. Concurrency in JADE is achieved by having the agent adopting some predefined behaviours.

## 2.5 Ontology

When talking about exchanging knowledge one can say that an ontology is a specification of concepts. Furthermore one can say that it's an attempt to create a simplified view of the world (we are trying to represent). Ontologies can be used by agents as a vocabulary when they communicate with each other. In this project we will focus on the Web Ontology Language (OWL) and especially on a service oriented ontology based on OWL called OWL-S.

### 2.5.1 OWL

OWL is a semantic markup language, allowing for exchanging information about ontologies, based on Resource Description Framework (RDF) [33]. One can say that OWL is a vocabulary extension to RDF but it also puts restrictions on the ordinary RDF vocabulary. The main benefit with OWL is that it provides a much richer expressiveness than the RDF Schema (RDFS). There are three different species of OWL: *OWL full*, which is a union of RDF and the OWL syntax and gives the user full expressiveness; *OWL DL* is a subset of OWL full and is closely related to Description Logic (DL); and finally *OWL lite*, which is a subset of OWL DL with focus on simplicity for developers. The usage of species in OWL increases the flexibility while the user can choose the species fulfilling his requirements.

### 2.5.2 OWL-S

OWL-S is an ontology based on the OWL language with the main purpose to provide a mark-up language for representation of Web services [34]. The idea is to allow for automatic discovery, invocation, composition and interoperability, and execution monitoring of Web services. Previous versions of OWL-S where known as DAML-S and where built upon a predecessor of OWL called DAML.

In OWL-S a service is represented by the class *Service*, which holds three other classes used when describing it. First there's the *ServiceProfile,* which is a class describing what is required of a requestor of a service and even more important — what is provided for them. The ServiceProfile becomes especially interesting when talking about provision and selection of services. Furthermore there is a class called ServiceModel describing how a service works; and finally a description on how to access the service, is found in the ServiceGrounding class. Due to the fact that this project is concerned with Provision and Selection of services we will focus on the ServiceProfile. We will also shortly consider the ServiceModel (and especially the ProcessModel) which has interesting features when describing composite services.

#### 2.5.2.1 The ServiceProfile

As mentioned above the ServiceProfile is a description on what to expect from a service. This can be used by both by a services requestor to describe a requested service or by a service provider to describe its provided services. To find a suitable service for the requestor is consequently a matter of matching the provided services against the requested.

The ServiceProfile holds information such as service name, text description and contact information (optional in version 1.0). It also holds functional information describing parameters, preconditions and the effect of the service. The ServiceProfile also contains attributes for classifying services. Some properties, especially interesting regarding this project, are *serviceName*, *hasInput*, *hasOutput* and *serviceCategory*. The first property serviceName is the name of the service and can be used as an identifier. hasInput refers to the one or several *Input* resources, i.e. parameters required when executing the service. In contrast to hasInput, hasOutput refers to one or several *ConditionalOutput*, this because one might not now the outcome of a service execution. The ConditionalOutput makes it possible to associate conditions to the output parameters. The final property of interest, serviceCategory, includes information for categorization of services. There are few (if any) constraints about how make use of this property. serviceCategory contains the four text fields: *categoryName* is the name of the category and it could e.g. be represented by literal or an URI; taxonomy refers to an URI or a literal of the taxonomy currently used; *value* points to the value of the service in the current taxonomy; and finally *code* which stores some code associated with type of service.

#### 2.5.2.2 The ServiceModel

Once a service has been selected the ServiceProfile becomes rather useless. In order to interact with the service a description of how the service works is needed. This is given by the *ServiceModel*. OWL-S 1.0 defines a subclass of the ServiceModel called the *ProcessModel*. The central concept of the ProcessModel is the *process* entity, i.e. a data transformation from a set of inputs to the corresponding set of outputs. Another viewpoint is that a process is a state transformer. In similarity to the ServiceProfile a process has inputs, output, preconditions and effects. When describing the same service these properties are naturally the same, though this isn't required.

The ProcessModel defines the three different types of processes, i.e. atomic, simple and composite. Atomic processes are directly invocable (by sending the appropriate input message). Furthermore they have no sub processes, i.e. they appear to be executed in an atomic way (to the requestor). Simple processes are not directly invocable, i.e. they are not associated with any grounding. But like atomic processes they appear to be executed in an atomic way. Simple processes are used as elements of abstraction, e.g. to view a special usage of an atomic process or in a simplified representation of a composite process. The final type, i.e. the *composite* process, consists of several other processes (which can include other composite processes). Each composite process must have some kind of control structure of its composition. The control construct can be associated with additional properties, allowing for ordering or conditional execution of the sub processes. The OWL-S specification predefines several control structures such as *Sequence*, *Split* and *If-Then-Else*.

### 2.5.3 Tools

When working with ontologies it's desirable to have an Application Programming Interface (API) easing the work, e.g. handling the parsing of input messages. In spite of the fact that OWL-S is a quite new technology, and there aren't a lot of tools available yet, we have managed to find some tools of interest.

#### 2.5.3.1 Jena

Jena is open source semantic Web framework for Java initiated by a research group at HP [35]. Jena includes an RDF API together with a parser and a writer of RDF in XML. Persistent storage is also supported by use of a database engine, which support RDQL [36] queries. The most interesting part in our case is the Ontology API that has support for OWL along with some other languages. The Ontology API is closely coupled to a rule based reasoning system.

#### 2.5.3.2 OWL-S API

OWL-S API is a Java API for managing OWL-S [37], i.e. to parse, write and execute services based on OWL-S. The API support several versions of OWL-S, including OWL-S 1.0. OWL-S API is built on top of Jena, which is providing the underlying data model.

## 2.6 Composition

When searching for a service it might be the case that a single service fulfilling the requirements is nowhere to be found. This doesn't mean that the requirements can't be fulfilled by combining several services into a composite one, i.e. using composition of services. In order to increase the flexibility and the hit rate of the system we considered composition of services. As mentioned in the section covering related work, the goal of the over-all project is to develop a technique for selection and composition of Web services based on logic with high expressive power. The main benefit of using logic, when composing services, is that it can be guaranteed that the composed service fulfills the requirements. We choseLinear Logic (LL) due to the fact that it fulfilled our requirements and it's probably one of the most investigated ones.

### 2.6.1 Linear Logic

Linear Logic (LL) can be seen as a refinement of classical logic; it considers process states, events, or resources rather than truth values. Furthermore the propositions aren't considered to be static unchanging facts but dynamical properties or finite resources. In order to simplify things one could say that assumptions correspond to resources and the conclusions to requirements fulfilled by spending the given resources.

We will go through some of the major changes between classical logic and LL. First of all two structural rules seen in the classical logic has been removed, namely *contraction* and *weakening*. One could say that the former allows us to use a premise (or assumption) unlimited number of times and the latter allows us to prove a proposition using irrelevant or unused premises. This isn't allowed in LL due to the fact that each assumption is expected to be used exactly once in each proof. Removing the two structural rules leads us to the next major change, i.e. introduction of two forms of *conjunctions* and *disjunctions*. Both of them have a multiplicative as well as an additive form. These are described further in Table 2.2. The final change to be considered is the introduction of modality, i.e. a storage or reuse operator. There are two different modality operators (both described in Table 2.2). Modality can be used to distinguish between non-consumable resources as information from consumable ones as memory. Another important concept, the linear implication, is also described in Table 2.2.

| Name | Example expression | Description |
|---|---|---|
| Additive conjunction | $A \& B$ | Either *A* or *B*, it's "one's own choice". |
| Multiplicative conjunction | $A \otimes B$ | This expression stands for the usage of A and B at the same time. |
| Additive disjunction | $A \oplus B$ | Either *A* or *B*, but it's "someone else's choice". |
| Multiplicative disjunction | $A \wp B$ | The meaning is "if not *A* then *B*". |
| Modality (unlimited creation) | $!A$ | This expression provides unlimited use of resource *A*. |
| Modality (unlimited comsumption) | $?A$ | This modality operation provides unlimited consumption of resource *A*. |
| Linear implication | $A \multimap B$ | The linear implication can be thought of as "*B* can be derived using *A* exactly once." |

**Table 2.2 Connectives and operators in linear logic.**

# 3 Analysis and design

The aim of this chapter is to discover the functionality required of the system, along with a plan on how it could be implemented. The first sections in the chapter will describe how the system might be used and what functionality to expect. Then there will be sections focusing on the interaction in the system. After that we will look at the information flow in the system.

## 3.1 Terminology

The most important terms, crucial for understanding this chapter, are explained in Table 3.1. Some of the terms are only defined within the scope of this project while others are widely accepted.

| Term | Explanation |
|---|---|
| Service Provision Agent (SPA) | An agent handling the service requestor's part in a negotiation of services. |
| Service Selection Agent (SSA) | An agent handling the service provider's part in a negotiation of services. |
| Directory Facilitator (DF) | A predefined agent holding a directory where other agents can publish themselves and search for others. |
| Ontology | An ontology is a specification of concepts. It can be used by agents as a vocabulary when they communicate with each other. |
| Virtual Organization (VO) | A set of individuals/organizations conformed under a set of rules for sharing resources. |
| VORegistry | A registry included in Globus Toolkit 3 allowing other Grid services to register and lookup services within a Virtual Organization. |

**Table 3.1 Explanation of the most important terms used in the document**

## 3.2 Scenarios of system usage

There will be two different types of users in the system; those who provide services and the ones requesting them. In the first two scenarios of system usage we will see how both sides can make use of the system. The final scenario is more detailed and it will be used as a base when testing the implemented prototype.

### 3.2.1 Service requestor

Clark Kent is studying computer science. In one of the courses he is taking, a huge mathematical problem has come in his way. Using his own work station isn't really an option due to the fact that he needs the results right away, and running the problem on his work station could take forever. Instead Clark reminds himself of a system available at school for selection and composition of Grid services. If he could find a Grid service with the required storage capacity, a fee not too big for his student loans, and with the required CPU capacity the problem would be solved. The next day Clark starts his school day by instantiating his own agent with the directives to find a suitable service. Clark feels relieved when his agent presents him plenty of services fulfilling his requirements. His only problem now is to choose among the services.

### 3.2.2 Service provider

Lois Lane runs a quite large company working with 3D rendering. The company has just invested in a new super computer with high computational power along with large data storage. At the present time the computer is rarely fully loaded and Lois hates to see the waste of resources. Therefore Lois comes up with the idea to offer computational power to outside companies and others, using Grid service technology. Lois talks to her staff administering the super computer and they suggest connecting the computer to a novel system where agents negotiate Grid services in a market place. Lois decides to go along with the suggestion and in a nearby future the money spent on the super computer will hopefully be repaid.

### 3.2.3 Detailed Scenario

We have all seen the huge success of Short Message Service (SMS), i.e. a service for mobile phones allowing for users to send short messages to each other. Lex Luther owns a small company providing services for mobile users. But due to the fact that his company is rather small, in comparison with the large mobile communications providers, he has a hard time trying to keep up with the competition. Lex must somehow make use of the competition between the large companies. He finds out that most of the companies provide services such as SMS, ring tones, MMS and number lookup to their users by the means of Grid service technology. One of the main reasons why Grid services are well-suited for these kinds of services is the ability to attach additional Service Data (such as cost and currency). Furthermore, it might be convenient to charge a user for all messages sent using its instance instead of charging each message sent.

When Lex hears about an agent-based system for Grid service provision and selection he comes up with the brilliant idea to use the system for providing the services offered by other companies. Each company will be represented by a Service Provision Agent, handling the providing part in a negotiation over services. Lex will provide the counterpart (Service Selection Agent) configured to select the cheapest service of interest, e.g. sending SMS.

Using this system Lex will always provide the cheapest service to his costumers. His own income will instead be based on advertisement on his nowadays well visited site.

## 3.3 Use cases

Use cases are a widely used method for capturing functionality and behaviour of a system. A Use case describes the interaction between the system and an outside party trying to achieve a goal while using the system. The focus of this project lies within selection and provision of Grid services. Therefore it becomes quite natural to use selection and provision of services as the two main types of Use cases. The Use cases will be described in details below.

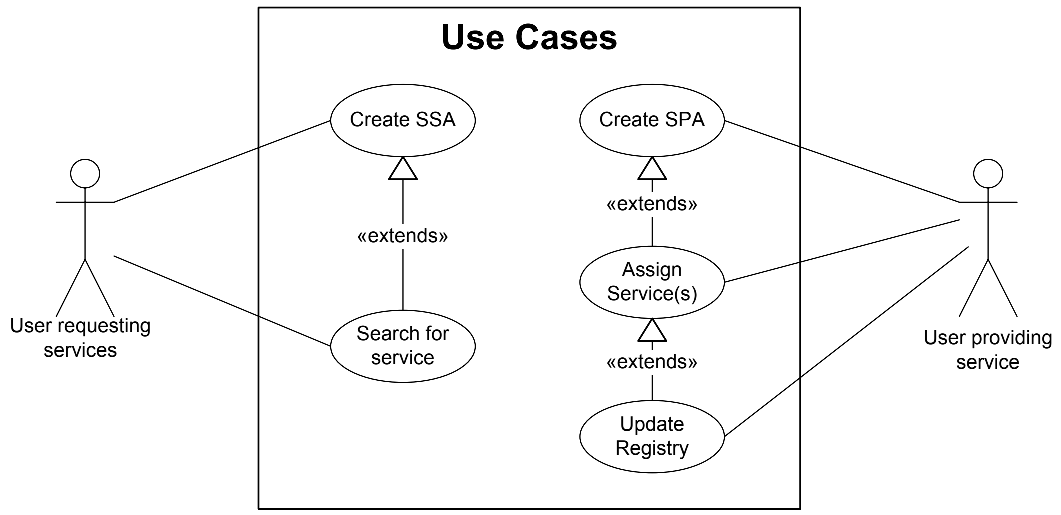


**Figure 3.1 Use cases of the system. Both selection and provision of service are divided into several minor Use cases.**

### 3.3.1 Provision of service

In order to select a service we must find a way to provision them first. This includes finding a way to describe Grid services and making these descriptions available for agents handling the service providing part in a negotiation, i.e. a Service Provision Agent (SPA). Furthermore the SPA must be found by the counterpart of a negotiation, i.e. the Service Selection Agent (SSA). This can be solved by having the SPA publishing itself in a Directory Facilitator. The Service Provision Agent must also be updated when new services appear or when services become unavailable.

The Use case Provision of service can be divided into three minor Use cases (as seen in Figure 3.1). First there is one describing the creation of a new Service Provision Agent. Secondly, there is a Use case covering assignment of a service or several services to a SPA, and finally a Use case dealing with updating of the registry.

**Creation of a Service Provision Agent**
When a new market place is set up the only agents initialized are a number of predefined agents included in the architecture, e.g. a Directory Facilitator. Therefore in order to provision services in the market place at least one Service Provision Agent must be instantiated.

**Assignment of Services**
To provision a service one must assign it to a Service Provision Agent. Assigning services one at a time might become inefficient when dealing with larger organizations. A solution to this problem is to assign an entire Virtual Organization to a SPA. But it isn't always the case that all the services are meant for provisioning. Therefore in our solution a registry (in our case the VORegistry), holding the address along with a description of each offered service, is given to the SPA instead. Our solution supports both assigning a single service at a time, or if one like, a whole registry.

When a new service is assigned to a SPA the properties of the new service must be translated into an ontology object (currently supported by the SPA). The ontology will be the vocabulary used when negotiating about services.

**Updating the registry**
When assigning a service or a Virtual Organization to a Service Provision Agent, it includes caching a description of the service or the current state of the registry at the SPA. Caching is an efficient way to avoid the communication delay when working with few registries and it also minimizes the logic required in the registry (such as advanced search algorithms). To keep the registry and the services coherent with the associated SPA we need to notify the SPA whenever changes are applied to the registry.

### 3.3.2 Selection of service

Selecting a service is a matter of matching the provided services against the one described by a user. A Service Selection Agent handles the user's part in a negotiation over services, and the counterpart, i.e. the service provider, is handled by a Service Provision Agent. In similarity to the Use case covering provision of service we have divided this Use case into separate parts (as seen in Figure 3.1). The parts are: Create Service Selection Agent and Search for service.

**Create Service Selection Agent**
As mentioned in the Use case describing creation of SPA there are no Service Selection Agents in the system when the market place is set up. Therefore one must instantiate at least one SSA in order to be able to make use of the system's selection service.

**Search for service**
When a Service Selection Agent has been instantiated one might want to start negotiating over services. In order to do so, a SPA or several SPAs (under the condition that they all use the same ontology) must be chosen. We have restricted our negotiation not to automatically include all the available SPAs. The motivation for this is mainly based on scalability issues; hence there can be a great number of SPAs in a market place. Another reason is the restriction to only use a single ontology when making a query and due to the fact that the system supports multiple ontologies. Instead the SSA chooses among the SPAs available in the Directory Facilitator. The goal is to enable for the user to specify the depth of the search.

When it's decided which of the SPAs to interact with, a sample service is constructed using the associated ontology supported both by the SSA and the SPAs. The reason for allowing multiple ontologies is to increase the flexibility of the system; it's not feasible to find an ontology suited for all Virtual Organizations and kinds of services.

The SPA searches its storages for services matching the sample service. The negotiation will hopefully result in a list of possible services.

## 3.4 Interaction

Whether we consider selection or provision of services it's required from the agents to interact with each other. First we will be using collaboration diagrams to describe this interaction on a higher level of abstraction. Then a more detailed description will be given for each agent-to-agent interaction in sequential diagrams. In similarity to the Use cases we have divided the interaction into two collaboration diagrams, one describing selection and the other provision of service.

### 3.4.1 Provision of service

Except for the three Use cases regarding provision of service a user's ability to create and add Grid services (to a Virtual Organization) has been included in the collaboration diagram (1.1-1.2 in Figure 3.2). This isn't really part of the system and it's therefore not described in any particular Use case. In contrast to this updating the registry (5.1-5.7 in Figure 3.2) has a direct effect on the system and is therefore described in a Use case.

As mentioned above a user providing services must instantiate a Service Provision Agent in order to provision its services in the system (2). After an instantiation a user can assign either single services or one or several Virtual Organizations to the SPA. Assigning a Virtual Organization can be seen as assigning multiple single services. Therefore we will only give a textual description of assignment of a VO.

The first step after a user has initiated an assignment of a Virtual Organization is to fetch all the services located in the Registry belonging to the Virtual Organization (4.2-4.3). The SPA fetches the information needed to construct a description of each service (using a supported ontology) found in the registry (4.4-4.5). Finally the SPA publishes itself at the Directory Facilitator (if not registered before) (4.6). If the registry would change after being assigned to a SPA (typically a new service is added) the registry triggers a notification message that is sent to the associated SPA (5.1-5.3). This would cause the SPA to update the list of services held by the registry (5.4-5.5) and to update its local storage (5.7).

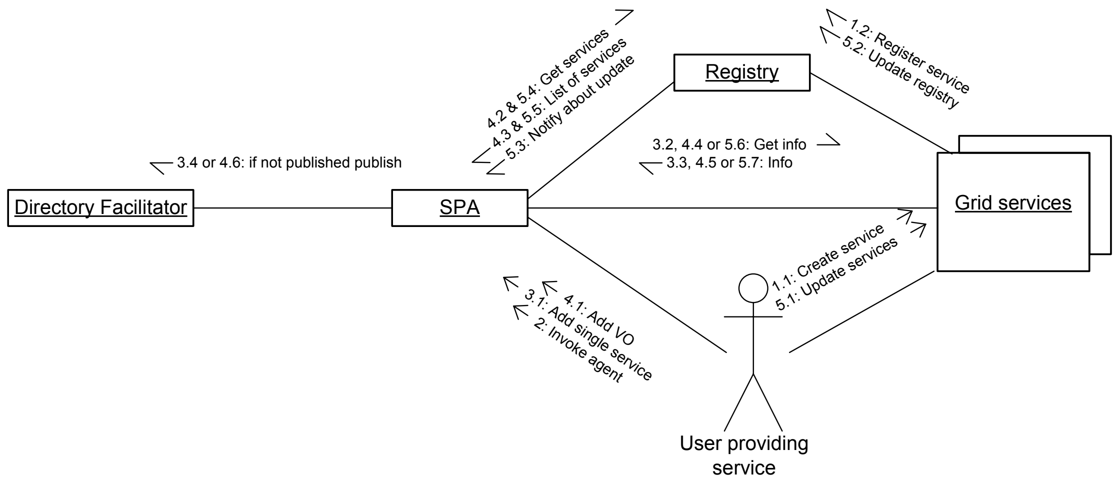


**Figure 3.2 Collaboration diagram describing the interaction between different parts of the system used to provision services.**

### 3.4.2 Selection of service

In the collaboration diagram describing selection of services (see Figure 3.3) the Service Selection Agent interacts with two different agents. When a search is initiated by an users the agent first downloads a list of available SPAs from the Directory Facilitator (2.1-2.3). Then the SSA automatically selects the SPAs to interact with (based on the supported ontologies). Finally the SSA translates (the user specified) service of interest into the ontology of concern. The SSA then starts negotiating with the SPA (or the SPAs) about suitable services (2.4-2.5). When the search has timed out the result is shown to the user (2.6).

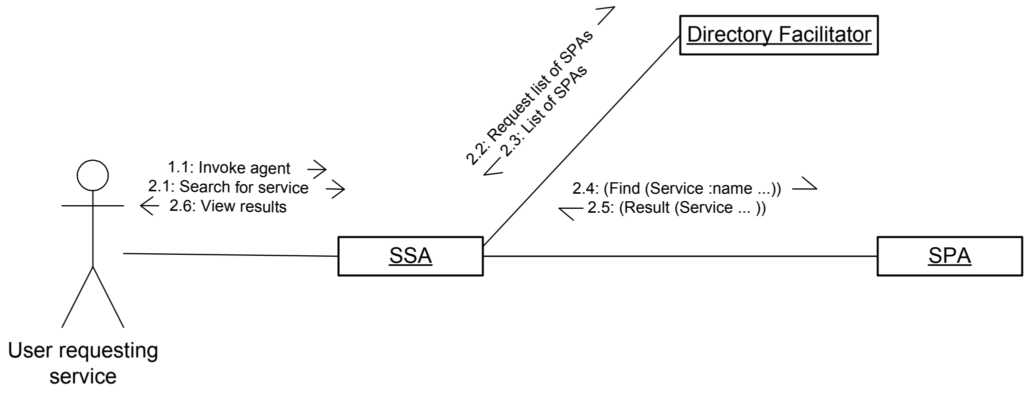


**Figure 3.3 Collaboration diagram describing the interaction between different parts of the system used to select services**

## 3.5 Agent-to-agent interaction

When two agents communicate with each other it's required to have an effective communication language. FIPA ACL is probably one of the most well-defined Agent Communication Languages (ACLs) and it's also widely accepted. Therefore it will be used as the outer language when agents communicate in this project. We'll be using sequential diagrams to specify the different FIPA communicative acts exchanged. A short description of the communicative acts used in the protocols will be given in Table 3.2.

| Communicative act | Description |
|---|---|
| inform | The sender of an inform ACL wants the receiver to believe that the contents of the message (i.e. a statement) is true. |
| request | The sender of a request ACL wants the receiver agent to perform some action. |

**Table 3.2 The FIPA Communicative acts used in the project.**

The agent-to-agent communication is only initiated by the Service Provision Agent and the Service Selection Agent. Therefore their interactions with other agents will be separated into two different sequential diagrams.

### 3.5.1 Service Provision Agent Protocols

As seen in Figure 3.4, a Service Provision Agent registers at the Directory Facilitator (DF). The communication with Directory Facilitator is specific for each agent platform, but it's described in Figure 3.4 using some kind of register message. When the SPA is registered at the DF it's considered to be available for any SSA wanting to start a negotiation.

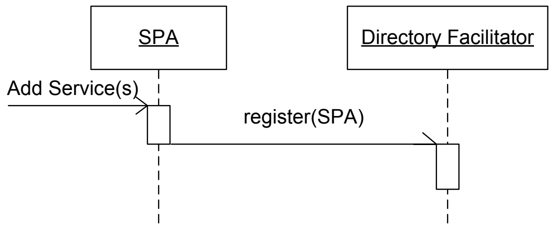


**Figure 3.4 The Protocols used by the Service Provision Agent. The register message is a platform specific method and is not to be mistaken for FIPA ACL message.**

### 3.5.2 Service Selection Agent Protocols

The steps required when selecting services in the system are described in Figure 3.5. These steps include the Service Selection Agent to interact with two other agents. The first agent to interact with is the Directory Facilitator (DF). This interaction is described in Figure 3.5 using messages to represent the platform specific methods as seen in Figure 3.4. The messages of concern are the search and the corresponding result messages. These are used by the SSA when it searches the DF for available SPAs.

Using an ontology the SSA formalizes the user's query, which is then sent to the selected SPA (or SPAs) using a FIPA ACL request message. The SPA tries to match the requested service with the ones assigned to it. The result is then sent back to the SSA using an FIPA ACL inform message.

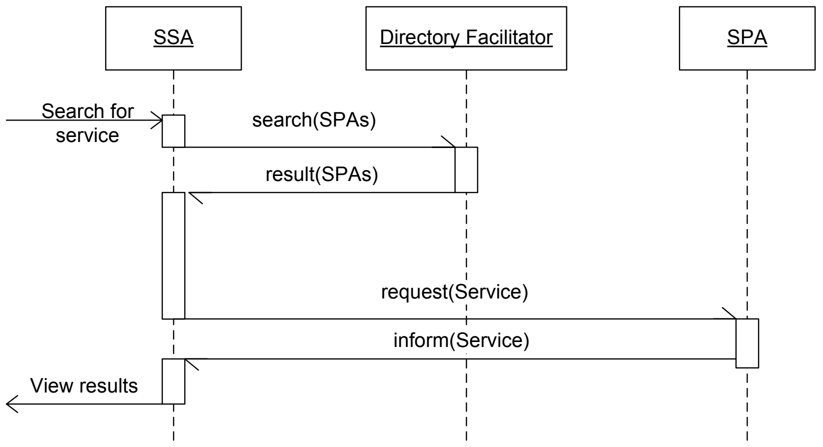


**Figure 3.5 The Protocols used by the Service Selection Agent. The search and result messages are platform specific methods and are not to be mistaken for FIPA ACL messages.**

## 3.6 Message Content

There are two messages, not specified by the agent platform, in the system. These are the *inform* and *request* messages seen in Figure 3.5. The request message is in this case sent (by a Service Selection Agent) as a proposition (to the receiving Service Provision Agent) to initiate a search. The message must therefore contain both its purpose, i.e. requesting a search for services, and not less important criteria for the requested service.

We will use FIPA RDF Content Language as our base content language; because it supports expressing actions and due to fact that it can easily be expressed in XML. An example action expressed in FIPA RDF is given in Figure 3.6. Each action has an *actor*, an *act* and optionally an *argument* element. The actor is the agent requested to execute the proposed action, which is described by the act. Input to the action can be held in the argument element. In Figure 3.6 it's requested of the SPA1 agent to find a service described by the argument element.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas#">

        <fipa:Action rdf:ID="45">
                <fipa:actor>SPA1</rdf:actor>
                <fipa:act>findService</rdf:act>
                <fipa:argument>
                <... description of requested service in OWL-S .../>
                </fipa:argument>
        </fipa:Action>
</rdf:RDF>)
```

**Figure 3.6 A FIPA RDF Action requesting a search at SPA1.**

Every action is identified with its own id. This is important when associating the action with its response. In Figure 3.7 is the response message to the action given in 3.6. The response message has an element named done, which notifies the requestor, of the proposed action, if it was successfully executed or not. If the action was successfully executed the result element will hold the produced output, if any. The result message in Figure 3.7 notifies the agent requesting the service in Figure 3.6 that the action was successfully carried out. Hopefully the receiver of the message will also find the best match when extracting the content of the argument element.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:fipa="http://www.fipa.org/schemas#">

        <rdf:Description about="45">
                <fipa:done>true</fipa:done>
                <fipa:result>
                <... service described in OWL-S .../>
                </fipa:result>
        </rdf:Description>
</rdf:RDF>
```

**Figure 3.7 A response containing the result of the FIPA RDF Action message.**

## 3.7 Matching algorithm

The goal of the system is to select the service best suited for a user's requirements. This is possible by translating the user's requirements into a synthetic service (represented in the chosen ontology) and then comparing it to the provided services expressed in the same ontology. This project will focus on the OWL-S ontology for representation of services. Therefore will OWL-S also be the base for the matching algorithm.

In order for the matching algorithm to achieve tolerable results, it should comprise following properties:

- Encourage the requestors as well as the providers of services to be detailed in their descriptions of services.
- Include semantic matching of inputs and outputs.
- Allowing for prioritizing the search categories.

Figure 3.6 shows a matching algorithm comprising these features. The algorithm matches the requested service against all the services provided by the Service Provision Agent. Each of the provided service is assigned an integer called a score — a high score means that that the service is well suited. The service that receives the highest score is the one returned to the requestor (which will encourage requestors and providers of services to give detailed descriptions of their services). The score is based on a weighted addition of the result of four different comparison methods. The weights make it possible for prioritizing between the methods, e.g. when using a simple SMS service the output parameters might be uninteresting. The first method matches the inputs parameters; the second the output parameters; the third the names of the services; and finally the fourth method matches the taxonomy. More detailed descriptions of the methods will be given in the upcoming sections.

```
Service Match(reqestedService, providedServices){
 int highestScore
 Service hasHighestScore

    for all providedService in providedServices do{
        int currScore = 0
        currScore += weightInput *
                matchInput(reqestedService, providedService)
        currScore += weightOutput *
                matchOutput(reqestedService, providedService)
        currScore += weightName *
                matchName(reqestedService, providedService)
        currScore += weightTaxonomy *
                matchTaxonomy(reqestedService, providedService)

        if currScore > highestScore do{
                highestScore   = currScore
                hasHigestScore = providedService
        }
    }
 return hasHigestScore
}
```

**Figure 3.8 The service matching algorithm.**

#### 3.7.1.1 Matching input parameters

When matching parameters the main concern is to match the type of the parameters rather than their names. Furthermore we're not interesting in the order the parameters are positioned in. A matching succeeds when there is a one-to-one mapping between the input parameters of the requested services and the input parameters of the provided service currently being matched. Another way for the matching to succeed is when no input parameter is specified in either description.

#### 3.7.1.2 Matching output parameters

The matching of output parameters is identical to the matching of input parameters (described in 3.7.1.1).

#### 3.7.1.3 Matching service names

Matching service names is done by comparing the names of the services using a lexical analyser. One approach is to demand the names to be identical. Another less restrictive approach is to allow one of the names to be a substring of the other, e.g. a service named *add* would match a service called *addition*. This last approach is the one chosen for our matching of names.

#### 3.7.1.4 Matching taxonomy

Taxonomy allows for valuation of services. This can be useful for requestors with a limited budget. There is no restriction in the OWL-S specification about how to use their taxonomy. The taxonomy consists of the four fields *Code*, *CategoryName*, *Taxonomy* and *Value*. When matching taxonomy we will require that the Code, CategoryName and Taxonomy fields are equal when comparing the two descriptions. The values are considered to match when the requestors value is less than, or equal to, the one being compared.

## 3.8 Information flow

The information flowing in the system is basically different forms of service descriptions. Figure 3.9 shows how these service descriptions are transformed as well as their connections to each other, i.e. via information processing classes. New information can enter the system in two different ways. One possibility is when Service Selection Agent (SSA) creates a new synthetic service (selection of service). The other, when a Service Provision Agent (SPA) fetches the WSDL document along with the associated Service Data Elements from a Grid service (provision of service). First we'll consider the former case when a new synthetic service is created.

When a user wants to locate a service, it first needs to specify the requirement for the service. This will basically result in a WSDL operation and some Service Data Elements. These objects will be converted into a single service description object in the preferable ontology, using a *WSDL 2 ontology translator* (i.e. a class translating a WSDL document into a specified ontology object). Once the object has been constructed it's to be sent to a SPA (as the content of an ACL) and is therefore (with the use of an *ontology writer*) transformed into an XML document representing the service. The receiving SPA resolves the ACL and extracts the XML document, which is retransformed back to an ontology object (using a class called an *ontology reader*). The object will be matched against the advertised service description objects (located in the local storage) using a *Service Matcher*. The Service Matcher will return the best matching object, which the ontology writer transforms in to an XML document. The XML document will be sent back to the

SSA in an ACL message. The content XML document is once again extracted form the ACL message and is transformed (using an ontology reader) into an ontology object. The ontology object hopefully matches the user's requirements.

The other flow of information is when a new Grid service is assigned to a SPA (called provision of service). Using the Grid Service Handle (GSH) of the Grid service the SPA fetches the WSDL document along with associated Service Data Elements. The *WSDL service creator* then converts the WSDL document into one or several WSDL operation objects (one for each operation not considered to be a standard Grid service operation). Each WSDL operation object is combined with the Service Data Elements in the *WSDL 2 Ontology translator* creating an ontology object representing the service. These objects are then stored in the local storage for latter comparison with incoming service requests.

The main reason, why service descriptions represented in XML are translated into object-based service description, is to facilitate the extraction of data from service descriptions. This is a huge benefit when handling a large number of service descriptions in the matching algorithm. The translation from XML to object doesn't necessarily cause information loss as long as the translator and the ontology object implements the entire ontology.

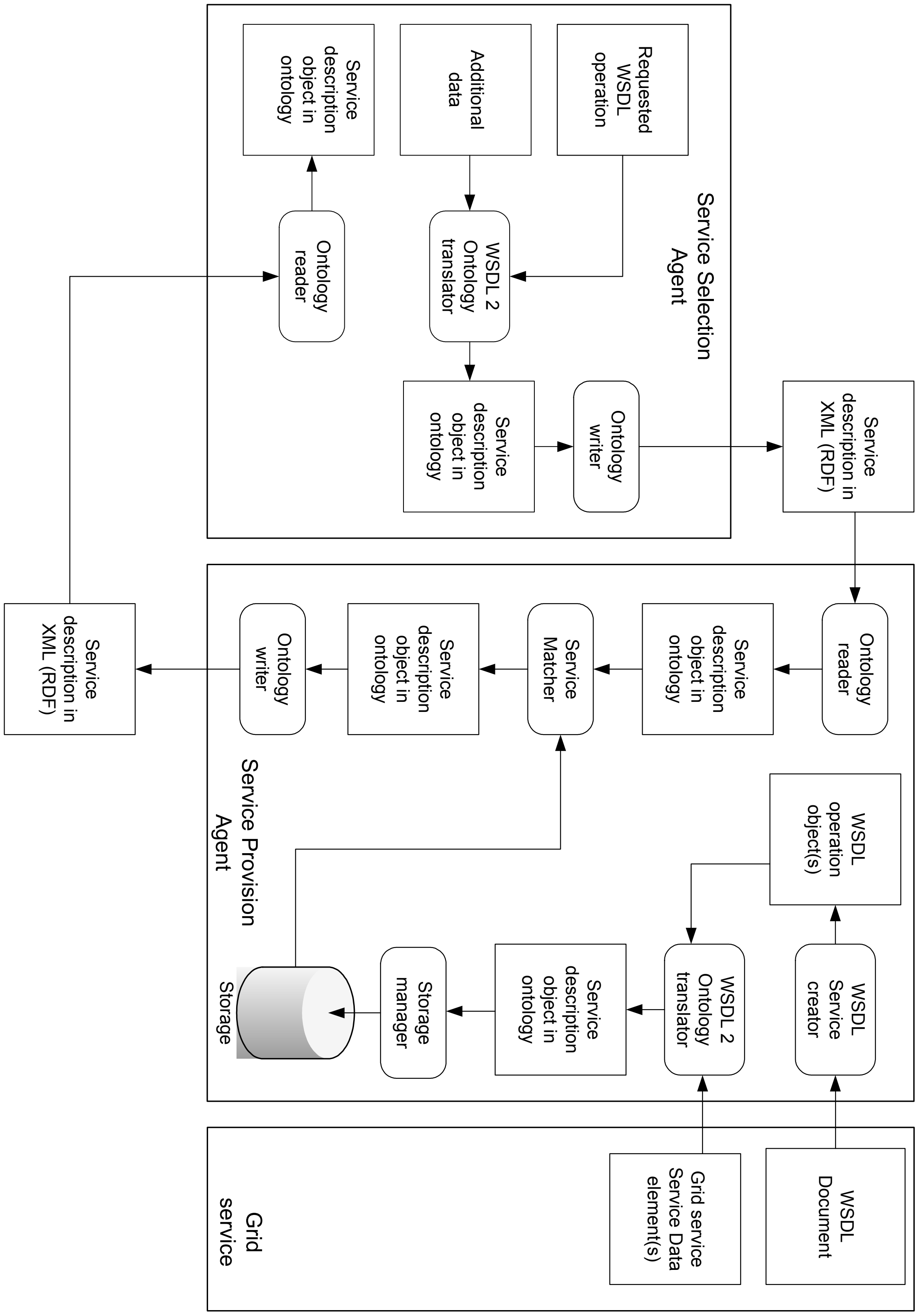

Service Selection Agent
Requested WSDL operation
Additional data
Service description object in ontology
WSDL 2 Ontology translator
Ontology reader
Service description object in ontology
Ontology writer
Service description in XML (RDF)
Service Provision Agent
Ontology reader
Service description object in ontology
Service Matcher
Service description object in ontology
Ontology writer
Service description in XML (RDF)
WSDL operation object(s)
WSDL Service creator
WSDL 2 Ontology translator
Service description object in ontology
Storage manager
Storage
Grid service
WSDL Document
Grid service Service Data element(s)

**Figure 3.9 The information flow in the system.**

## 3.9 Composition

As mentioned in the section covering composition in related technologies (2.8), one can increase the hit rate and flexibility of the system by introducing composition of services. In this section we'll see how support for composition could be implemented in our prototype. Our proposed design is closely related to a system architecture for Web service composition described by J. Rao, P. Küngas and M. Matskin [13]. Their architecture is based on a Linear Logic theorem prover called RAPS. First of all we will give an overview of our proposed design. Then the most crucial parts of the design will be described in further details.

### 3.9.1 Overview of the proposed design

Extending our design to include composition of services will result in replacing the proposed matching algorithm with a new one, which supports composition. The agent-to-agent communication doesn't necessarily need to be changed; the SSA can still send the requested service in a *findService* action and receive the corresponding result in a response message. So the SPA receiving a findService message will try to compose a service on behalf of the SSA (using its locally stored service descriptions). Notice that the matching algorithm could still result in service represented by a single atomic process (i.e. a single invocable service).

Figure 3.10 visualizes the actions taken by the (composition) matching algorithm of a SPA when receiving a request message. First of all the locally stored advertised services are translated (using the translator) into extralogical axioms in LL, which are latter used by the theorem prover. The requested service description however is translated into a LL sequent, i.e. the formalized statement that we are trying to prove. The classes and other properties are sent to the adaptor, which will ask the semantic reasoner to investigate if there are any subtypes. The subtypes (if any) are then sent to the theorem prover as LL axioms. The LL theorem now tries to prove the sequent, i.e. to see if the advertised services can be combined to fulfill the requirements. If a proof can be derived it's translated into OWL-S process model.

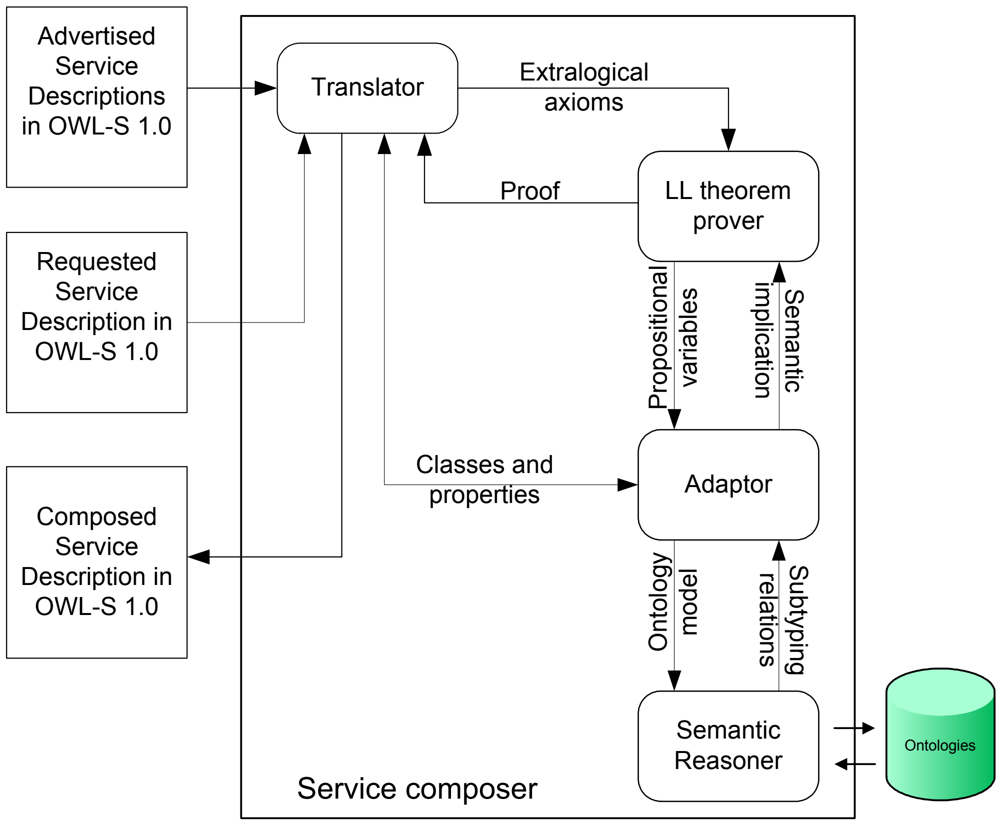


**Figure 3.10 The design of the service composer.**

### 3.9.2 LL representation of services and proof intuition

As said in the section covering the OWL-S ServiceModel (2.5.2.2) a service is either represented by an atomic or a composite process. The latter is included in a composite service description and is built upon combining a set of atomic processes (which is included in an atomic service description). Despite of the ServiceModel all services can have functional and non-functional requirement. Inputs, outputs and exceptions are considered to be functional. Non-functional (such as price and CPU load) on the other hand are classified into certain categories. Considering these facts this will bring us to the following LL formula for requesting a composite service:

$$\Gamma;\Delta_c \P (I \multimap (O \oplus E)) \otimes \Delta_r$$

The Γ in the formula represents the extralogical axioms, i.e. the advertised (atomic) services presented in LL (in the form $\Delta_c \P (I \multimap (O \oplus E)) \otimes \Delta_r$). The $\Delta_c$ represents a conjunction of non-functional constraints, and the $\Delta_r$ is a conjunction corresponding of non-functional results. The functionality of requested composite service is described by $(I \multimap (O \oplus E))$. $I$ represents the input parameters and $O$ the corresponding output parameters. $E$ is a representation of the exception thrown by the service. Intuitively one could say that the composite service is built by combining atomic services that together take the input set $I$ and generates output set $O$ (or $E$).

# 4 Implementation

The prototype is implemented using the JADE platform. Our intent was to implement two prototypes, the other based on the Agora platform. Since the Agora platform wasn't fully implemented, we have restricted our implementation to only support the JADE platform. In this chapter we will go through some details about the implementation of the prototype, and when doing that we will use the created Java packages as our basic structure. The implementation of the agent architecture has been divided into the five packages: agents, content, grid, matcher and storage. Except for the listed packages we will also look at the additional service data element added to the Grid services.

The prototype consists of 27 packages, 34 classes and 10 interfaces including the ones used for testing. The total size of the source classes is 109 Kbytes and they together contain 3812 lines of code (including comments and white spaces). In addition to this several classes of the owl-s-1.0.1 API has been rebuilt to suit our project. In order to test the prototype a Grid service providing mobile services was implemented. The implementation includes source code, a Service Data Element, and the obligatory *gwsdl* document.

## 4.1 Development platform

We used several software tools to ease the development of the prototype. Due to the fact that both JADE 3.1 [38] and GT 3.2 [39] were to be used and that they both support Java, it became a natural choice when considering implementation languages. The following version and edition was used: Java (TM) 2 SDK (Standard Edition) Version 1.4.2 [40]. The code of the prototype was written in Borland JBuilder X Enterprise [41]. As mentioned above we used the APIs of JADE 3.1 and GT 3.2. In addition to those libraries we used owl-s-1.0.1 [37] and Jdom 1.0 [42] (including their libraries).

When developing the Grid service we used Eclipse 3.0 [43] with the additional Globus Toolkit Plug-in for Eclipse 0.2.0 [44]. The reason why we used a different environment when developing Grid services was the nice features of the plug-in.

## 4.2 Agents

The *agents* package holds the agents implementations for the supported platforms. As mentioned above JADE is the only considered platform at the time, so the package includes a single package called *jade*. The jade package holds implementations of the agents based on the JADE platform, i.e. the *ServiceProvisionAgent* and the *ServiceSelectionAgent*.

### 4.2.1 ServiceProvisionAgent

The Service Provision Agent (SPA) is implemented by the class ServiceProvisionAgent which extends the obligatory JADE Agent class. Considering the Use case Provision of service (3.3.1) the SPA should, except for creation of the agent (which is handled by the platform), include functionality for assigning services and updating of a registry. In addition to this the ServiceProvisionAgent includes functionality for performing service matching (using the matcher package), based on incoming requests from a Service Selection Agent. As mentioned in the section describing JADE (2.3.6.1), the actions of each agent are based on behaviours. The behaviours needed to cover the functionality required of a SPA are listed in Table 4.1.

| Behaviour | Description |
|---|---|
| addService | Assigns a service to a SPA given a GSH. Using the WSDL document representing the service all the operations, not considered to be standard Grid service operations, are translated into service descriptions. The descriptions are then handed to the local storage. |
| addVO | Fetches the GSHs of all the services located in the registry. Each of the services is added to the SPA's storage using the addService behaviour. |
| ListenForReq | A cyclic behaviour that listens for incoming requests. If a message is received it will be parsed and the right action will be taken. |
| RegisterSPA | Registers itself at the Directory Facilitator. |
| SearchAndResponse | Searches the local storage for the requested service and sends the result back. |

**Table 4.1 Behaviours of the Service Provision Agent.**

**Assignment of a service**
In figure 4.1 is the method for assigning a new service to a SPA. The first thing it does is to add new addService behaviour. If the agent isn't already registered at the Directory Facilitator it will add a RegisterSPA behaviour and finally start listening for incoming requests, i.e. adding a ListenForReq behaviour.

```
public void addService(String GSH){

  addBehaviour(new addService(this, GSH));

  if (!isRegistered){
    isRegistered = true;
    addBehaviour(new RegisterSPA(this));
    addBehaviour(new ListenForReq(this));
  }
}
```

**Figure 4.1 Method for assigning a new service.**

### 4.2.2 ServiceSelectionAgent

The Service Selection Agent (SSA) is implemented by the class ServiceSelectionAgent and just like the ServiceProvisionAgent class it extends the JADE Agent class. Considering the Use case Selection of service 3.3.2) one can easily see that the class ServiceSelectionAgent must include functionality, allowing users to search for services. In similarity to the ServiceProvisionAgent, creation of the agent is supported by the platform, and won't therefore be covered. The behaviours used by the agent are listed in Table 4.2.

| Behaviour | Description |
|---|---|
| GetSPAs | Searches the Directory Facilitator for available SPAs. |
| Receive | A cyclic behaviour that listens for incoming result messages. Messages are parsed and if it contains search results it's stored in a result vector. The behaviour can be terminated by calling the *setDone* method. |
| SearchSPA | Searches a given SPA for the requested services, i.e. sending an ACL requested with a findService Action. |
| Timeout | A "waker" behaviour, i.e. a behaviour that sleeps for a while and then wakes up. Timeout wakes up after a given timeout and terminates the collection of search results and starts evaluating the results. |

**Table 4.2 Behaviours of the Service Selection Agent.**

**Search for service**

One interesting method is the search method (seen in Figure 4.2), i.e. the method initializing a search for a service. Every search is identified with a unique identification number which rendered by calling the *getID* method. Before starting a new search the vector responsible for temporary storage of the result is cleared. The first behaviour executed will be the GetSPAs, which fetches a list of available SPAs. Secondly a parallel behaviour, containing one or more SearchSPA behaviours, is executed. Each of the SearchSPA behaviours will send a request to one of the available SPAs. Finally a second parallel behaviour is executed including both a receive behaviour (collecting results) and a Timeout behaviour (terminating the search after the given timeout). The Timeout behaviour will also sort the collected results which will render the best suited service.

```
public void search(Object service, int maxResults, int timeout){
  String id = getID();
  SequentialBehaviour s = new SequentialBehaviour();
  ParallelBehaviour  p1 = new ParallelBehaviour();
  ParallelBehaviour  p2 = new ParallelBehaviour();
  Receive receive       = new Receive(this, id);

  results.removeAllElements();

  for (int i = 0; i < Math.min(CONCURRENTSEARCH, maxResults); i++)
    p1.addSubBehaviour(new SearchSPA(this, service, id));

  p2.addSubBehaviour(receive);
  p2.addSubBehaviour(new Timeout(this, timeout, receive));

  s.addSubBehaviour(new GetSPAs(this));
  s.addSubBehaviour(p1);
  s.addSubBehaviour(p2);

  addBehaviour(s);
}
```

**Figure 4.2 Method for search for a service.**

## 4.3 Content

The *content* package includes classes for managing the content carried in the ACL messages together with classes used when working with WSDL documents. The package has been divided into three sub packages, i.e. *lang*, *owls* and *wsdl*.

### 4.3.1 lang

The lang package includes several classes used when working with the content languages included in the ACL messages. The language used in the prototype is FIPA RDF expressed in XML. Because there are no restrictions about how to express FIPA RDF we have decided to have interfaces representing both the *Action* and the *Description* classes. The interfaces define methods for translating the class into a string representation along with method for setting and getting the different parameters. In the prototype we have classes implementing both of the interfaces for expressing FIPA RDF, in XML (seen in section 3.6).

The lang package also includes the interface *createObject* which defines methods for extracting the content from a given ACL message and returning the associated object, e.g. an Action. In the prototype we have included support for XML in the class *createObjectImpl*.

### 4.3.2 owls

Like the name reveals the *owls* package includes classes with functionality for managing the OWL-S ontology (2.4.2). The package consists of a *Writer* and a *Reader* interface and implementations of those interfaces supporting OWL-S version 1.0 [34]. The Writer interface defines a method for converting an ontology object into to a string. The reverse method, i.e. converting a string into an ontology object, is defined by the Reader interface. The interfaces allows for implementing various ontologies. The reader and writer implementations are closely connected to the OWL-S API, which has been rewritten to suit this project; mostly because it included uncompleted parts and didn't include methods vital to the project.

### 4.3.3 wsdl

The *wsdl* package contains an interface defining methods for translating WSDL documents into ontology-based service descriptions. In addition to this, it contains another interface, defining methods for creating new synthetic services descriptions. The former interface is used by a SPA when adding a new Grid service and the latter can be used by a SSA when creating a service description, holding the user's requirements. The prototype supports the OWL-S ontology language and is implemented using the rewritten OWL-S API mentioned above.

## 4.4 Grid

The classes communicating directly with Grid services are located in the package *grid.services*. The package contains methods for extracting Service Data elements given a GSH. This is done using method defined in the Globus Toolkit 3 API, *findServiceData*. The Service Data element in this project is defined with respect to OWL-S (more about this in section 4.6). Finally the package contains descriptions of the obligatory Grid service operations. These descriptions are used when deciding the method to advertise given a WSDL document— there is no point in advertising an obligatory Grid service operation.

## 4.5 Matcher

The next package called *matcher* includes classes for matching the requested service description, against the ones being advertised. Each of the supported ontologies must be represented by a class implementing the standard *ServiceMatcher* interface. Furthermore, a service description can only be compared to other service descriptions advertised in the same ontology. The Matcher implemented in the prototype will only support matching of service descriptions expressed in OWL-S and it's a realization of the algorithm described in section 3.7.

Matching of input and output parameter types, in the implemented matching algorithm, is based on the URI of the types. This means that providers and requestors of services can define their own types. The implemented matcher is rather strict and will only accept a precise one-to-one parameter mapping, i.e. the requested and the provided service must have an equal amount of parameters as well as an equal amount of each parameter type.

## 4.6 Storage

The classes responsible for storing the assigned services at a Service Provision Agent are found in the *storage* package. The only requirement is the usage of an obligatory interface called *Storage*. This means that the storage can be implemented using various technologies, e.g. using a database or a simple Vector (currently supported by the prototype). The Storage interface defines methods for adding, removing and listing all available services. The latter is used when calling the method for matching of services. When extending the prototype to support several ontologies one must be able to fetch all provisioned services for each ontology. One possibility is to have storage representing each ontology. Another approach is to store an identifier of the ontology along with the service description. This way the storage can decide to only return the service descriptions defined in a given ontology.

## 4.7 Grid service extension

When describing a service one might want to express non-technical properties, or technical ones not found in a WSDL document. One way to solve the problem is to define Service Data elements holding the additional properties. The ontology currently supported by the prototype OWL-S, includes such properties. Some are crucial in our algorithm for matching services, e.g. the data fields of the Service Category. In the proposed solution we created a Service Data element called *OwlsDataType* holding the necessary properties (seen in Figure 4.3). The element is defined as an XML Schema and contains the four strings *categoryName*, *taxonomy*, *value* and *code*. To keep the flexibility the Service Data Element is defined in a separate file called *OwlsSDE.xsd.*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="OwlsData"
        targetNamespace=
        "http://globus.org/master/thesis/service/OwlsService/OwlsSDE"
        xmlns:tns=
        "http://globus.org/master/thesis/service/OwlsService/OwlsSDE"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

<wsdl:types>
<schema targetNamespace=
"http://globus.org/master/thesis/service/OwlsService/OwlsSDE"
attributeFormDefault="qualified" elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema">

        <complexType name="OwlsDataType">
                <sequence>
                        <element name="categoryName" type="string"/>
                        <element name="taxonomy" type="string"/>
                        <element name="value" type="string"/>
                        <element name="code" type="string"/>
                </sequence>
        </complexType>

</schema>
</wsdl:types>
</wsdl:definitions>
```

**Figure 4.3 The Service Data Type OwlsDataType.**

The Service Data type can then be imported into any Grid service description using *import* seen in Figure 4.4.

```
<import location="OwlsSDE.xsd" namespace=
"http://globus.org/master/thesis/service/OwlsService/OwlsSDE"/>
```

**Figure 4.4 Importing the OWL-S Service Data type.**

Then one must create the element inside the Service Data namespace (sd) and set the properties of the new element (as seen in Figure 4.5). The OwlsDataType in Figure 4.5 has the following properties: it must occurs once and only once; its values cannot be changed over time (mutability); the values cannot be modified; and its value cannot be nil (nillable). Service Data elements are declared inside the *portType* element.

```
<sd:serviceData name="OwlsData" type="data:OwlsDataType"
        minOccurs="1" maxOccurs="1" mutability="mutable"
        modifiable="false" nillable="false">
</sd:serviceData>
```

**Figure 4.5 Instatiating a OWL-S Service Data element.**

Due to the fact that the values of the SDE cannot be changed they must be set inside the Grid service when it's instantiated, i.e. inside the *postCreate* method shown in Figure 4.6. The first thing after calling the constructor of the super class is to create a SDE. Then we instantiate the class *OwlsDataType*, that represents the OWL-S Service Data type. After the values of the OwlsDataType object has been set, we simply add them to our SDE. At this point the SPAs can access the SDE using the Grid Service Handle of the requested service along with methods for finding Service Data in GT3.

```
public void postCreate(GridContext context)
        throws GridServiceException {

        // Call super class's postCreate
        super.postCreate(context);

        // Create a Owls Service Data element
        OwlsSDE = this.getServiceDataSet().create("OwlsData");

        // Create an OwlsDataType instance
        owlsDataValue = new OwlsDataType ();

        owlsDataValue.setCategoryName("currency");
        owlsDataValue.setCode("LESS THAN");
        owlsDataValue.setTaxonomy("sek");
        owlsDataValue.setValue("10.5");

        // Sets the values of the Service Data OwlsDataType values
        OwlsSDE.setValue(owlsDataValue);

        // Add the Service Data Element to the Service Data set.
        this.getServiceDataSet().add(OwlsSDE);
}
```

**Figure 4.6 The initialization method, optional for GT3 Grid services.**

## 4.8 User manual

The prototype runs on top of the JADE platform. So in order to run the agents a JADE container must be initialized. The class path of the container will also be the one used when invoking the agents. Therefore all the library dependencies of the agents must be included when bringing up the agent container, e.g. *GT 3.2* libraries, *JADE 3.1*, *Jdom 1.0*, *owl-s-1.0.1* (rebuild), the libraries of *owl-s-1.0.1*, our *OwlsDataType*, and finally the classes of the master thesis. After the container has been brought up agents can be invoked using the container GUI. When a Service Provision Agent is invoked a simple GUI is presented, with functionality for assigning a simple service or a whole registry. At this point there is no GUI for the Service Selection Agent. Services can be created by calling the *NewServiceImpl.createService()* method. A search is initiated by calling the search method with the defined parameters (including a service description). A more detailed description can be found in Appendix B.

# 5 Validation

This chapter will cover the most important parts of the validation of the implemented prototype. One can say that validation means testing whether the implementation fulfils the proposed design. The first two sections will validate minor (nevertheless important) parts of the system, i.e. generating a Grid service ontology description and assignment of a Registry. The final section will be validating the whole system when realizing a detailed scenario of system usage (described in section 3.2.3).

## 5.1 Generating a Grid service ontology description

In order to test whether the generator of Grid service descriptions worked satisfactory, a sample Grid service was implemented (or at least parts of it). The sample Grid service defines four simple operations (seen in Table 5.1), where each operation is supposed to implement a service for mobile phones. One of the main reasons why Grid services are well-suited for these kinds of services is the ability to attach additional Service Data, which in this case is based on taxonomy. The functionality of the operations isn't really implemented. Hence, this is only a test considering generation of Grid service descriptions and invocation of the service won't be needed.

| Method name | Input | Output | Intended functionality |
|---|---|---|---|
| sendSMS | String message, long number | boolean status | To send a SMS with the message as content to the number given by the long. The method returns true if the method succeeds. |
| sendMMS | String message, long number | boolean status | To send the given MMS message to the number held by the long. The method returns true if the method succeeds. |
| sendRingTone | int tone, long number | boolean status | The intended functionality is to send the ring tone represented by the given int to the given number. The method returns true if the method succeeds. |
| sendPicture | int picture, long number | boolean status | The intent is to send the picture associated with the given int to the given number. The method returns true if the method succeeds. |

**Table 5.1 The methods implemented in the sample Grid service.**

In addition to the listed operations the Grid service imported the OwlsDataType seen in section 4.6. categoryName where set to *currency*, code to *LESSTHAN*, taxonomy to *sek* and the value where set to *1.3*. These values where assigned to each of the ontology descriptions.

When deploying the Grid service to the application server, a WSDL document representing the service is generated (see Figure 5.1). When running our test program *test.TestProgram* with the GSH representing the sample Grid service it generated four ontology descriptions (one for each operation). The TestProgram writes the ontology descriptions represented in RDF/XML into a text document. The generated service descriptions can be found in Appendix D. A visual description of the validation test can be seen in Figure 5.1.

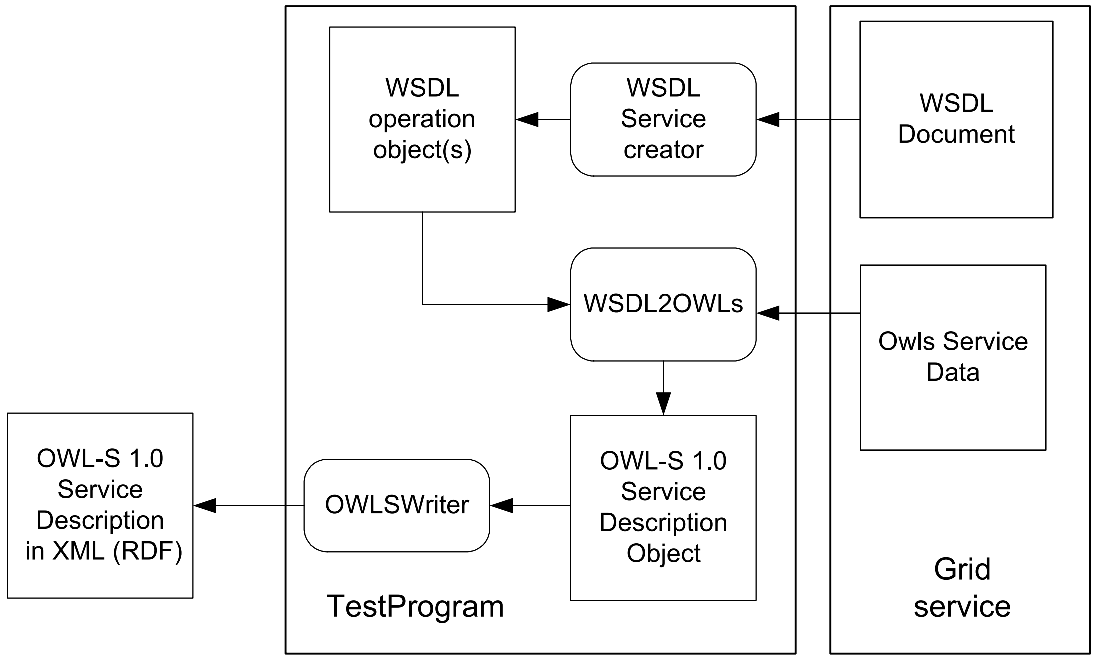


**Figure 5.1 Validation of Grid service description generation.**

In order to validate the generated services each one of them where syntactically checked using the OWL-S Validator [45] — a web-based tool for validation of OWL-S documents. All of the considered OWL-S service descriptions passed the syntactic checker. The validation results can be seen in Appendix E.

## 5.2 Assignment of a Registry

In order to test the functionality of the VORegistry ten different GSHs where assigned to it. The assignment where implemented in the *test.TestProgram*. Then a Service Provision Agent where instantiated and given the directions to extract the services from the registry (defined as findServiceData in Figure 5.2). The interaction between the different parts of the system (in validation test) can be seen in the collaboration diagram below, i.e. Figure 5.2.

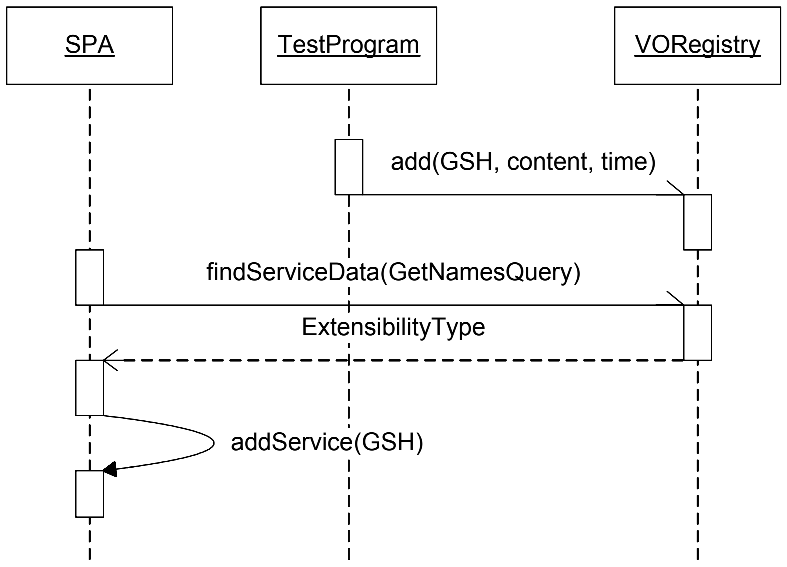


**Figure 5.2 Validation of the interaction with the registry.**

The SPA prints the extracted addresses before trying to assign them. The printout can be seen in Figure 5.3.

```
Registry Entry 0: http://www.globus.org/some/grid/service01
Registry Entry 1: http://www.globus.org/some/grid/service02
Registry Entry 2: http://www.globus.org/some/grid/service03
Registry Entry 3: http://www.globus.org/some/grid/service04
Registry Entry 4: http://www.globus.org/some/grid/service05
Registry Entry 5: http://www.globus.org/some/grid/service06
Registry Entry 6: http://www.globus.org/some/grid/service07
Registry Entry 7: http://www.globus.org/some/grid/service08
Registry Entry 8: http://www.globus.org/some/grid/service09
Registry Entry 9: http://www.globus.org/some/grid/service10
```
**Figure 5.3 Prinout of a SPA extracing service from a VORegistry.**

These are also the addresses added to the registry by the test program. Considering this simple example one can easily see that the interaction with the registry fulfils our requirements.

## 5.3 Realizing a detailed scenario of system usage

The last validation test is the most complex one and it will cover both provision and selection of services. The test will be an attempt to realize the detailed scenario of system usage described in section 3.2.3. That scenario describes the business man Lex who tries to take advantage of large companies providing services for mobile phone users. The idea is to provide and select the best suited services using an agent-based system for Grid service provision and selection.

In our realization of the scenario there are four companies providing services and each one of them is represented by a unique Service Provision Agent (as seen in Figure 5.4). The SPAs registers at the Directory Facilitator after its sample services (using a method in the TestProgram) has been stored locally. The service requested by the SSA, or at least a representation of that service, is fetched from the test program as well. The services provisioned by each agent are listed in Table 5.2.

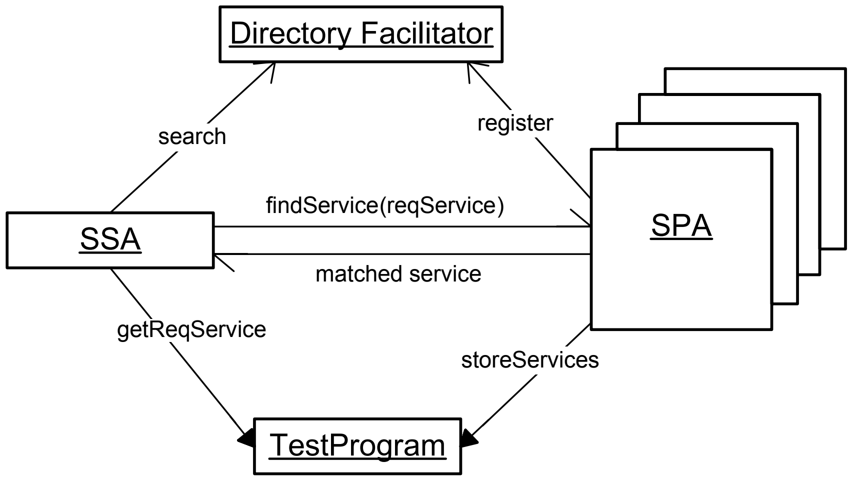


**Figure 5.4 The validation of the detailed scenario of system usage.**

| Agent | Operations | Service Category |
|---|---|---|
| spa1 | **sendSMS**(message:String, number:long) -> (status:boolean)<br>**sendMMS**(message:MMS, number:long) -> (status:boolean)<br>**sendRingTone**(tone:int, number:long) -> (status:boolean)<br>**sendPicture**(picture:int, number:long) -> (status:boolean) | **catogoryName:** currency<br>**code:** LESSTHAN<br>**taxonomy:** sek<br>**value:** 1.3 |
| spa2 | **SMSSender**(message:String, number:int) -> void<br>**MMSSender**(message:MMS, number:int) -> void<br>**RingToneSender**(tone:int, number:int) -> void<br>**PictureSender**(pic:int, number:int) -> void | **catogoryName:** currency<br>**code:** LESSTHAN<br>**taxonomy:** sek<br>**value:** 1.4 |
| spa3 | **sendSMS**(message:String, number:long) -> (status:boolean)<br>**sendMMS**(message:MMS, number:long) -> (status:boolean)<br>**sendRingTone**(tone:int, number:long) -> (status:boolean)<br>**sendPicture**(picture:int, number:long) -> (status:boolean) | **catogoryName:** currency<br>**code:** LESSTHAN<br>**taxonomy:** sek<br>**value:** 1.6 |
| spa4 | **birthdaySMS**(message:string, number:long, time:dateTime) -> (status:boolean)<br>**birthdayMMS**(message:MMS, number:long, time:dateTime) -> (status:boolean)<br>**birthdayRingTone**(tone:int, number:long, time:dateTime) -> (status:boolean)<br>**birthdayPicture**(pic:int, number:long, time:dateTime) -> (status:boolean) | **catogoryName:** currency<br>**code:** LESSTHAN<br>**taxonomy:** sek<br>**value:** 1.3 |

**Table 5.2 The SPAs and their services in the detailed scenario of system usage.**

We will see three different scenarios of service selection. In the first scenario a service for sending SMS will be selected. The second scenario will focus on selection of a MMS service, and in the final scenario the target will be a service suitable for sending pictures at a given time.

The test results, i.e. the printouts of the executions, are presented in Appendix E. The results shows the requested service, the services obtained from each SPA and finally the service considered (by the SSA) to be the best match. Considering these simple scenarios the matching algorithm selects the most suitable service.

# 6 Evaluation

In this chapter the performance of the system will be tested. We will first look at the time consumed by different parts of the system. Then we will try to measure the memory usage of the most vital parts of the system. The evaluation is technique for developers to see where the system can be improved and to use the measurement to calculate how the system behaves in untested environments.

## 6.1 Test-bed platform

The computer running the tests was an AMD Athlon 1800+ with 512 MB in RAM. The operative system was Microsoft Windows XP Professional SP1.

To run the agent platform (and the agents) the following software were used: Java (TM) 2 SDK (Standard Edition Version 1.4.2) [40], Jade 3.1 agent platform [38], a modified version of owl-s-1.0.1 [37], Jdom 1.0 [42], and the GT 3.2 libraries [39].

The hosting environment used for GT3.2 where Apache Ant 1.6 [46] together with Tomcat 4.1 [47] (i.e. a servlet container). Furthermore we used JUnit 3.8.1 [48], HSQLDB 1.7.2 [49] and Python version 2.3.4 [50].

In addition to above mentioned tools we also used Borland® Optimizeit™ Enterprise Suite 6 [51] to run some fine-grained evaluation tests. Opimizeit is a tool for isolating and resolving performance of Java (J2EE) applications.

## 6.2 Performance based on time measurements

This section will concentrate on performance based on time measurement. Three different test scenarios will be run testing the agent architecture. The first will measure the time spent on different parts (e.g. methods or behaviors) of the system when dealing with provision of service. The second will focus on selection of service, and the final one will be a fine-grained evaluation of the matching algorithm.

Due to the fact that the standard function for system time in Java is rather imprecise, especially when ran under Windows, we created our own timer based on the undocumented class *sun.misc.Perf*. The class, included in Java SDK since version 1.4.2, allows for accessing the high performance timer of the CPU. Our timer is based on comparing the frequency with the clock ticks of the CPU.

The measurement will be based on printouts, e.g. printing the time in the beginning of a method and comparing it with a printout in the end. We also ran some additional tests, using Optimizeit, to resolve where the bottlenecks might be located. Otimizeit allows us to see the time consumed by different parts of the system, using methods as the level of granularity.

### 6.2.1 Provision of service

To provision a Grid service a Service Provision Agent creates service descriptions, given the Grid service's WSDL document and additional Service Data. To test the performance of a SPA provisioning a service, the example Grid service from the validation test was used (5.1). The service was provisioned ten times before the test values were recorded. The main reason for not having the first values recorded is the high latency of the web server when a Grid service is instantiated (and the Grid service host environment is outside the scope of this project). The values recorded from the test can be seen in Table 6.1 and Figure 6.1. As seen in the figure as well as the table it's the creation of the WSDL object that dominates the consumed time.

| The concerned system function | Start time (ms since test start) | Duration (ms) | Percentage of the total time consumption (%) |
|---|---|---|---|
| Add service | 0 | 1209 | 100.0 |
| Create WSDL | 2 | 1131 | 93.5 |
| Get Service Data | 1134 | 54 | 4.5 |
| Store service 1 | 1189 | 2 | 0.2 |
| Store service 2 | 1192 | 1 | <0.1 |
| Store service 3 | 1194 | 3 | 0.2 |
| Store service 4 | 1198 | 8 | 6.6 |

**Table 6.1 The time consumed by different parts of the system when provisioning a service.**

The bottom bar in Figure 6.1, i.e. *Add service*, represents the entire behavior of provisioning a Grid service. *Create service,* i.e. the part responsible for more than 90 % of total time consumption, parses the WSDL document of the given Grid service and translates it into a *WSDLService* object. Get Service Data simply fetches the service data of the Grid service.Given a WSDLService object and additional Service Data, *Store service* translates them into a single service description (in the preferable ontology), which is finally stored in the local storage (in the evaluation tests a simple Java Vector was used). As seen in the validation example (5.1) the Grid service will be represented by four ontology objects. This can also bee seen in Figure 6.1 where the Store service method is represented four times (one for each ontology object).

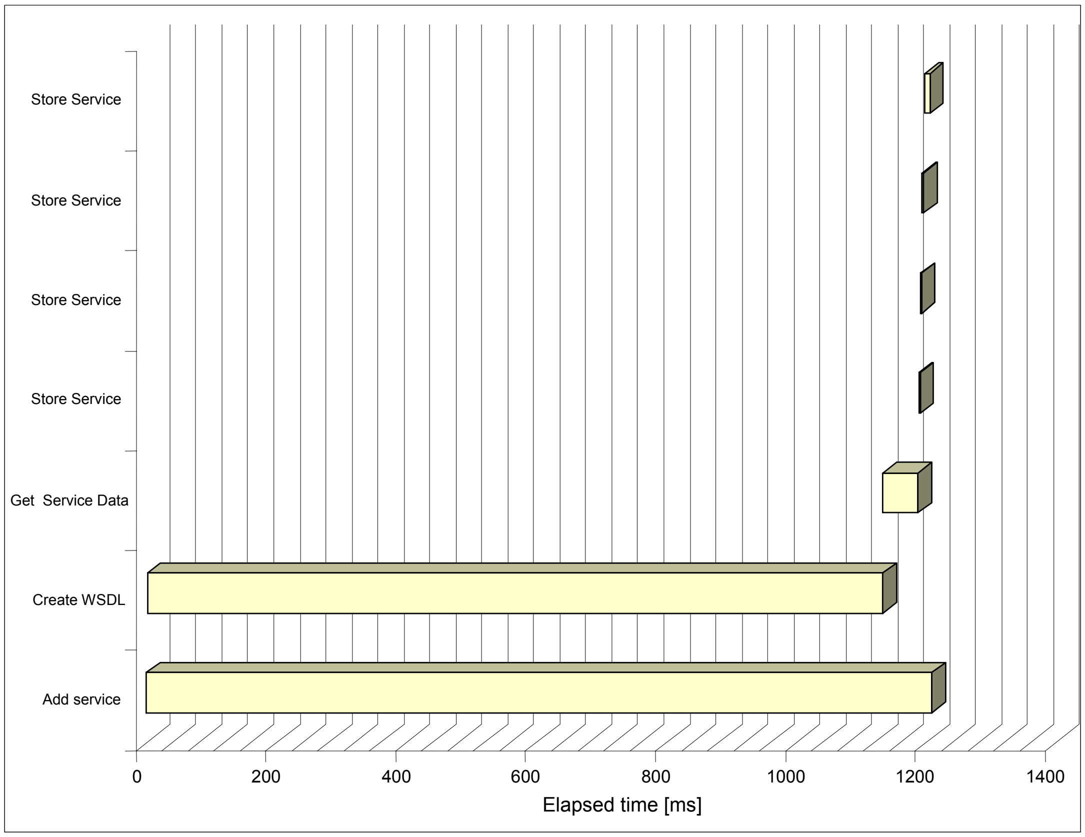


**Figure 6.1 The time consumed by different parts of the system when provisioning a service.**

In order to motive the large time consumption of create WDSL we did a supplementary evaluation test using Optimizeit. Our intent was to locate the methods responsible for the high execution costs. As expected the supplementary test also held the creation of WSDL objects responsible for being the most time consuming part. When traversing through the methods calls it's quite easy to see that our WSDL parser, i.e. *org.apache.axis.wsdl.gen.Parser*, is responsible for the large time consumption. Traversing further through the method calls we found that the parser builds a symbolic table, which was the single largest contribution to the high execution costs. If one would like the system to become more time efficient (regarding provision of services) it might be a good idea to replace the current parser with a new one, less time consuming.

### 6.2.2 Selection of service

In similarity to the evaluation test regarding selection of service (6.2.2) we used an example from Chapter 5. When evaluating selection of service we used the realization of the detailed scenario of system usage (5.3). We did time measurements on the Service Selection Agent as well as the four Service Provision Agents. The results can be seen in Table 6.2 and Figure 6.2 (as with provision of service the test was run ten times before any values were recorded). Each color in the diagram represents a unique agent.

| Agent | The concerned system function | Start time (ms since test start) | Duration (ms) | Percentage of the total time consumption (%) |
|---|---|---|---|---|
| ssa1 | search | 0 | 813 | 100.0 |
| | init search | 0 | 1 | 0.1 |
| | Get SPAs | 1 | 16 | 2.0 |
| | search SPA #1 | 17 | 16 | 2.0 |
| | search SPA #2 | 34 | 15 | 1.8 |
| | search SPA #3 | 67 | 15 | 1.88 |
| | search SPA #4 | 114 | 12 | 1.5 |
| | parsing msg | 165 | 136 | 16.7 |
| | parsing msg | 318 | 19 | 2.3 |
| | parsing msg | 338 | 24 | 3.0 |
| | parsing msg | 363 | 14 | 1.7 |
| | timeout | 809 | 4 | 0.5 |
| | sort results | 809 | 4 | 0.5 |
| spa1 | parsing msg | 29 | 4 | 0.5 |
| | Match and response | 33 | 114 | 14.0 |
| | Maching | 63 | 24 | 3.0 |
| spa2 | parsing msg | 49 | 2 | 0.2 |
| | Match and response | 111 | 54 | 6.6 |
| | Maching | 148 | 4 | 0.5 |
| spa3 | parsing msg | 82 | 13 | 1.6 |
| | Match and response | 169 | 125 | 15.4 |
| | Maching | 275 | 4 | 0.5 |
| spa4 | parsing msg | 128 | 2 | 0.2 |
| | Match and response | 171 | 77 | 9.5 |
| | Maching | 77 | 4 | 0.5 |

**Table 6.2 The time consumed by different part of the system when selecting a service.**

The bottom bar represents the entire search initiated by the SSA, i.e. all the behaviors and methods executed in order to achieve a search in the system. The reason why the search doesn't terminate when the last result message has been parsed is that it's based on a timeout (set to 500 ms in the example). When the SSA timeouts it sorts the incoming results and selects the best suited service. The reason why the first *parsing of message* consumes a larger amount of time than its followers could be the parallel execution of agents. We also think that this parallel execution has an impact on the time consumed by other agents, such as `spa 3` and `spa 4` seen in Figure 6.2. So the latency here will be a balance between of the timeout, the allowed concurrency and the (not so easily controlled) load of the host running the agent platform. The first two parameters are user defined while the last one dependent of the hosting environment.

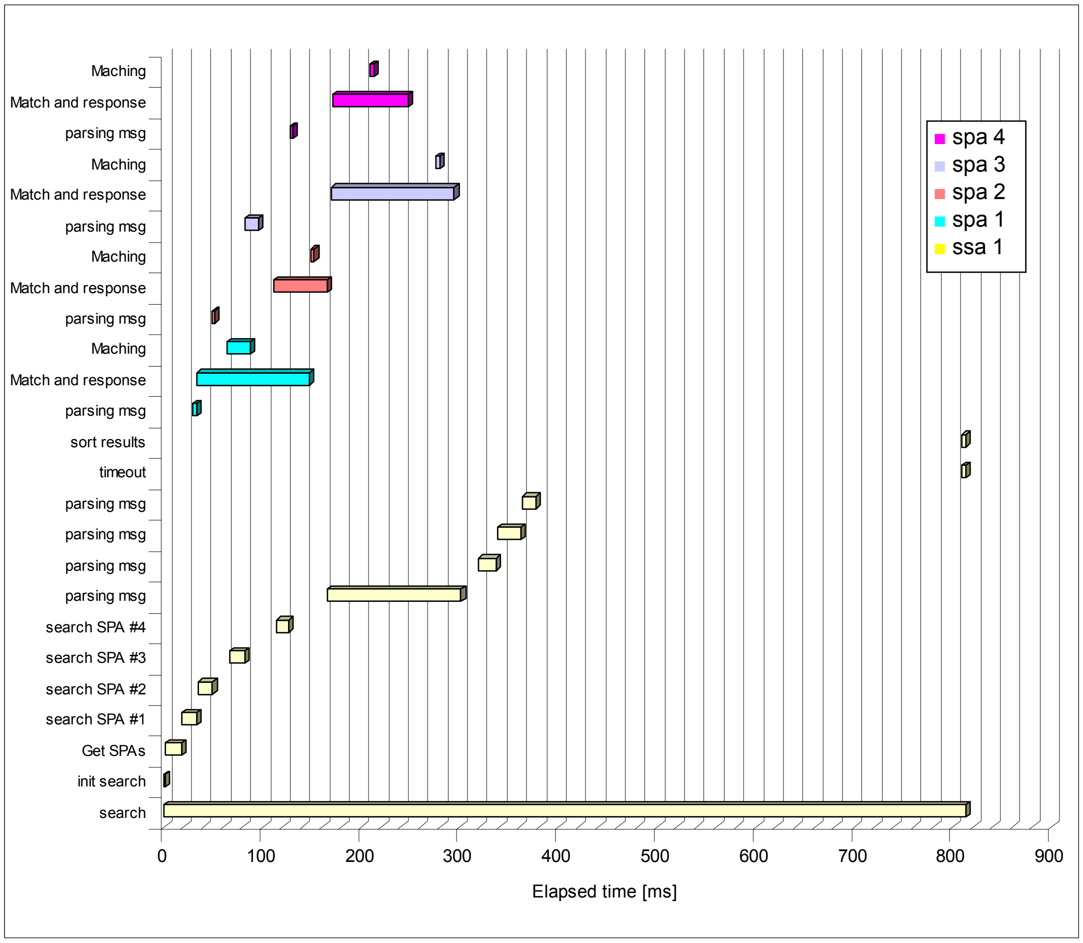


**Figure 6.2 The time consumed by different part of the system when selecting a service.**

As with the former evaluation test (covering provision of service) we used Optimizeit to locate the bottlenecks of the system use case (running the same test case). The results showed that the Jade platform itself consumes nearly 60 % of the overall CPU usage. Apart from the platform, reading service descriptions consumes nearly 22 %, and writing them about 18 %, of the overall CPU usage. So, if one would like to minimize the response time of a search one might start consider optimizing the ontology reader and writer implementations.

## 6.2.3 Matching algorithm

The final time-based evaluation tests, considering the matching algorithm, will not be run on top of the agent platform. Instead the tests were implemented in the class *test.TestProgram*.

Each time the algorithm tries to find the best suited service, it matches all the advertised services against the requested one. The first test concerns the scalability of the algorithm, i.e. how the algorithm would be affected as the number of advertised services increases. The measured affect is off course the time consumed to find the best match. The algorithm was tested with 5, 10, 20, 40, 60, 80, 100, and finally 120 synthetically generated services that were advertised. Each service had a random name, two random inputs, one random output, as well as a random value of the taxonomy. The test results can be seen in Figure 6.3. The tests were executed several times before values were recorded. Furthermore, each test was recorded ten times and an average calculated. As seen in the figure the time consumed by the algorithm scales linear with the number of advertised services.

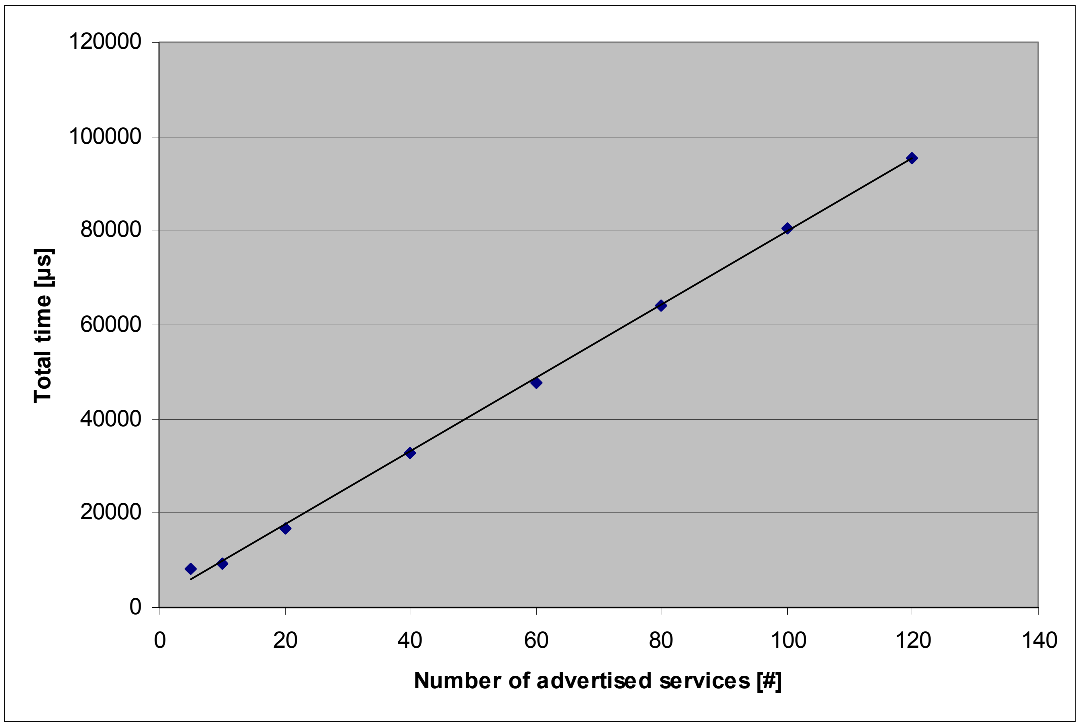


**Figure 6.3 The scalability of the matching algorithm.**

If there is a large number of services in the system one might want to improve the scalability of the algorithm. One of the most common ways to improve scalability is to add some kind of parallelism, e.g. the algorithm could spawn a new thread for each search criteria. Due to the fact that thread handling is quite inefficient in Java and that the solution requires the server to have multiple processors (to gain speedup) it might not be satisfying (especially when it comes to a marketplace with a large number of agents). Another solution could be to narrow the search down into several more detailed search criteria (like in a binary search), e.g. one could divide the advertised services into groups, dependent of the number of input (or output) parameters.

The next test focused on how the consumed time was balanced inside the algorithm. The test considered the four matching passes, where each of them was measured with respect to time consumption. The test was executed matching a service against 100 advertised ones. The test results can be seen in Figure 6.4. As can be seen in the figure, the matching of inputs consumes more time than corresponding matching of outputs. The reason for this lies within the fact that the requested as well as the advertised services has two input parameters, and only one output parameter. One could claim that the consumption of the input matching should be four times as high as the output matching (due to the fact that each requested parameter is compared to every advertised one). The reason why this isn't the case is quite simple; the matched parameters are no longer to be considered to be a match for the parameters left. The matching of names consumes nearly half as much as the matching of taxonomy, even though it only compares one field (the name) instead of four. This is a result of the two-way matching described in section 3.7.1.3.

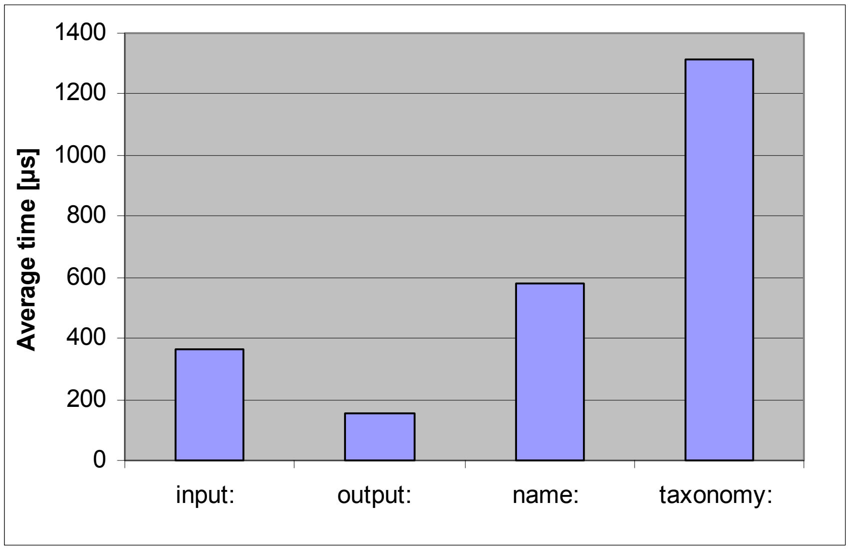


**Figure 6.4 The time consumed by the different matching passes.**

## 6.3 Memory usage

The aim of this section is to evaluate the memory usage of parts our agent-based system. The JADE platform with the required libraries but without any of our agents consumes about 14 Megabytes when ran on our test bed platform (this value is highly platform and hardware specific). It's said that the JADE platform is highly adjustable to its target host [52].

When running our memory usage test we used the Sizeof class described in the article [53]. The class uses widely used method of calculating used memory as:

*Runtime.totalMemory() - Runtime.freeMemory().*

The most interesting part of the class is its garbage collections method (preferable executed in advanced of the method calculating used memory). The intent is to stabilize the heap before measuring the amount of used memory.

### 6.3.1 Agents

If we want to achieve scalability it's important to allow a great number of simultaneous agents in the system. Therefore it's interesting to measure the memory usage of each invoked agent. It's not possible to set an upper limit hence it's dependent of the host environment. To get the average memory consumption the memory usage was measured before and after the invocation of several agents. The tests covered both the SPA and the SSA. The results can be seen in Table 6.3.

### 6.3.2 Service description

The memory usage of the service description is another object with a great impact on the scalability. Hence each service will be represented by a service description and because there will probably be a greater number of services descriptions than agents (hence SPAs not providing services are unnecessary). To measure the memory used to store a service description object we created an example service in OWL-S. The size was measured by instantiating the service object hundreds of times and to use this data to calculate an average memory usage. The values of can be seen in Table 6.3.

| Object | Memory usage (per instance) in Kbytes | Percentage of their combined sizes (%) |
|---|---|---|
| Service Selection Agent | 7.5 | 5.9 |
| Service Provision Agent | 20.4 | 16.0 |
| OWL-S Service description object | 99.3 | 78.0 |
| Total | 127.3 | 100.0 |

**Table 6.3 Memory usage.**

One can easily see that the number of provisioned services in the system (which will render service description objects) has the greatest impact on the memory consumption. This will also be the factor limiting the scalability of the system (with respect to memory usage).

## 6.4 Evaluation summary

The first evaluation tests considered the performance of the system related to time, i.e. the time consumption in different parts of the system. We constructed three different test-cases, where the first covered provision of service. Running the first test made it quite clear that our WSDL parser consumed over 90 % of the overall time. The second test, focusing on selection of services, held the Jade platform responsible for over 60 % of time consumption. Apart for the platform the conversion of ontology objects into XML documents, and vice versa, were the most time consuming parts (with about 18 % of the overall time consumption for writing and 22 % for reading XML documents). The third test, considering time consumption, was a fine-grained test of the matching algorithm. The algorithm consists of four different matching passes, where the matching of taxonomy was the most time consuming one (according to the test). We consider the following parts to be potential bottlenecks: the WSDL parser, the writer and the reader of XML documents, and finally the matching algorithm when it comes to large amounts of advertised services.

In addition to the time-based evaluation tests we also measured the memory usage of the system. Our measurements included both of the agents as well as the OWL-S service description, where the latter consumes 78 % of the total amount of memory (when combining the compared objects).

# 7 Conclusions

In this report we have described our approach to create *an agent-based system for Grid service provision and selection* and how to extend this system to support composition. The system is based on communicating agents which negotiate services on behalf of providers and requestors of services. Grid service can be provisioned one at a time or several services can be assigned using a registry. When a service is provisioned, i.e. assigned to a Service Provision Agent (SPA), it's translated into a service description in any suitable ontology (e.g. OWL-S). This translation is based on the WSDL document as well as additional Service Data Elements held by the Grid service. When a service requestor wants to initiate a search for services, its Service Selection Agent (SSA) creates a synthetic service description representing the requested service. Using the synthetic service the agent sends messages requesting search to be carried out at some (by the agent) selected SPAs. When a request for a search is received at an SPA it extracts all its advertised services (i.e. the provisioned ones) from it local storage and match them against the requested one, using a matching algorithm. The best matches (at most one service per SPA) are returned the SSA.

In order to test our ideas a prototype, realizing our design (except for composition), was implemented in Java [40] on top of Jade [38]. We ran several tests validating our prototype against the requirements defined in the design. The tests covered: validating generation of service descriptions using a sample WSDL document (along with Service Data Elements); assignment of a registry holding Grid Service Handles; and the final test considered the entire system, when realizing a detailed scenario of system usage. According these validation tests our prototype fulfils the requirement defined in the proposed design.

The system was also evaluated using the implemented prototype running on a test-bed (described in section 6.1). The first evaluation tests measured the time consumed by different parts of the system. The system was tested regarding provision of service, selection of service, and finally the matching algorithm were carefully tested. When considering provision of service the part of the system interacting with the grid container (in our case an Apache server) consumed more than 90 % of the total time. This could be a result of that the test-bed isn't powerful enough to run both the agent platform and the grid container. But when running a more fine-grained test one can easily see that the Apache WSDL parser was, without any doubt, the most time consuming element. The second test considered selection of services using the detailed scenario of system usage realized in our final validation test. The overall time consumed by this test is highly dependent of the timeout value (the time waiting for results) set be the SSA when it initiates a search. Apart for the timeout value one can see that both the parsing and the writing of XML-messages consume about 20 % of the time. This wasn't any surprise and won't be considered to be a major problem, due to the fact that parsing usually is time consuming and in our approach it's unavoidable. Furthermore, the tests showed that the Jade agent platform itself consumed most (about 60 %) of the execution time. The final evaluation test (based on time measurements) was a fine-grained evaluation of the implemented matching algorithm. This test covered both measurements of the time consumed by matching different parts of the service descriptions, as well as the increasing overall time consumption when adding advertised services to the local storage. In the former test case one can see that the matching of taxonomy consumes the largest amount time. This is a result of the four fields matching required when comparing two taxonomies. One can also see that the time spent on input and output parameter matching is a function of the number of parameters at the requested as well as the advertised services — increasing the number of parameters will increase the time consumed by matching them. The latter evaluation test of the matching algorithm was an attempt to measure its scalability with respect to time consumption. As one could guess the time consumption of the algorithm scaled linearly with respect to the number of advertised services. This could be improved by narrowing down the search criteria rather than adding parallelism to the algorithm. The reason why parallelism isn't well suited for the system lies within the facts, that thread handling is quite inefficient in Java, and that the system might handle a large number of agents concurrently.

Apart from measuring the time consumption of different parts of the system prototype it was tested regarding its memory usage. The results show that a service description consumes nearly 80 % of the total memory usage, which in this case is a service description, combined with instances of both of the agents. Due to the fact that our prototype stores its advertised service descriptions in a Java vector makes it highly dependent of the amount of available memory at the hosting machine. The prototype doesn't hold any constraint on how the local storage is implemented (as long as it implements the given interface). One solution when it comes to large amount of service descriptions would be to use data bases.

We have seen how the scalability issues regarding memory usage can be solved, but not the ones concerning time consumption. There are several ways to control the time consumed upon initiating a search for service. The easiest way is probably to set the timeout value. However, this is problematic because initiating an extensive search may render in loosing results arriving after the timeout. To solve the search scalability issues (regarding time consumption) one must adjust the timeout value as well as the maximum allowed search concurrency to suit the hosting environment.

The system presented in this thesis will hopefully offer great guidance for the future work, i.e. the overall project working towards a novel solution for *Agent-Enabled Logic-Based Web Services Selection and Composition* [1].

# 8 Future work

This master thesis project is the first step in a research project towards a novel solution for *Agent-Enabled Logic-Based Web Services Selection and Composition* [1]. In order to simplify things one could say that the future work of this project is to convert the presented system into the architecture described in [1]. One of the most crucial changes will be to replace Grid services with its predecessor Web services. This will probably be rather trivial, while most of the technologies used are adapted for Web services. However, there are some parts of the current system that might need some extra elaboration before implemented in the new architecture. For example the additional information stored in Service Data Elements must be presented using some other technology. Furthermore, the VORegistry won't be available when using Web services, a similar solution using an UDDI registry instead should be feasible.

In addition to converting the system to handle Web services, there is work left regarding the design as well as the implementation of the logic-based service composition algorithm. This includes translating Web service into Linear Logic as well as writing the theorem prover.

It's also crucial to investigate the security aspects of the system before taking it into use.
If the future architecture is implemented on top of Jade one can with rather small means enforce user-to-agent authentication as well as message integrity and confidentiality. The permissions granted for each users of the Jade platform can also be specified in a policy file. Except for the agent platform it might be necessary to have some security regarding interacting with the UDDI registry and the invocation of the provisioned Web services.

## 9 References

[1] M Matskin and V Vlassov, Agent-Enabled Logic-Based Web Services Selection and Composition. Research Project Proposal 2004.

[2] T. Bellwood et al., Universal description, discovery and integration specification (UDDI) 3.0. [Online], 2003. Available: http://uddi.org/pubs/uddi-v3.00-published-20020719.htm

[3] E. Christensen et al., Web services description language (WSDL) 1.1. [Online]. Available: http://www.w3.org/TR/wsdl/

[4] D. Box et al., Simple object access protocol (SOAP) 1.1, 2001. [Online]. Available: http://www.w3.org/TR/SOAP/

[5] L Moreau et. al., On the Use of Agents in a BioInformatics Grid. 2003.

[6] S. Wang et. al.,A Multi-Agent System Architecture for End-User Level Grid Monitoring Using Geographic Information Systems (MAGGIS): Architecture and Implementation. 2003.

[7] J. Kopena and W. Regli. DAMLJessKB: A tool for reasoning with the semantic web. October 28, 2002.

[8] I. Horrocks, F. van Harmelen, and P. Patel-Schneider. DAML+OIL.Technical report. [Online]. http://www.daml.org/2001/03/daml+oil-index.html, March 2001.

[9] Sandia National Laboratories. Java Expert System Shell. [Online]. http://herzberg.ca.sandia.gov/jess/

[10] Stefan Tang. Matching of Web Service specifications using DAML-S descriptions. March 18th, 2004.

[11] D. Martin, M. Burstein G. Denker et. al. DAML-S (and OWL-S) 0.9 Draft Release. [Online]. http://www.daml.org/services/daml-s/0.9/

[12] J.-Y. Girard. Linear Logic. Theoretical Computer Science, Vol. 50, pp. 1--102, 1987.

[13] J. Rao, P. Küngas, M. Matskin. Logic-based Web Services Composition: from Service Description to Process Model. 2004.

[14] J. Rao, P. Küngas, M. Matskin. Application of Linear Logic to Web Service Composition. 2003.

[15] I. Foster, C. Kesselman, J. Nick and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002.

[16] Object Management Group. CORBA™/IIOP™ Specification. [Online]. http://www.omg.org/technology/documents/formal/corba_iiop.htm

[17] Sun Microsystems. Java Remote Method Invocation. [Online]. http://java.sun.com/products/jdk/rmi/

[18] T. Bray et. al., Extensible Markup Language (XML) 1.0 (Second Edition). [Online]. http://www.w3.org/TR/2000/REC-xml-20001006

[19] World Wide Web Consortium. Hypertext Transfer Protocol [Online]. http://www.w3.org/Protocols/

[20] S.Tuecke et. al., Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum, June 2003.

[21] World Wide Web Consortium. Web Services Description Language (WSDL) Version 1.2. W3C Working Draft 3 March 2003, [Online]. http://www.w3.org/TR/2003/WDwsdl12-20030303

[22] L. Ferreira et. al. Introduction to Grid Computing with Globus. IBM Reedbooks, September 2003. [Online]. http://www.redbooks.ibm.com/

[23] T. Sandholm and J. Gawor. Globus Toolkit 3 Core – A Grid Service Container Framework. July 2003.
[24] M. Wooldrige and N. R. Jennings. Intelligent agents: theory and practice. 1995.
[25] Katia P. Sycara. Multiagent Systems. Publication of The American Association for Artificial Intelligence, Summer 1998.
[26] Foundation for Intelligent Physical Agents. FIPA 97 Specification, Part 2. [Online]. http://www.fipa.org/specs/fipa00018/OC00018A.html
[27] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification. [Online]. http://www.fipa.org/specs/fipa00008/
[28] X3T2 Ad Hoc Group. Knowledge Interchange Format Specification. [Online]. http://logic.stanford.edu/kif/specification.html
[29] World Wide Web Consortium. RDF/XML Syntax Specification (Revised). [Online]. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/
[30] Pierre-Antoine Champin. RDF Tutorial. April 5, 2001.
[31] Foundation for Intelligent Physical Agents. FIPA RDF Content Language Specification. [Online]. http://www.fipa.org/specs/fipa00011/
[32] S. A. Petersen, J. Rao and M. Matskin. AGORA Multi-agent Architecture for Implementing Virtual Enterprises. In Proceedings of the Norwegian Information Technology Conference (NIK'2003), Tapir, 2003.
[33] World Wide Web Consortium. OWL Web Ontology Language Overview. [Online]. http://www.w3.org/TR/owl-features/
[34] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. [Online]. http://www.daml.org/services/owl-s/1.0/
[35] Brian McBride. An Introduction to RDF and the Jena RDF API. [Online]. http://jena.sourceforge.net/tutorial/RDF_API/
[36] World Wide Web Consortium. RDQL - A Query Language for RDF. [Online]. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/
[37] Evren Sirin. OWL-S API. [Online]. http://www.mindswap.org/2004/owl-s/api/
[38] Telecom Italia Lab. Jade 3.1. [Online]. http://jade.tilab.com/
[39] The Globus Alliance. Globus Toolkit 3.2. [Online]. http://www-unix.globus.org/toolkit/
[40] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE). [Online]. http://java.sun.com/j2se/
[41] Borland. JBuilder X Enterprise. [Online]. http://www.borland.com/jbuilder/
[42] The JDOM ™ Project. Jdom 1.0. [Online]. http://jdom.org/
[43] Eclipse Foundation. Eclipse 3.0. [Online]. http://www.eclipse.org/
[44] B. Sotomayor, M. López and T. Sánchez. A Globus Toolkit Plug-in for Eclipse. [Online]. http://gt3ide.sourceforge.net
[45] Evren Sirin. OWL-S Validator. [Online]. http://www.mindswap.org/2004/owl-s/validator/
[46] Apache Software Foundation. Apache Ant 1.6. [Online]. http://ant.apache.org/
[47] Apache Software Foundation. Tomcat 4.1. [Online]. http://jakarta.apache.org/tomcat/index.html
[48] E. Gamma and K. Beck. JUnit 3.8.1. [Online]. http://www.junit.org/
[49] HSQLDB Developers Group. HSQLDB 1.7.2. [Online]. http://hsqldb.sourceforge.net/
[50] Python Software Foundation. Python 2.3.4. [Online]. http://www.python.org/
[51] Borland. Borland® Optimizeit™ Enterprise Suite 6. [Online] http://www.borland.com/optimizeit

[52] F. Bellifemine, G. Caire, A. Poggi and G. Rimassa. JADE A White Paper. September 2003.
[53] Vladimir Roubtsov. Java Tip 130: Do you know your data size? [Online]. http://www.javaworld.com/javaworld/javatips/jw-javatip130.html

## A. Abbreviations

| | |
|---|---|
| ACL | Agent Communication Language |
| API | Application Programming Interface |
| CORBA | Common Object Request Broker Architecture |
| DF | Directory Facilitator |
| FIPA | Foundation for Intelligent Physical Agents |
| GSH | Grid Service Handle |
| GSR | Grid Service Reference |
| GT3 | Globus Toolkit 3 |
| HTTP | Hypertext Transfer Protocol |
| JADE | Java Agent Development Environment |
| KQML | Knowledge Query and Manipulation Language |
| MAS | Multi-Agent Systems |
| MMS | Multimedia Messaging Service |
| OGSA | Open Grid Services Architecture |
| OGSI | Open Grid Services Infrastructure |
| OWL | Web Ontology Language |
| OWL-S | Ontology Web Language for Services |
| RDF | Resource Description Framework |
| RDFS | Resource Description Framework Schema |
| RPC | Remote Procedure Call |
| SDE | Service Data Element |
| SMS | Short Message Service |
| SOAP | Simple Object Access Protocol |
| SPA | Service Provision Agent |
| SSA | Service Selection Agent |
| UDDI | Universal Discovery Description and Integration |
| URI | Uniform Resource Identifier |
| VO | Virtual Organization |
| WSDL | Web Services Description Language |
| XML | eXtensible Markup Language |

## B. Prototype manual

This appendix is supposed to guide users attempting to make use of the prototype system. First is a short description on how to bring up a Jade agent container, which is necessary in order to run the prototype system's agents. We will also give a brief description covering how providers and requestors can interact with the system.

### The agent platform

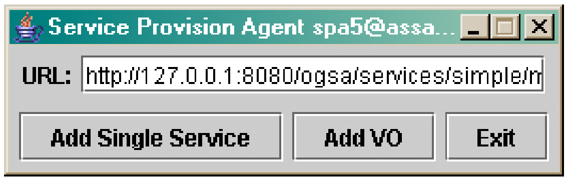
In order to run the agents one must set up the agent platform. In Jade this means bringing up a main container. After the main container is brought up several other containers can be connected to it extending the platform. We used the following command to bring up a main container:

```
java -classpath <classpath> jade.Boot -gui
```

The classpath (represented by `<classpath>`) must include the agent classes along with their class dependencies. The example container is brought up using the `-gui` option, which allows us to interact with the agent platform using a GUI. The GUI can be used to invoke as well as terminate agents.

### Service provider

Using the system for provision of services one must at least invoke one agents.jade.SPA.*ServiceProvisionAgent*. Our suggestion is to use the Jade GUI when invoking agents (which is covered in the Jade documentation [38]). When a ServiceProvisionAgent is invoked it automatically presents a simple GUI (seen in Appendix figure 1).


Service Provision Agent spa5@assa...
URL:
http://127.0.0.1:8080/ogsa/services/simple/n
Add Single Service
Add VO
Exit

**Appendix figure 1 The ServiceProvsionAgent GUI.**

The GUI allows for assignment of a single service as well as a VORegistry.

### Service Requestor

If one would like to use the system for selection of services instead, one should invoke an agents.jade.SSA.ServiceSelectionAgent. In similarity to the above mentioned ServiceProvisionAgent, agent invocation can be managed using the Jade GUI. But the agents differ when it comes to their own GUIs; hence the ServiceSelectionAgent doesn't provide one. Instead a WSDL-based service can be synthetically created using the *NewServiceImpl.createService()* method. Then the service can be translated into an OWL-S ontology object using the *WSDL2OWLs.createService()* method. To initiate a search for a service one simply call the *search()* method with the requested ontology object (along with the parameters *timeout* and *maxResults*). These methods are described further in the prototype documentation (see Appendix F).

## C. Generated service descriptions in OWL-S

### sendPicture

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:grounding="http://www.daml.org/services/owl-
                     s/1.0/Grounding.owl#"
    xmlns:profile="http://www.daml.org/services/owl-s/1.0/Profile.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:process="http://www.daml.org/services/owl-s/1.0/Process.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:service="http://www.daml.org/services/owl-s/1.0/Service.owl#">
  <process:ProcessModel rdf:ID="ProcessModel">
    <process:hasProcess>
      <process:AtomicProcess rdf:ID="Process">
        <process:hasOutput>
          <process:Output rdf:ID="out0">
            <rdfs:label>value</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#boolean"/>
          </process:Output>
        </process:hasOutput>
        <process:hasInput>
          <process:Input rdf:ID="in1">
            <rdfs:label>arg2</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#long"/>
          </process:Input>
        </process:hasInput>
        <process:hasInput>
          <process:Input rdf:ID="in0">
            <rdfs:label>arg1</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#int"/>
          </process:Input>
        </process:hasInput>
      </process:AtomicProcess>
    </process:hasProcess>
    <service:describes>
      <service:Service rdf:ID="Service">
        <service:presents>
          <profile:Profile rdf:ID="Profile">
            <profile:ServiceCategory rdf:parseType="Resource">
              <profile:value>1.3</profile:value>
              <profile:categoryName>currency</profile:categoryName>
              <profile:taxonomy>sek</profile:taxonomy>
              <profile:code>LESSTHAN</profile:code>
            </profile:ServiceCategory>
            <profile:hasOutput rdf:resource="#out0"/>
            <profile:hasInput rdf:resource="#in0"/>
            <profile:hasInput rdf:resource="#in1"/>
            <rdfs:label>sendPicture</rdfs:label>
            <profile:serviceName>sendPicture</profile:serviceName>
            <profile:textDescription>
               sendPicture(arg1:int, arg2:long) -> (value:boolean)
            </profile:textDescription>
            <service:presentedBy rdf:resource="#Service"/>
          </profile:Profile>
        </service:presents>
        <service:supports>
          <grounding:WsdlGrounding rdf:ID="Grounding">
            <service:supportedBy rdf:resource="#Service"/>
            <grounding:hasAtomicProcessGrounding>
              <grounding:WsdlAtomicProcessGrounding rdf:ID=
               "AtomicProcessGrounding">
                <grounding:wsdlOperation>
                  <grounding:WsdlOperationRef>

               <grounding:portType>OwlsServicePort</grounding:portType>

                  <grounding:operation>sendPicture</grounding:operation>
                  </grounding:WsdlOperationRef>
                </grounding:wsdlOperation>

<grounding:wsdlOutputMessage>sendPictureOutputMessage</grounding:wsdlOut
putMessage>
                <grounding:owlsProcess rdf:resource="#Process"/>
                <grounding:wsdlOutputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>

<grounding:wsdlMessagePart>value</grounding:wsdlMessagePart>
                    <grounding:owlsParameter rdf:resource="#out0"/>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlOutputMessageParts>
                <grounding:wsdlInputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#in0"/>

<grounding:wsdlMessagePart>arg1</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                  <grounding:wsdlMessageMap>

<grounding:wsdlMessagePart>arg2</grounding:wsdlMessagePart>
                    <grounding:owlsParameter rdf:resource="#in1"/>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlInputMessageParts>

<grounding:wsdlInputMessage>sendPictureInputMessage</grounding:wsdlInput
Message>

<grounding:wsdlDocument>http://127.0.0.1:8080/ogsa/services/simple/math/
service?WSDL</grounding:wsdlDocument>
              </grounding:WsdlAtomicProcessGrounding>
            </grounding:hasAtomicProcessGrounding>
          </grounding:WsdlGrounding>
        </service:supports>
        <service:describedBy rdf:resource="#ProcessModel"/>
      </service:Service>
    </service:describes>
  </process:ProcessModel>
</rdf:RDF>
```

**sendSMS**

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:grounding="http://www.daml.org/services/owl-
                      s/1.0/Grounding.owl#"
    xmlns:profile="http://www.daml.org/services/owl-s/1.0/Profile.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:process="http://www.daml.org/services/owl-s/1.0/Process.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:service="http://www.daml.org/services/owl-s/1.0/Service.owl#">
  <process:ProcessModel rdf:ID="ProcessModel">
    <process:hasProcess>
      <process:AtomicProcess rdf:ID="Process">
        <process:hasOutput>
          <process:Output rdf:ID="out0">
            <rdfs:label>value</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#boolean"/>
          </process:Output>
        </process:hasOutput>
        <process:hasInput>
          <process:Input rdf:ID="in1">
            <rdfs:label>arg2</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#long"/>
          </process:Input>
        </process:hasInput>
        <process:hasInput>
          <process:Input rdf:ID="in0">
            <rdfs:label>arg1</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#string"/>
          </process:Input>
        </process:hasInput>
      </process:AtomicProcess>
    </process:hasProcess>
    <service:describes>
      <service:Service rdf:ID="Service">
        <service:presents>
          <profile:Profile rdf:ID="Profile">
            <profile:hasOutput rdf:resource="#out0"/>
            <profile:hasInput rdf:resource="#in0"/>
            <profile:hasInput rdf:resource="#in1"/>
            <profile:ServiceCategory rdf:parseType="Resource">
              <profile:value>1.3</profile:value>
              <profile:code>LESSTHAN</profile:code>
              <profile:taxonomy>sek</profile:taxonomy>
              <profile:categoryName>currency</profile:categoryName>
            </profile:ServiceCategory>
            <rdfs:label>sendSMS</rdfs:label>
            <profile:serviceName>sendSMS</profile:serviceName>
            <service:presentedBy rdf:resource="#Service"/>
            <profile:textDescription>
               sendSMS(arg1:string, arg2:long) -> (value:boolean)
            </profile:textDescription>
          </profile:Profile>
        </service:presents>
        <service:supports>
          <grounding:WsdlGrounding rdf:ID="Grounding">
            <service:supportedBy rdf:resource="#Service"/>
            <grounding:hasAtomicProcessGrounding>
              <grounding:WsdlAtomicProcessGrounding
rdf:ID="AtomicProcessGrounding">

<grounding:wsdlOutputMessage>sendSMSOutputMessage</grounding:wsdlOutputM
essage>
                <grounding:wsdlOutputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#out0"/>

<grounding:wsdlMessagePart>value</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlOutputMessageParts>
                <grounding:owlsProcess rdf:resource="#Process"/>
                <grounding:wsdlInputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>

<grounding:wsdlMessagePart>arg1</grounding:wsdlMessagePart>
                    <grounding:owlsParameter rdf:resource="#in0"/>
                  </grounding:wsdlMessageMap>
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#in1"/>

<grounding:wsdlMessagePart>arg2</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlInputMessageParts>
                <grounding:wsdlOperation>
                  <grounding:WsdlOperationRef>

<grounding:portType>OwlsServicePort</grounding:portType>
                    <grounding:operation>sendSMS</grounding:operation>
                  </grounding:WsdlOperationRef>
                </grounding:wsdlOperation>

<grounding:wsdlInputMessage>sendSMSInputMessage</grounding:wsdlInputMess
age>

<grounding:wsdlDocument>http://127.0.0.1:8080/ogsa/services/simple/math/
service?WSDL</grounding:wsdlDocument>
              </grounding:WsdlAtomicProcessGrounding>
            </grounding:hasAtomicProcessGrounding>
          </grounding:WsdlGrounding>
        </service:supports>
        <service:describedBy rdf:resource="#ProcessModel"/>
      </service:Service>
    </service:describes>
  </process:ProcessModel>
</rdf:RDF>
```

## sendRingTone

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:grounding="http://www.daml.org/services/owl-
s/1.0/Grounding.owl#"
    xmlns:profile="http://www.daml.org/services/owl-s/1.0/Profile.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:process="http://www.daml.org/services/owl-s/1.0/Process.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:service="http://www.daml.org/services/owl-s/1.0/Service.owl#">
  <process:ProcessModel rdf:ID="ProcessModel">
    <process:hasProcess>
      <process:AtomicProcess rdf:ID="Process">
        <process:hasOutput>
          <process:Output rdf:ID="out0">
            <rdfs:label>value</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#boolean"/>
          </process:Output>
        </process:hasOutput>
        <process:hasInput>
          <process:Input rdf:ID="in1">
            <rdfs:label>arg2</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#long"/>
          </process:Input>
        </process:hasInput>
        <process:hasInput>
          <process:Input rdf:ID="in0">
            <rdfs:label>arg1</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#int"/>
          </process:Input>
        </process:hasInput>
      </process:AtomicProcess>
    </process:hasProcess>
    <service:describes>
      <service:Service rdf:ID="Service">
        <service:presents>
          <profile:Profile rdf:ID="Profile">
            <profile:serviceName>sendRingTone</profile:serviceName>
            <profile:hasOutput rdf:resource="#out0"/>
            <rdfs:label>sendRingTone</rdfs:label>
            <profile:hasInput rdf:resource="#in0"/>
            <profile:hasInput rdf:resource="#in1"/>
            <profile:ServiceCategory rdf:parseType="Resource">
              <profile:taxonomy>sek</profile:taxonomy>
              <profile:code>LESSTHAN</profile:code>
              <profile:categoryName>currency</profile:categoryName>
              <profile:value>1.3</profile:value>
            </profile:ServiceCategory>
            <service:presentedBy rdf:resource="#Service"/>
            <profile:textDescription>
               sendRingTone(arg1:int, arg2:long) -> (value:boolean)
            </profile:textDescription>
          </profile:Profile>
        </service:presents>
        <service:supports>
          <grounding:WsdlGrounding rdf:ID="Grounding">
            <service:supportedBy rdf:resource="#Service"/>
            <grounding:hasAtomicProcessGrounding>
              <grounding:WsdlAtomicProcessGrounding
rdf:ID="AtomicProcessGrounding">

<grounding:wsdlInputMessage>sendRingToneInputMessage</grounding:wsdlInpu
tMessage>

<grounding:wsdlOutputMessage>sendRingToneOutputMessage</grounding:wsdlOu
tputMessage>
                <grounding:wsdlOperation>
                  <grounding:WsdlOperationRef>

<grounding:portType>OwlsServicePort</grounding:portType>

<grounding:operation>sendRingTone</grounding:operation>
                  </grounding:WsdlOperationRef>
                </grounding:wsdlOperation>
                <grounding:owlsProcess rdf:resource="#Process"/>
                <grounding:wsdlOutputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#out0"/>

<grounding:wsdlMessagePart>value</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlOutputMessageParts>

<grounding:wsdlDocument>http://127.0.0.1:8080/ogsa/services/simple/math/
service?WSDL</grounding:wsdlDocument>
                <grounding:wsdlInputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>

<grounding:wsdlMessagePart>arg1</grounding:wsdlMessagePart>
                    <grounding:owlsParameter rdf:resource="#in0"/>
                  </grounding:wsdlMessageMap>
                  <grounding:wsdlMessageMap>

<grounding:wsdlMessagePart>arg2</grounding:wsdlMessagePart>
                    <grounding:owlsParameter rdf:resource="#in1"/>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlInputMessageParts>
              </grounding:WsdlAtomicProcessGrounding>
            </grounding:hasAtomicProcessGrounding>
          </grounding:WsdlGrounding>
        </service:supports>
        <service:describedBy rdf:resource="#ProcessModel"/>
      </service:Service>
    </service:describes>
  </process:ProcessModel>
</rdf:RDF>
```

**sendMMS**

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:grounding="http://www.daml.org/services/owl-
                     s/1.0/Grounding.owl#"
    xmlns:profile="http://www.daml.org/services/owl-s/1.0/Profile.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:process="http://www.daml.org/services/owl-s/1.0/Process.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:service="http://www.daml.org/services/owl-s/1.0/Service.owl#">
  <process:ProcessModel rdf:ID="ProcessModel">
    <process:hasProcess>
      <process:AtomicProcess rdf:ID="Process">
        <process:hasOutput>
          <process:Output rdf:ID="out0">
            <rdfs:label>value</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#boolean"/>
          </process:Output>
        </process:hasOutput>
        <process:hasInput>
          <process:Input rdf:ID="in1">
            <rdfs:label>arg2</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#long"/>
          </process:Input>
        </process:hasInput>
        <process:hasInput>
          <process:Input rdf:ID="in0">
            <rdfs:label>arg1</rdfs:label>
            <process:parameterType rdf:resource=
               "http://www.w3.org/2001/XMLSchema#string"/>
          </process:Input>
        </process:hasInput>
      </process:AtomicProcess>
    </process:hasProcess>
    <service:describes>
      <service:Service rdf:ID="Service">
        <service:presents>
          <profile:Profile rdf:ID="Profile">
            <profile:hasOutput rdf:resource="#out0"/>
            <profile:hasInput rdf:resource="#in0"/>
            <profile:serviceName>sendMMS</profile:serviceName>
            <profile:hasInput rdf:resource="#in1"/>
            <profile:textDescription>
               sendMMS(arg1:string, arg2:long) -> (value:boolean)
            </profile:textDescription>
            <rdfs:label>sendMMS</rdfs:label>
            <service:presentedBy rdf:resource="#Service"/>
            <profile:ServiceCategory rdf:parseType="Resource">
              <profile:code>LESSTHAN</profile:code>
              <profile:value>1.3</profile:value>
              <profile:taxonomy>sek</profile:taxonomy>
              <profile:categoryName>currency</profile:categoryName>
            </profile:ServiceCategory>
          </profile:Profile>
        </service:presents>
        <service:supports>
          <grounding:WsdlGrounding rdf:ID="Grounding">
            <service:supportedBy rdf:resource="#Service"/>
            <grounding:hasAtomicProcessGrounding>
              <grounding:WsdlAtomicProcessGrounding
rdf:ID="AtomicProcessGrounding">
                <grounding:wsdlOperation>
                  <grounding:WsdlOperationRef>
                    <grounding:operation>sendMMS</grounding:operation>

<grounding:portType>OwlsServicePort</grounding:portType>
                  </grounding:WsdlOperationRef>
                </grounding:wsdlOperation>

<grounding:wsdlOutputMessage>sendMMSOutputMessage</grounding:wsdlOutputM
essage>
                <grounding:wsdlInputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#in0"/>

<grounding:wsdlMessagePart>arg1</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#in1"/>

<grounding:wsdlMessagePart>arg2</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlInputMessageParts>
                <grounding:owlsProcess rdf:resource="#Process"/>

<grounding:wsdlInputMessage>sendMMSInputMessage</grounding:wsdlInputMess
age>
                <grounding:wsdlOutputMessageParts
rdf:parseType="Collection">
                  <grounding:wsdlMessageMap>
                    <grounding:owlsParameter rdf:resource="#out0"/>

<grounding:wsdlMessagePart>value</grounding:wsdlMessagePart>
                  </grounding:wsdlMessageMap>
                </grounding:wsdlOutputMessageParts>

<grounding:wsdlDocument>http://127.0.0.1:8080/ogsa/services/simple/math/
service?WSDL</grounding:wsdlDocument>
              </grounding:WsdlAtomicProcessGrounding>
            </grounding:hasAtomicProcessGrounding>
          </grounding:WsdlGrounding>
        </service:supports>
        <service:describedBy rdf:resource="#ProcessModel"/>
      </service:Service>
    </service:describes>
  </process:ProcessModel>
</rdf:RDF>
```

## D. OWL-S validation results

### Validation Results

**Number of services found:** 1
**Number of valid services:** 1

**Service:** http://www.it.kth.se/~it00_gni/sendSMS.xml#Service (Version: 1.0)
**Name:** sendSMS
**Description:** sendSMS(arg1:string, arg2:long) -> (value:boolean)

### Validation Results

**Number of services found:** 1
**Number of valid services:** 1

**Service:** http://www.it.kth.se/~it00_gni/sendMMS.xml#Service (Version: 1.0)
**Name:** sendMMS
**Description:** sendMMS(arg1:string, arg2:long) -> (value:boolean)

### Validation Results

**Number of services found:** 1
**Number of valid services:** 1

**Service:** http://www.it.kth.se/~it00_gni/sendRingTone.xml#Service (Version: 1.0)
**Name:** sendRingTone
**Description:** sendRingTone(arg1:int, arg2:long) -> (value:boolean)

### Validation Results

**Number of services found:** 1
**Number of valid services:** 1

**Service:** http://www.it.kth.se/~it00_gni/sendPicture.xml#Service (Version: 1.0)
**Name:** sendPicture
**Description:** sendPicture(arg1:int, arg2:long) -> (value:boolean)

## E. Execution printouts of validation of detailed scenario

### SMS service

```
ssa1: REQUIRED SERVICE:
ssa1:
ssa1: operation: SMS(msg:string, number:long) -> (status:boolean)
ssa1: Category name: currency
ssa1: code: LESSTHAN
ssa1: taxonomy: sek
ssa1: value: 1.5

ssa1: RESULT SERVICES:
ssa1:
ssa1: operation: sendSMS(message:string, number:long) ->
(status:boolean)
ssa1: Category name: currency
ssa1: code: LESSTHAN
ssa1: taxonomy: sek
ssa1: value: 1.6
ssa1:
ssa1: operation: birthdaySMS(message:string, number:long, time:dateTime)
->(status:boolean)
ssa1: Category name: currency
ssa1: code: LESSTHAN
ssa1: taxonomy: sek
ssa1: value: 1.2
ssa1:
ssa1: operation: SMSSender(message:string, number:int) -> ()
ssa1: Category name: currency
ssa1: code: LESSTHAN
ssa1: taxonomy: sek
ssa1: value: 1.4
ssa1:
ssa1: operation: sendSMS(message:string, number:long) ->
(status:boolean)
ssa1: Category name: currency
ssa1: code: LESSTHAN
ssa1: taxonomy: sek
ssa1: value: 1.3

ssa1: BEST MATCH:
ssa1:
ssa1: operation: sendSMS(message:string, number:long) ->
(status:boolean)
ssa1: Category name: currency
ssa1: code: LESSTHAN
ssa1: taxonomy: sek
ssa1: value: 1.6
```

**MMS service**

```
ssa2: REQUIRED SERVICE:
ssa2:
ssa2: operation: MMS(msg:MMS, number:int) -> (status:boolean)
ssa2: Category name: currency
ssa2: code: LESSTHAN
ssa2: taxonomy: sek
ssa2: value: 1.5

ssa2: RESULT SERVICES:
ssa2:
ssa2: operation: MMSSender(message:MMS, number:int) -> (status:boolean)
ssa2: Category name: currency
ssa2: code: LESSTHAN
ssa2: taxonomy: sek
ssa2: value: 1.4
ssa2:
ssa2: operation: sendMMS(message:MMS, number:long) -> (status:boolean)
ssa2: Category name: currency
ssa2: code: LESSTHAN
ssa2: taxonomy: sek
ssa2: value: 1.3
ssa2:
ssa2: operation: birthdayMMS(message:MMS, number:long, time:dateTime) ->
(status:boolean)
ssa2: Category name: currency
ssa2: code: LESSTHAN
ssa2: taxonomy: sek
ssa2: value: 1.2
ssa2:
ssa2: operation: sendMMS(message:MMS, number:long) -> (status:boolean)
ssa2: Category name: currency
ssa2: code: LESSTHAN
ssa2: taxonomy: sek
ssa2: value: 1.6

ssa2: BEST MATCH:
ssa2:
ssa2: operation: MMSSender(message:MMS, number:int) -> (status:boolean)
ssa2: Category name: currency
ssa2: code: LESSTHAN
ssa2: taxonomy: sek
ssa2: value: 1.4
```

**Picture on date**

```
ssa3: REQUIRED SERVICE:
ssa3:
ssa3: operation: Picture(pic:int, number:long, time:dateTime) ->
(status:boolean)
ssa3: Category name: currency
ssa3: code: LESSTHAN
ssa3: taxonomy: sek
ssa3: value: 2.0

ssa3: RESULT SERVICES:
ssa3:
ssa3: operation: sendPicture(picture:int, number:long) ->
(status:boolean)
ssa3: Category name: currency
ssa3: code: LESSTHAN
ssa3: taxonomy: sek
ssa3: value: 1.3
ssa3:
ssa3: operation: PictureSender(picture:int, number:int) ->
(status:boolean)
ssa3: Category name: currency
ssa3: code: LESSTHAN
ssa3: taxonomy: sek
ssa3: value: 1.4
ssa3:
ssa3: operation: sendPicture(picture:int, number:long) ->
(status:boolean)
ssa3: Category name: currency
ssa3: code: LESSTHAN
ssa3: taxonomy: sek
ssa3: value: 1.6
ssa3:
ssa3: operation: birthdayPicture(picture:int, number:long,
time:dateTime) -> (status:boolean)
ssa3: Category name: currency
ssa3: code: LESSTHAN
ssa3: taxonomy: sek
ssa3: value: 1.2

ssa3: BEST MATCH:
ssa3:
ssa3: operation: birthdayPicture(picture:int, number:long,
time:dateTime) -> (status:boolean)
ssa3: Category name: currency
ssa3: code: LESSTHAN
ssa3: taxonomy: sek
ssa3: value: 1.2
```

## F. Prototype documentation

### Package agents.jade.SPA

This package contains the classes specific for the Service Provision Agent based on the Jade platform.

### Class ServiceProvisionAgent

```
public class ServiceProvisionAgent extends Agent
```

Title: ServiceProvisionAgent
Description: An agent handeling the service providing part in a negotiation about services.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

#### Fields

**time**

```
publictime
```

#### Constructors

**ServiceProvisionAgent**

```
public ServiceProvisionAgent()
```

#### Methods

**setup**

```
protected void setup()
```

Is called by the platform when the agent is brought up. It initiates the translator, the storage, the matcher and the gui.

---

**match**

```
public java.lang.Object match(Object service)
```

Matches the given service against the ones located in storage.

**Parameters**
 **service -** the service to be matched.
**Returns**
 The best match.

---

**addService**

`public void addService(String GSH)`

Assigns the given service to the agent.
**Parameters**
 **GSH -** the GSH of the servie to be added.

---

**addVirtualOrganization**

`public void addVirtualOrganization(String GSH)`

Assigns all the services found in the registry to the agent.
**Parameters**
 **GSH -** the GSH of the registry to be searched.

---

**storeService**

`public void storeService(WSDLOperation service, Object[] data)`

Translates a WSDL service into the current ontology and stores it in the local storage.
**Parameters**
 **service -** the service to be translated and stored.
 **data -** additional service data used in translation.

---

**register**

`public void register()`

Registers the agent at the Directory Facilitator.

---

**print**

`public void print(String msg)`

Prints the given string to the standard output.
**Parameters**
 **the -** message to be printed.

---

**takeDown**

`protected void takeDown()`

takeDown is called before Agent termination and includes clean-up instructions.

## Package agents.jade.SPA.behaviours

This package contains the classes implementing behaviours for the Service Provision Agent based on the Jade platform.

### Class SearchAndResponse

```
public class SearchAndResponse extends OneShotBehaviour
```

Title: SearchAndResponse
Description: Searches the local storage for the requested service and and returns the result to the requestor.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

#### Constructors

**SearchAndResponse**

```
public SearchAndResponse(ServiceProvisionAgent spa, ACLMessage msg,
Action action)
```

Constructor of SearchAndResponse.
**Parameters**
**spa -** the agent holding the storage.
**msg -** the request message.
**action -** the reqeusted action.

#### Methods

**action**

```
public void action()
```

Searches the local storage and returns the results.

### Class RegisterSPA

```
public class RegisterSPA extends OneShotBehaviour
```

Title: RegisterSPA
Description: Register the invoking agent at the Directory Facilitator.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**

1.0

### Constructors

**RegisterSPA**

```
public RegisterSPA(ServiceProvisionAgent spa)
```

Constructor of RegisterSPA.
**Parameters**
**spa -** the SPA to be registered.

### Methods

**action**

```
public void action()
```

Registers the SPA.

## Class ListenForReq

```
public class ListenForReq extends CyclicBehaviour
```

Title: ListenForReq
Description: Listens for incoming search requests.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

**ListenForReq**

```
public ListenForReq(ServiceProvisionAgent spa)
```

Constructor of ListenForReq.
**Parameters**
**spa -** the initiating agent.

### Methods

**action**

```
public void action()
```

Listens for incoming messages. If it's a valid request the request is handed over to a SearchAndResponse behaviour.

## Class addVO

```
public class addVO extends OneShotBehaviour
```

Title: addVO
Description: Searches the given registry and assigns all the found services to the requesting agent.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

**addVO**

```
public addVO(ServiceProvisionAgent spa, String GSH)
```

Constructor of addVO.
**Parameters**
**spa -** the requesting agent.
**GSH -** the GSH of the registry to be searched.

### Methods

**action**

```
public void action()
```

Searches the registry and for each service found it calls the addService method at requesting agent.

## Class addService

```
public class addService extends OneShotBehaviour
```

Title: addService
Description: A behaviour adding the service represented by the given GSH.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

**addService**

```
public addService(ServiceProvisionAgent spa, String GSH)
```

Constructor of addService.
**Parameters**
  **spa -** the requesting agent.
  **GSH -** the GSH of the service to be added.

### Methods

**action**

```
public void action()
```

The method adding the service.

## Package agents.jade.SPA.gui

This package contains the classes implementing a simple GUI for the Service Provision Agent based on the Jade platform.

## Class GUI

```
public class GUI extends JFrame implements ActionListener
```

Title: GUI
Description: A simple GUI for the Service Provision Agent.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
  1.0

### Constructors

**GUI**

```
public GUI(ServiceProvisionAgent spa)
```

Constructor of GUI
**Parameters**
  **spa -** the agent represented by the GUI.

### Methods

**actionPerformed**

```
public void actionPerformed(ActionEvent e)
```

The method called when an action is performed. Could be to add a service, add a VO or to exit.

---

**showErrorMsg**

```
public void showErrorMsg(String msg)
```

Shows an err msg dialog.
**Parameters**
**msg -** the message to be shown.

## Package agents.jade.SSA

This package contains the classes specific for the Service Selection Agent based on the Jade platform.

## Class ServiceSelectionAgent

```
public class ServiceSelectionAgent extends Agent
```

Title: ServiceSelectionAgent
Description: An agent handeling the service requesting part in a negotiation about services.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

**ServiceSelectionAgent**

```
public ServiceSelectionAgent()
```

### Methods

**setup**

```
protected void setup()
```

Is called by the platform when the agent is brought up. It initiates e.g. the reader, the writer and the matcher.

---

**addResult**

```
public void addResult(String service)
```

Adds a search result to the result vector.
**Parameters**
**service -** the service description.

---

### getResults

`public java.lang.Object[] getResults()`

Return a array holding the result services.
**Returns**
The result services.

---

### sortResults

`public java.lang.Object sortResults(Object service)`

Sorts the results using the matching algorithm.
**Parameters**
**service -** the requested service.
**Returns**
The best match.

---

### printService

`public void printService(Object service)`

Prints the given service to the standard output.
**Parameters**
**service -** the service to be printed.

---

### printResults

`public void printResults()`

Prints the services in the result vector.

---

### updateSPAs

`public void updateSPAs(DFAgentDescription[] description)`

Updates the list of available SPAs.
**Parameters**
**description -** description of the requested SPA

---

### getNextSPA

`public synchronized DFAgentDescription getNextSPA()`

Returns the next SPA in the list of available SPAs.
**Returns**
The next SPA or null if empty.

---

**print**

```
public void print(String msg)
```

Prints the given string to the standard output.
**Parameters**
 **msg -** the message to be printed.

---

**getID**

```
public java.lang.String getID()
```

Returns a new search ID.
**Returns**
 The new id.

---

**search**

```
public void search(Object service, int maxResults, int timeout)
```

Initiates a search for services matching the given service.
**Parameters**
 **service -** the requested service.
 **maxResults -** maximum number of results.
 **timeout -** the timeout, which is the time waiting for result messages.

---

**takeDown**

```
protected void takeDown()
```

takeDown is called before Agent termination and includes clean-up instructions.

## Package agents.jade.SSA.behaviours

Begin with a one sentence summary about this package. Follow with the remainder of your description. Package Specification This package contains the classes implementing behaviours for the Service Selection Agent based on the Jade platform.

## Class Timeout

```
public class Timeout extends WakerBehaviour
```

Title: Timeout
Description: A behaviour that sleeps a given time and then terminates the behaviour receiving messages.
Copyright: Copyright (c) 2004

Company: IMIT/KTH
**Version**
1.0

## Constructors

#### Timeout

```
public Timeout(ServiceSelectionAgent ssa, int timeout, Receive
receive, Object service)
```

Constuctor of Timeout.
**Parameters**
**ssa -** the agent initiating the timeout.
**timeout -** the timeout in milli seconds.
**receive -** the receiving behaviour.
**service -** the requested service.

## Methods

#### handleElapsedTimeout

```
protected void handleElapsedTimeout()
```

The method executed after the timeout. It terminates the receiving behaviour and sorts the results.

## Class SearchSPA

```
public class SearchSPA extends OneShotBehaviour
```

Title: SearchSPA
Description: A behaviour that sends a search request to the SPA with the service description fetch form given agent.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

## Constructors

#### SearchSPA

```
public SearchSPA(ServiceSelectionAgent ssa, Object service, String
id)
```

Constructor of SearchSPA.
**Parameters**
**ssa -** the agent requesting the search.
**service -** the requested service.
**id -** the id of the search.

## Methods

**action**

```
public void action()
```

Fetches a SPA from the list of available SPAs and sends it a search request.

## Class Receive

```
public class Receive extends SimpleBehaviour
```

Title: Receive
Description: A behaviour initated by an agent to listen for search result messages util done is set to true.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

**Receive**

```
public Receive(ServiceSelectionAgent ssa, String id)
```

Constructor of Receive.
**Parameters**
**ssa -** the agent waiting for results.
**id -** the identification value of the search.

### Methods

**action**

```
public void action()
```

Listens for incoming result messages. If a result message is received it's stored at the agent.

---

**setDone**

```
public void setDone(boolean done)
```

Sets the flag that decides if the behaviour should continue to listen for messages.
**Parameters**
**done -** true to terminate the behaviour otherwise false.

---

#### done

```
public boolean done()
```

Is used by the platform to see if the behaviour is finnished.
**Returns**
True if the behaviour is done otherwise false.

## Class GetSPAs

```
public class GetSPAs extends OneShotBehaviour
```

Title: GetSPAs
Description: A behaviour called by an agent to get the list of available Service Provision Agents from the Directory Facilitator.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

#### GetSPAs

```
public GetSPAs(ServiceSelectionAgent ssa)
```

Constructor of GetSPAs.
**Parameters**
**ssa -** the agent requesting the list.

### Methods

#### action

```
public void action()
```

Searches the Directory Facilitator for available SPAs and returns the given results to the agent.

## Package agents.jade.ServiceDescriptions

This package contains classes focusing on agent service descriptions for the Jade platform.

## Class ServiceDescriptions

```
public class ServiceDescriptions
```

Title: ServiceDescriptions

Description: A class creating agent Service Descriptions.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

#### ServiceDescriptions

`public ServiceDescriptions()`

### Methods

#### getSPAOWLSDescription

`public static ServiceDescription getSPAOWLSDescription()`

Creates a Service Description of a Service Provision Agent supporting fipa-rdf0 and OWL-S 1.0.
**Returns**
A service description of a SPA.

## Package grid.services

This package contains classes and package associated with Grid services.

## Class GSOperation

`public class GSOperation`

Title: GSOperation
Description: A class containing the standard Grid service operation names.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Fields

#### DESTROY

`public static finalDESTROY`

---

#### FINDSERVICEDATA

`public static final`**`FINDSERVICEDATA`**

---

#### SETSERVICEDATA

`public static final`**`SETSERVICEDATA`**

---

#### REQUESTTERMINATIONAFTER

`public static final`**`REQUESTTERMINATIONAFTER`**

---

#### REQUESTTERMINATIONBEFORE

`public static final`**`REQUESTTERMINATIONBEFORE`**

### Constructors

#### GSOperation

`public` **`GSOperation`**`()`

## Package grid.services.serviceData

This package include classes and interfaces for managing Service Data.

## Interface GetServiceData

`public interface` **`GetServiceData`**

Title: GetServiceData
Description: An interface for resolving service data given a GSH.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
    1.0

### Methods

#### getServiceData

`public java.lang.Object[]` **`getServiceData`**`(String GSH)`

## Package grid.services.serviceData.impl

### Class OWLSData

```
public class OWLSData implements GetServiceData
```

Title: OWLSData
Description: An implementation of the GetSerivce Data interface that supports the OWLSDataType.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

#### Constructors

**OWLSData**

```
public OWLSData()
```

Constructor of OWLSData.

#### Methods

**getServiceData**

```
public java.lang.Object[] getServiceData(String GSH)
```

Returns an array of service data objects given a GSH.
**Parameters**
**GSH -** the GSH of the service.
**Returns**
The array of service data objects.

## Package content.owls

This package contains interfaces and packages for ontology-based service descriptions.

### Interface Writer

```
public interface Writer
```

Title: Writer
Description: An interface for writers of service descriptions.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Methods

**write**

```
public java.lang.String write(Object service)
```

---

**getOntology**

```
public java.lang.String getOntology()
```

## Interface Reader

```
public interface Reader
```

Title: Reader
Description: An interface for readers of service descriptions.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Methods

**read**

```
public java.lang.Object read(String service)
```

---

**getOntology**

```
public java.lang.String getOntology()
```

## Package content.owls.impl

This package holds implementation for managing OWL-S 1.0 ontology service descriptions.

## Class WriterImpl

```
public class WriterImpl implements Writer
```

Title: WriterImpl
Description: A class implementing the Writer interface. It support OWL-s 1.0.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

**WriterImpl**

```
public WriterImpl()
```

Constructor of WriterImpl.

### Methods

**getOntology**

```
public java.lang.String getOntology()
```

Returns the supported ontology.
**Returns**
the ontology.

---

**write**

```
public java.lang.String write(Object service)
```

Given a service object it returns a string representing the service.
**Parameters**
**service -** the service object to be converted.
**Returns**
the string representing the service.
**Throws**
**-**

## Class ReaderImpl

```
public class ReaderImpl implements Reader
```

Title: ReaderImpl
Description: A class implementing the Reader interface. It support OWL-s 1.0.
Copyright: Copyright (c) 2004
Company: IMIT/KTH

**Version**
1.0

### Constructors

**ReaderImpl**

`public ReaderImpl()`

Constructor of ReaderImpl.

### Methods

**getOntology**

`public java.lang.String getOntology()`

Returns the supported ontology.
**Returns**
the ontology.

---

**read**

`public java.lang.Object read(String service)`

Parses the given string and returns a Service Object.
**Parameters**
**service -** the string representing the service.
**Returns**
the Service object.
**Throws**
**-** **-**

## Package content.lang

This package contain language specific classes as well as interfaces used when agents communicate.

## Interface Description

`public interface Description`

Title: Description
Description: An interface for an Description message.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

## Methods

### getLanguage

```
public java.lang.String getLanguage()
```

---

### getOntology

```
public java.lang.String getOntology()
```

---

### getEncoding

```
public java.lang.String getEncoding()
```

---

### setDone

```
public void setDone(boolean done)
```

---

### setDone

```
public void setDone(String done)
```

---

### setResult

```
public void setResult(String result)
```

---

### getDone

```
public boolean getDone()
```

---

### getResult

```
public java.lang.String getResult()
```

---

### toString

```
public java.lang.String toString()
```

## Interface CreateObject

```
public interface CreateObject
```

Title: CreateObject
Description: An interface for parsing message content.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Methods

**createObject**

```
public java.lang.Object createObject(ACLMessage msg)
```

## Interface Action

```
public interface Action
```

Title: Action
Description: An interface for an Action message.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Fields

**FINDSERVICE**

```
public static finalFINDSERVICE
```

### Methods

**getLanguage**

```
public java.lang.String getLanguage()
```

**getOntology**

```
public java.lang.String getOntology()
```

---

**getEncoding**

```
public java.lang.String getEncoding()
```

---

**setActor**

```
public void setActor(String actor)
```

---

**setAct**

```
public void setAct(String act)
```

---

**setArgument**

```
public void setArgument(String arg)
```

---

**getID**

```
public java.lang.String getID()
```

---

**getActor**

```
public java.lang.String getActor()
```

---

**getAct**

```
public java.lang.String getAct()
```

---

**getArgument**

```
public java.lang.String getArgument()
```

---

**toString**

`public java.lang.String **toString**()`

## Package content.lang.fipardf0

This package holds classes implementing fipa-rdf-0.

## Class FipaDescription

```
public class FipaDescription extends FIPA implements
Description
```

Title: FipaDescription
Description: A class representing a FIPA RDF Description.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
  1.0

### Fields

**DESC**

`public static finalDESC`

---

**ABOUT**

`public static finalABOUT`

---

**DONE**

`public static finalDONE`

---

**RESULT**

`public static finalRESULT`

---

**TRUE**

`public static final`**`TRUE`**

---

**FALSE**

`public static final`**`FALSE`**

## Constructors

**FipaDescription**

`public` **`FipaDescription`**`(String about)`

Constuctor of FipaDescription.
**Parameters**
  **about -** the id of the action it's describing.

## Methods

**setDone**

`public void` **`setDone`**`(boolean done)`

Sets whether the action was successfully executed.
**Parameters**
  **done -** true if it was successful otherwise false.

---

**setDone**

`public void` **`setDone`**`(String done)`

Sets whether the action was successfully executed.
**Parameters**
  **done -** representing true or false.

---

**setResult**

`public void` **`setResult`**`(String result)`

Sets the result of the action.
**Parameters**
  **result -** the results.

---

**getDone**

`public boolean getDone()`

Returns the status of the action.

**Returns**

True if it was successful otherwise false.

---

**getResult**

`public java.lang.String getResult()`

Returns the status of the action.

**Returns**

A string representing the status.

---

**build**

`public Document build()`

Builds the document of the description.

**Returns**

The result document.

## Class FipaAction

`public class FipaAction extends FIPA implements Action`

Title: FipaAction
Description: A class representing a FIPA RDF Action.
Copyright: Copyright (c) 2004
Company: IMIT/KTH

**Version**

1.0

### Fields

**ACTION**

`public static finalACTION`

---

**ID**

`public static finalID`

---

**ACTOR**

`public static final`**`ACTOR`**

---

**ACT**

`public static final`**`ACT`**

---

**ARG**

`public static final`**`ARG`**

## Constructors

**FipaAction**

`public `**`FipaAction`**`(String id)`

Constructor of FipaAction.
**Parameters**
**id -** the id of the message.

## Methods

**setActor**

`public void `**`setActor`**`(String actor)`

Sets the actor of the action.
**Parameters**
**actor -** the actor.

---

**setAct**

`public void `**`setAct`**`(String act)`

Sets the act of the action.
**Parameters**
**act -** the act.

---

**setArgument**

`public void `**`setArgument`**`(String arg)`

Sets the argument of the action.
**Parameters**
**arg -** the argument.

---

### getID

```
public java.lang.String getID()
```

Returns the id of the action.
**Returns**
The id.

---

### getActor

```
public java.lang.String getActor()
```

Returns the actor of the action.
**Returns**
The actor.

---

### getAct

```
public java.lang.String getAct()
```

Returns the act of the action.
**Returns**
The act.

---

### getArgument

```
public java.lang.String getArgument()
```

Returns the argument of the action.
**Returns**
The argument.

---

### build

```
public Document build()
```

Builds the document of the action.
**Returns**
The result document.

## Class CreateObjectImpl

```
public class CreateObjectImpl implements CreateObject
```

Title: CreateObjectImpl
Description: Prarses the content of ACL messages into fipa-rdf0.
Copyright: Copyright (c) 2004

Company: IMIT/KTH
**Version**
1.0

## Constructors

### CreateObjectImpl

`public CreateObjectImpl()`

Constructor of CreateObjectImpl.

## Methods

### createObject

`public java.lang.Object createObject(ACLMessage msg)`

Parses the content ACLMessage into the associated FIPA object.
**Parameters**
**msg -** the ACLMessage carring the content of interest.
**Returns**
The parsed FIPA object.

---

### isFipaDescription

`public boolean isFipaDescription(Document doc)`

Checks whether the given document is a FIPA Description.
**Parameters**
**doc -** the document to be checked.
**Returns**
True if the document is a FIPA Description, false otherwise.

## Package content.wsdl

This package contains interfaces and packages for managing WSDL documents.

## Interface WSDL2Onto

`public interface WSDL2Onto`

Title: WSDL2Onto
Description: An interface for translating a WSDL document into a service object.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Methods

**createService**

```
public java.lang.Object createService(WSDLOperation op, Object[]
data)
```

## Interface NewService

```
public interface NewService
```

Title: NewService
Description: An interface for creating new WSDL service descriptions.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Methods

**createServce**

```
public WSDLOperation createServce(String operationName, String[]
inNames, QName[] inTypes, String[] outNames, QName[] outTypes)
```

## Package content.wsdl.impl

This package contains classes for creating new WSDL document as well as converting existing ones.

## Class WSDL2OWLs

```
public class WSDL2OWLs implements WSDL2Onto
```

Title: WSDL2OWLs
Description: An implementation of the WSDL2OWLs interface supporting OWL-S 1.0. This class is an reconstruction of an class in the owl-s api.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

### Constructors

### WSDL2OWLs

```
public WSDL2OWLs()
```

Constructor of WSDL2OWLs.

## Methods

### createService

```
public java.lang.Object createService(WSDLOperation op, Object[]
data)
```

Creates a new service object in OWL-S.
**Parameters**
  **op -** the WSDL operation to be converted.
  **data -** additional service data.
**Returns**
  the constructed service object.

---

### createOWLS

```
public Service createOWLS(WSDLOperation op, String serviceName,
String textDescription, String[] inputNames, String[] inputTypes,
String[] inputGroundings, String[] outputNames, String[] outputTypes,
String[] outputGroundings)
```

Creates a service object in OWL-S given a WSDL operation.
**Parameters**
  **op -** the WSDL opreation.
  **serviceName -** name of the service.
  **textDescription -** a text description of the service.
  **inputNames -** array containing input names.
  **inputTypes -** array containing input types.
  **inputGroundings -** array containing input groundings.
  **outputNames -** array containing output names.
  **outputTypes -** array containing output types.
  **outputGroundings -** array containing output groundings.
**Returns**
  the generated service object.

## Class NewServiceImpl

```
public class NewServiceImpl implements NewService
```

Title: NewServiceImpl
Description: An implementation of the NewService interface for creation of WSDL service descriptions.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
  1.0

### Constructors

**NewServiceImpl**

```
public NewServiceImpl()
```

Constructor of NewServiceImpl.

### Methods

**createServce**

```
public WSDLOperation createServce(String operationName, String[]
inNames, QName[] inTypes, String[] outNames, QName[] outTypes)
```

Creates a WSDL service descriptions.
**Parameters**
 **operationName -** the service name.
 **inNames -** the input names.
 **inTypes -** the input types.
 **outNames -** the output names.
 **outTypes -** the output types.
**Returns**
 the generated WSDL service description.

## Package matcher

This package contains interface for service matching as well as implementations of service matchers.

## Interface ServiceMatcher

```
public interface ServiceMatcher
```

Title: ServiceMatcher
Description: An interface for matching of service descriptions.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
 1.0

### Methods

**match**

```
public java.lang.Object match(Object subject, Object[] advertisment)
```

## Package matcher.impl

This package holds implementations of service matchers.

## Class OWLSMatcher

```
public class OWLSMatcher implements ServiceMatcher
```

Title: OWLSMatcher
Description: An implementation of the ServiceMatcher interface with support for matching of OWL-S 1.0.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
    1.0

### Constructors

#### OWLSMatcher

```
public OWLSMatcher()
```

Constructor of OWLSMatcher.

---

#### OWLSMatcher

```
public OWLSMatcher(int weightInput, int weightOutput, int weightName,
int weightTaxonomy)
```

Constructor of OWLSMatcher that set the weights of matching.
**Parameters**
  **weightInput -** the weight for input matching.
  **weightOutput -** the weight for output matching.
  **weightName -** the weight for names matching.
  **weightTaxonomy -** the weight for taxonomy matching.

### Methods

#### match

```
public java.lang.Object match(Object service, Object[] services)
```

Matches the given service against the array of services and returns the best match found.
**Parameters**
  **service -** the requested service.
  **services -** the advirtsed services.
**Returns**
  The best match.

## Package storage

This package specifies a storage interfaces and holds a package with storage implementations.

### Interface Storage

```
public interface Storage
```

Title: Storage
Description: An interface for storing of service objects.
Copyright: Copyright (c) 2004
Company: IMIT/KTH
**Version**
1.0

#### Methods

**add**

```
public void add(Object service)
```

---

**remove**

```
public void remove(Object service)
```

---

**getServices**

```
public java.lang.Object[] getServices()
```

## Package storage.impl

This package holds storage implementations.

### Class StorageImpl

```
public class StorageImpl implements Storage
```

Title: StorageImpl
Description: An implementation of of the Storage inteface using a simple Vector.
Copyright: Copyright (c) 2004

Company: IMIT/KTH
**Version**
1.0

## Constructors

**StorageImpl**

```
public StorageImpl()
```

Constructor of StorageImpl.

## Methods

**add**

```
public void add(Object service)
```

Adds a service to the storage.
**Parameters**
**service -** the service to be added.

---

**remove**

```
public void remove(Object service)
```

Removes a service from the storage.
**Parameters**
**service -** the service to be removed.

---

**getServices**

```
public java.lang.Object[] getServices()
```

Returns a list of all available services.
**Returns**
All the available services.