A Peer-to-Peer Distributed Semantic Service Registry

Rune Peter Bjørnstad
Enschede, The Netherlands.
August 2004.

Thesis for a Master of Science degree in Computer Science from the University of Twente, Enschede, The Netherlands.

Graduation Committee:

MSc. Tommy Gagnes
MSc. Ole-Erik Hedenstad
Dr. ir. M. J. van Sinderen
Dr. L. Ferreira Pires
i. Summary

Building distributed systems normally requires detailed information about the individual components that make up the system. For example, component locations are often stored in naming servers or registries, which serves as centralized sources of location information. A component is found by querying the registry for its location. In stable situations this scenario works well, but in highly dynamic systems, like the ones that are likely to appear in the future Network-Based Defense, relying on a centralized registry is a huge security risk. If the registry is removed from the system, for example as a result of sabotage, the remaining components are not able to find each other, and thus is the system crippled.

In this thesis I propose an architecture for a service registry that is distributed using a Peer-to-Peer network. It is designed to be operational as long as there are components on the network. Instead of centralizing information, the information is kept at the component hosts themselves, which are dynamically connected through the Peer-to-Peer network. The result is service registry without centralization, which stays operational as long as there are components on the network, and where the information is always up-to-date.

In the future Network-Based Defense, the components that make up the system are often nondeterministic, meaning that they may join and leave the system in a nondeterministic fashion. This uncertainty in the system means that the availability of a specific component cannot be guaranteed. Instead of stating dependencies between specific components, a component can be dependent of component capabilities, which can be provided by several components on the network. Finding components with certain capabilities becomes a problem.

In the proposed registry architecture, components or services are represented using semantically rich information. This is to allow a client to direct capability specific queries to the registry, in order to find a service that possesses a required set of capabilities. This is in contrast to the current situation where knowledge of a service’s implementation details is often required in order to locate a service.

A prototype of the proposed registry architecture has been developed in order to demonstrate the core functionality of the registry. The prototype relies on Jxta for Peer-to-Peer networking and Jena for representing and querying semantic service description. These descriptions are defined using the OWL-S ontology. In addition, a demonstration environment has been provided that enables experimentation with the registry.

I have found that distribution of a service registry using Peer-to-Peer networking is a successful method for avoiding centralization. There are, however, performance implications with this solution that restricts its application domain. Systems that require registry functionality, but who need to avoid centralization can profit from the proposed solution.

Finding services based on their semantic properties is a registry feature that provides some interesting possibilities. Due to open issues regarding OWL-S as a service description ontology, I consider the technology only applicable for defining the semantic capabilities of a service. I find the process and grounding related features of OWL-S not ready for production.
ii. Preface

This thesis describes the result of the assignment, which was done for the Norwegian Defense Research Establishment and is part of the Dutch Master of Science degree. This thesis was written for the Architecture of Distributed Systems group at the University of Twente.

This thesis would have been impossible without the support and motivation from my supervisors, both at the Norwegian Defense Research Establishment and the University of Twente. Thanks go out to Tommy Gagnes, Ole-Erik Hedenstad and Marten J. van Sinderen, without your support this would not have been possible.

Oslo, July 2004.

Rune Peter Bjørnstad.
### iii. Table of Content

I. SUMMARY ........................................................................................................ I

II. PREFACE ......................................................................................................... II

III. TABLE OF CONTENT ................................................................................... III

IV. LIST OF FIGURES ........................................................................................ VI

1 INTRODUCTION ................................................................................................. 1
   1.1 MOTIVATION ................................................................................................ 1
   1.2 OBJECTIVE .................................................................................................. 3
   1.3 APPROACH .................................................................................................... 4
   1.4 STRUCTURE .................................................................................................. 4

2 REPRESENTING CONCEPTS USING ONTOLOGIES ......................................... 6
   2.1 INTRODUCTION ............................................................................................... 6
   2.2 USING RDF AS AN ONTOLOGY DESCRIPTION LANGUAGE ...................... 7
      2.2.1 The RDF/XML Syntax .......................................................................... 8
   2.3 EXPRESSING MEANING USING RDF SCHEMA .......................................... 9
   2.4 THE WEB ONTOLOGY LANGUAGE ............................................................... 11
      2.4.1 The structure of an OWL-based ontology ............................................. 11
      2.4.2 OWL Class Definitions ....................................................................... 12
      2.4.3 OWL Property Definitions .................................................................. 12
      2.4.4 OWL Class Restrictions ...................................................................... 14
   2.5 SUMMARY ..................................................................................................... 14

3 QUERYING ONTOLOGIES .................................................................................. 16
   3.1 INTRODUCTION ............................................................................................... 16
   3.2 USING OWL-QL FOR QUERYING ONTOLOGIES ....................................... 16
      3.2.1 Answer Dialogues .................................................................................. 17
      3.2.2 Query Patterns ...................................................................................... 17
      3.2.3 Variable Binding .................................................................................... 18
      3.2.4 Assumption Support .............................................................................. 18
   3.3 USING RDQL FOR QUERYING ONTOLOGIES ............................................ 19
      3.3.1 The RDQL Syntax ................................................................................ 20
      3.3.2 Data-Model modification for OWL Queries .......................................... 20
   3.4 SUMMARY ..................................................................................................... 22

4 REPRESENTING SERVICES USING ONTOLOGIES .......................................... 23
   4.1 INTRODUCTION ............................................................................................... 23
   4.2 USING OWL-S FOR SERVICE REPRESENTATION ..................................... 23
   4.3 OWL-S DOCUMENT STRUCTURE ................................................................. 24
   4.4 THE SERVICE ONTOLOGY .......................................................................... 25
   4.5 SERVICE PROFILE ....................................................................................... 25
      4.5.1 Non-Functional Profile Properties ......................................................... 26
      4.5.2 Parameter support ................................................................................. 26
   4.6 PROCESS MODEL .......................................................................................... 27
      4.6.1 Atomic Processes .................................................................................. 27
      4.6.2 Composite Processes .......................................................................... 28
   4.7 SERVICE GROUNDING ............................................................................... 29
      4.7.1 Process Binding ...................................................................................... 30
      4.7.2 Parameter Binding ................................................................................ 31
      4.7.3 WSDL Message Mapping ..................................................................... 32
   4.8 SUMMARY ..................................................................................................... 33
5 DISTRIBUTED REGISTRIES

5.1 INTRODUCTION

5.2 PEER-TO-PEER NETWORKING

5.3 JXTA-BASED PEER-TO-PEER NETWORKING

5.3.1 Jxta Concepts

5.3.2 The Jxta Architecture

5.3.3 The Peer Resolver Protocol

5.4 SUMMARY

6 A PROPOSED SEMANTIC, DISTRIBUTED SERVICE REGISTRY

6.1 INTRODUCTION

6.2 THE ARCHITECTURE

6.2.1 Garbage Collection

6.3 THE QUERY SERVICE

6.3.1 Administration

6.4 THE QUERY DISTRIBUTION SERVICE

6.5 USAGE SCENARIOS

6.5.1 Runtime Dynamic Proxy

6.5.2 Service Type specific Proxy

6.6 SUMMARY

7 THE CONCEPTUAL QUERY SERVICE

7.1 INTRODUCTION

7.2 OWL-S SERVICE DESCRIPTION ADMINISTRATION

The OWL-S Verifier

OWL-S Document Version Control

7.3 THE QUERY PROCESSOR

The Knowledge base

7.4 SUMMARY

8 THE CONCEPTUAL DISTRIBUTION SERVICE

8.1 INTRODUCTION

8.2 THE DISTRIBUTION SERVICE ARCHITECTURE

8.2.1 Registry Interface

8.2.2 QueryService Interface

8.2.3 Administration Interface

8.3 THE QUERY ROUTER

8.3.1 Peer Group Routing

8.3.2 Provider PeerGroup Handler

8.4 THE ONTOLOGY RESOLVER

8.5 SUMMARY

9 THE DEMONSTRATOR

9.1 INTRODUCTION

9.2 REGISTRY CONFIGURATION

9.3 QUERY SERVICE IMPLEMENTATION

9.3.1 The QueryService remote interface

9.3.2 The Administration interface

9.3.3 The Knowledge base

9.3.4 Distribution Service Association

9.4 DISTRIBUTION SERVICE IMPLEMENTATION

9.4.1 The Query Router

9.4.2 The Provider Interface

9.4.3 The Registry Interface

9.4.4 The Administration Interface

9.5 SAMPLE ONTOLOGY OF CONCEPTS
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Updating object references at registry.</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Invoking methods on remote accessible service.</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>A malfunctioning registry halts the whole system.</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Two examples of a &quot;tanker&quot;.</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>Semantic expressiveness of ontologies.</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>Triplet as a directed graph.</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Example triplets represented as directed graph.</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>RDF graph representation of XML serialization.</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>Using a resource as the object of a triplet.</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>Several alternative RDF query languages.</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>OWL-QL Dialogue.</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>RDF graph of subclass relation.</td>
<td>19</td>
</tr>
<tr>
<td>13</td>
<td>RDF Graph with implied OWL Inference step.</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>Full-Inference Mode.</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>On-Demand Inference Mode.</td>
<td>22</td>
</tr>
<tr>
<td>16</td>
<td>Separation between Interface description and protocol bindings.</td>
<td>23</td>
</tr>
<tr>
<td>17</td>
<td>Structure of OWL-S Service Description.</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>Sequence control structure.</td>
<td>29</td>
</tr>
<tr>
<td>19</td>
<td>Service Grounding using WSDL.</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>SOAP-based datatype conversion.</td>
<td>31</td>
</tr>
<tr>
<td>21</td>
<td>Client/Server(left) and Peer-to-Peer(right) Paradigm.</td>
<td>35</td>
</tr>
<tr>
<td>22</td>
<td>Cross-application interoperability using translators.</td>
<td>35</td>
</tr>
<tr>
<td>23</td>
<td>Endpoint Service.</td>
<td>38</td>
</tr>
<tr>
<td>24</td>
<td>ResolverQuery directed at query handler &quot;OWL-QL&quot;.</td>
<td>39</td>
</tr>
<tr>
<td>25</td>
<td>Peer-to-Peer distributed service registry.</td>
<td>40</td>
</tr>
<tr>
<td>26</td>
<td>Interaction between querying peer and hosting peer.</td>
<td>41</td>
</tr>
<tr>
<td>27</td>
<td>Sequence Diagram of Distribution Service usage.</td>
<td>41</td>
</tr>
<tr>
<td>28</td>
<td>Difference between a Peer-to-Peer distributed registry and UDDI registries.</td>
<td>42</td>
</tr>
<tr>
<td>29</td>
<td>Sequence Diagram of interaction between Distribution Service and Query Service.</td>
<td>43</td>
</tr>
<tr>
<td>30</td>
<td>Administration of QueryService using SNMP agent.</td>
<td>44</td>
</tr>
<tr>
<td>31</td>
<td>Typical static UDDI Registry configuration.</td>
<td>44</td>
</tr>
<tr>
<td>32</td>
<td>The proposed Peer-2-Peer distributed registry.</td>
<td>45</td>
</tr>
<tr>
<td>33</td>
<td>Hosting OWL-S description on behalf of other hosts.</td>
<td>45</td>
</tr>
<tr>
<td>34</td>
<td>A proxy-base usage of the proposed registry.</td>
<td>46</td>
</tr>
<tr>
<td>35</td>
<td>Runtime Dynamic Proxy.</td>
<td>46</td>
</tr>
<tr>
<td>36</td>
<td>Sequence Diagram describing service type proxy.</td>
<td>47</td>
</tr>
<tr>
<td>37</td>
<td>Query Service Architecture.</td>
<td>49</td>
</tr>
<tr>
<td>38</td>
<td>Query Architecture.</td>
<td>49</td>
</tr>
<tr>
<td>39</td>
<td>Query Processor.</td>
<td>52</td>
</tr>
<tr>
<td>40</td>
<td>Augmentation of RDQL query by the Query Modifier.</td>
<td>52</td>
</tr>
<tr>
<td>41</td>
<td>Jena Reasoning architecture.</td>
<td>53</td>
</tr>
<tr>
<td>42</td>
<td>A possible environment that the peer-to-peer network operates in.</td>
<td>55</td>
</tr>
<tr>
<td>43</td>
<td>Distribution Service Layering.</td>
<td>56</td>
</tr>
<tr>
<td>44</td>
<td>Distribution Service.</td>
<td>56</td>
</tr>
<tr>
<td>45</td>
<td>Distribution Service interfaces.</td>
<td>56</td>
</tr>
<tr>
<td>46</td>
<td>Client-Registry interaction.</td>
<td>58</td>
</tr>
<tr>
<td>47</td>
<td>Super Peer-based routing.</td>
<td>59</td>
</tr>
<tr>
<td>48</td>
<td>Jxta Application-based Peer Groups.</td>
<td>59</td>
</tr>
<tr>
<td>49</td>
<td>Relation between Profile-class and Jxta-Group.</td>
<td>61</td>
</tr>
<tr>
<td>50</td>
<td>PeerGroup Handlers.</td>
<td>62</td>
</tr>
<tr>
<td>51</td>
<td>Client host JBoss configuration.</td>
<td>65</td>
</tr>
<tr>
<td>52</td>
<td>Service Provider JBoss configuration.</td>
<td>65</td>
</tr>
<tr>
<td>53</td>
<td>The basic demonstrator functionality.</td>
<td>66</td>
</tr>
<tr>
<td>54</td>
<td>JBoss client configuration.</td>
<td>67</td>
</tr>
<tr>
<td>55</td>
<td>JBoss service provider configuration.</td>
<td>67</td>
</tr>
<tr>
<td>56</td>
<td>Interface and implementation of Query Service.</td>
<td>68</td>
</tr>
<tr>
<td>57</td>
<td>Wrapping QueryService inside JMX MBean.</td>
<td>68</td>
</tr>
<tr>
<td>58</td>
<td>DistributionService usage of QueryService.</td>
<td>69</td>
</tr>
<tr>
<td>59</td>
<td>Association between Query Service and Knowledge base.</td>
<td>70</td>
</tr>
<tr>
<td>60</td>
<td>Implementation of Distribution Service interfaces.</td>
<td>71</td>
</tr>
</tbody>
</table>
Figure 61: Loading DistributionService into JBoss using MBean. 72
Figure 62: Distribution Service implementation environment. 72
Figure 63: Class-diagram of Query Router. 73
Figure 64: Sub-graph of the military ontology illustrating Jxta relation. 73
Figure 65: Sequence Diagram illustrating interaction with Query Router. 74
Figure 66: Association between Distribution Service and Query Service using PeerGroupHandler. 75
Figure 67: Sequence Diagram illustrating handleGroup method. 75
Figure 68: Client invocation of the Distribution Service. 76
Figure 69: Graph of military sample ontology (OWL-version of [71]). 78
Figure 70: The Domain hierarchy of OWL classes. 79
Figure 71: OWL graph of SensorOutput hierarchy. 79
Figure 72: OWL graph of ClassificationCapability hierarchy. 80
Figure 73: Package hierarchy of sample ontology class library. 81
Figure 74: Datatypes classes. 81
Figure 75: Sensor classes. 82
Figure 76: Sequence diagram illustrating ISAR proxying. 82
Figure 77: Sensor proxies provided by the Demonstrator. 83
Figure 78: Sensor Output classes. 83
Figure 79: Sequence diagram of TrackHandler invocation. 84
Figure 80: Sensor Domain classes. 84
Figure 81: Classification Capability classes. 84
Figure 82: Simulation of enemy airplane position. 85
Figure 83: The place of the OWL-S Invoker in a complete architecture. 91
Figure 84: Usage of the Web Service Invocation Framework. 92
Figure 85: The main classes provided by the OWL-S Invocation Framework. 92
Figure 86: XSLT Transformation of parameters. 95
Figure 87: Type conversion using Marshaller and Unmarshaller. 97
Figure 88: Sequence diagram of EJB method invocation using the DynamicInvoker. 98
Figure 89: Sequence Diagram illustrating demonstrator application client. 99
1 Introduction

In a Network-Based Defense (NBD), NBD components may appear and disappear continuously or change network bearer, something that imposes serious demands on system administration. The NBD infrastructure will be composed of a vast amount of different systems, spanning multiple middleware, network and hardware solutions.

Systems interoperating in such a setting cannot rely on static locations of resources or the continuous availability of services. A given system cannot stop working simply because a required cooperating service is not available at the time. There may be other services on the network that can just as well do the job. Finding and invoking these substitute services does in most cases require human interaction. Since services come in many flavours, finding a service that performs the needed functionality is only a step in the right direction. If the implementation details of the service are incompatible with the existing system, the service cannot be used even though it provides the needed functionality.

1.1 Motivation

This thesis considers how future Network Based Defence components will interoperate in the battlespace. In this context the need for information grows, and the networking conditions get worse. The availability of high-priority centralized servers and services is a risk to the system. It is acknowledged that systems are implemented using various compatible and incompatible middleware solutions, and the paradox is that the need for tighter integration between systems will only grow.

In a typical middleware-based system, services are remotely accessible and can thus be invoked across network barriers. This means that a service's address needs to identify the network location of the service, as well as versioning information and serialization attributes. Consider, for example, the following CORBA [01] Interoperable Object Reference (IOR), which identifies a service object:

IOR:0000000000000000f49444c3a52616e646f6d3a312e3000000000001000000000000000000005
e6706c616e78792e6473672e63732e7463642e696500062200002ca5e706c616e78792e6473672e63732e7463642e69653a52616e646f6d3a312e3000000000001000000000000000000005
e6706c616e78792e6473672e63732e7463642e69653a52616e646f6d3a312e3000000000001000000000000000000005

Using the IOR a client application is able to locate a remote service object, and invoke operations on it. The problem is that an IOR is only readable by a computer, and they can change regularly. Because of this most middleware platforms involves some kind of Lookup Service (LUS), or Registry.

The registry provides the core knowledge of a system, namely object references, or IOR in a CORBA-based application. The registry relates a static, human-readable object name with a possibly transient object reference. Whenever a service is started, it locates its designated registry, and updates its object reference together with its object name (see Figure 1).
Whenever an agent/client in the system needs to invoke methods on a service, it first queries the registry for the object reference of that service, using its human-readable object name. With the object reference, for example a CORBA IOR, the agent is able to locate the service object and invoke its methods (see Figure 2).

Although this approach works in stable situations, the cooperating services are totally dependent of the registry, and if it fails, the whole system fails, even though the individual services are up and running (see Figure 3).

In the event that a registry is malfunctioning, or removed from the system, the system components are unable to interoperate, even though they are all up and running. In the future NBD, building systems with such an obvious point-of-attack, is not an option and alternatives have to be found.

Depending on the middleware platform used, the registry’s capability ranges from simple name matching systems, like the CORBA Naming Service, to UDDI [02].
registries, which does not only know the location of components, but also keeps meta-data concerning the system. In the future NBD, the systems and services of cooperating parties are required to interoperate on-demand, regardless of implementation details. This is already possible today, through the emergence of Web Services [03]. By supplying the registry with enough information about components, direct invocation of those components is possible.

In a typical Web Services-based system, the UDDI registry provides Web Services Description Language (WSDL) documents, which goes far beyond IOR when it comes to describing services. Still its focus is on service location and its operations, and not on capabilities. It is a desired feature of the future NBD that services are able to locate other services, based on their capabilities. The motivation behind this is that if a certain service fails in the middle of a transaction, the remaining parties can locate a substitute service, capable of delivering the same functionality, completely without human interaction. This is the main focus of this report.

1.2 Objective

The objective of this report is to suggest a distributed architecture for a service registry, where services can be found based on their described semantic capabilities, rather than their processes, and where all centralization is avoided. Because of the requirements of the future NBD, services should be found and used regardless of their implementation details, something that delegates a lot of responsibility to service descriptions.

The Web Services Description Language [04] makes it possible to describe services, regardless of their implementation details, in an abstract way. Using a WSDL service description, a service user is able to invoke methods, regardless of the implementation specifics of the service.

By examining a WSDL description of a service it is not possible to determine what the service actually does. This makes WSDL only part of a possible solution for the required registry. This is also the case when it comes to the use of WSDL in Web services architectures. For this reason the Universal Description, Discovery and Integration [02] registry was designed. In a UDDI registry a WSDL document is associated with some fixed meta-data, which defines what type of service the WSDL document describes.

The meta-data support of UDDI is, however, limited and business oriented. It does relate WSDL documents with taxonomy-based category codes, but with the emergence of the Semantic Web [05], and ontology-based service description languages like DAML-S [07], which has evolved into OWL-S [06], WSDL is too limited as a service description language. This thesis proposes an architecture that enables service descriptions based on powerful ontology languages like OWL [08] and RDF Schema [09], for example OWL-S, to represent the semantics of services.

There is work in progress that tries to integrate DAML-S/OWL-S support into UDDI registries [10], [11], [12], and ebXML [13] registries [14]. This integration works, but involves mapping DAML-S/OWL-S constructs to existing similar constructs within the registries. This fails to harness the power of ontologies since ontology-based query reasoning is not supported. In addition both UDDI and ebXML registries are centralized by design, though both provide some decentralization support. This thesis proposes an architecture of a truly distributed registry where obvious points-of-attack are avoided. The proposed architecture should go beyond ontology mapping and provide sufficient query reasoning support.
Massimo et al [15] describes a broker-based architecture that uses OWL-S to describe services. A broker mediates between a service requester and several service providers, where it accepts ontology-based queries and invokes methods on found services. The architecture supports OWL reasoning, and is thus powerful when it comes to describing and querying for capabilities. This broker-based architecture, however, does contain centralization because the service providers advertise OWL-S descriptions at the broker itself. This thesis should propose an architecture that provides ontology reasoning, but avoid centralization.

A Gnutella [17] peer-to-peer distributed registry for DAML-S service descriptions is described in [16]. By distributing the registry on a peer-to-peer network, centralization is avoided and there is no more obvious points-of-attack. This work demonstrates that a peer-to-peer distributed registry is a feasible solution; however, it fails to harness the real power of ontologies by not supporting query reasoning. In addition, the Gnutella peer-to-peer network relies on HTTP-based communication, something that may not always be available. This thesis should propose an architecture that is operational under most conditions, and totally independent of the underlying network bearer.

The Jxta-based [18] RDF meta-data infrastructure Edutella [19] provides a peer-to-peer-based system for querying and publishing RDF-based ontologies. The project JxtaSearch [20] provides a similar functionality. Although ontology support is limited to RDF, and that some centralization is introduced into the system, the solutions provided by both Edutella and JxtaSearch are promising when it comes to designing a distributed semantic registry.

**1.3 Approach**

The first step in the design of Universal, Distributed, Semantic Registry is to study existing technologies on the subject. This involves studies of existing service descriptions technologies, query technologies and distributed registry technologies.

Since the services need to be represented with semantic properties, aligning this research with the Semantic Web [05] effort is an implied requirement. This also means that the query technology used in the proposed architecture should be capable of reasoning about the semantics of services.

After selecting suitable technologies, an overall conceptual architecture is proposed. The selected technologies are then integrated into the architecture, together with alterations and additions where needed.

This assignment aims at providing a proof of concept. With this regard a prototype has been built, demonstrating the core functionality of the architecture.

**1.4 Structure**

The rest of this report is structured as follows:

- Chapter 2 gives an overview of the current work with regard to concept representation using ontologies. This includes ontologies based on RDF, RDF/S and OWL.
- Chapter 3 gives an overview of the current work with regard to querying ontologies. A comparison is made between existing, suitable technologies.
- Chapter 4 gives an overview of the current work with regard to service representation using ontologies. This includes an introduction to OWL-S, as
well as some suggestions regarding the use of OWL-S in the proposed architecture.

- Chapter 5 gives an overview of current technologies with regard to distributed registries. It gives a brief introduction to distributed technologies that does fit in with the requirements of this report, and holds a strong emphasis on Peer-to-Peer technologies, with Jxta in general.

- Chapter 6, 7 and 8 describes the architecture of the proposed distributed lookup service. First an overall conceptual architecture is given in chapter Error! Reference source not found., then the Query Service is described in chapter Error! Reference source not found., and finally the Distribution Service is described in chapter Error! Reference source not found.

- Chapter 9 describes the demonstrator that is provided to demonstrate the core functionality of the proposed architecture.

- In chapter 10 the solution is discussed, and finally a conclusion is given in chapter 11.
2 Representing Concepts Using Ontologies

2.1 Introduction

In biology research it has been normal to classify species in a given system. This involves grouping species, putting constraints on relationships between species and defining the type of a species. As an example, consider the dog. It is a derivative of a wolf, which belongs to the canine family of mammals. This clear definition of hierarchy allows us to identify a species in far greater detail than would have been the case without such a system. The definition system itself is referred to as an ontology, i.e. the ontology of mammals.

The semantic expressiveness of an ontology system depends on the language constructs available. Consider, for example, a taxonomy, which is one of the most basic forms of ontologies (see Figure 5). The United Nations Standard Products and Services Code System (UNSPSC) [21] taxonomy defines classification codes for categories of products and services. Thus by relating the UNSPSC code 41111939 to a service, it is labelled as an Acoustic Sensor, according to the UNSPSC definitions. Although classification codes may be suitable in many situations, consider the example in Error! Reference source not found..

![Figure 4: Two examples of a “tanker”.](image)

The term tanker can refer to a military tank operator or an oil tanker. Although a human being is able to see the meaning of the term by looking at the image, a computer might not. By defining concepts using an ontology language that is semantically strong, it is possible for a computer to distinguish between the two types of tankers. The tank operator is a specialization of a military profession, and the oil tanker is a specialization of a ship. Depending on the semantic requirements present, the ontology language should be chosen with care (see Figure 5).

![Figure 5: Semantic expressiveness of ontologies.](image)
The graph displays the semantic expressiveness of various ontology languages ranging from basic classification taxonomies to the Web Ontology Language (OWL) [08].

With the Semantic Web [05] initiative, comes a vision where all resources on the Internet are defined using ontologies. This does not only relate to the individual resources located on a single web page, but also relations and constraints between web pages and services. By clearly classifying resources on the Internet, highly sophisticated search engines could become far more efficient in its searches. Depending on the detail of these ontologies, automated agents could do the browsing for you, because they are able to deduct the semantic meaning of resources.

2.2 Using RDF as an Ontology description language

When defining an ontology, an ontology language is needed. The computer industry seems to agree that XML [22] is the language of choice for describing these. The problem is that XML is far too general. The Resource Description Framework (RDF) [23] provides a concrete serialization context that is an extension of XML and provides us with the basic tools for defining concepts and relationships between concepts. RDF is based on the idea of identifying things using Unified Resource Identifiers (URI) [24], and the assumption that every single piece of knowledge can be defined using triples. A triple can, for now, be regarded as three pieces of information, often referred to as subject, predicate and object.

When describing a concept using RDF, the specific concept is referred to as the subject of a triple, for example, a Person. A subject may have several properties, for example, a person has a name and a gender. In RDF, a property is referred to as a predicate and relates a specific subject with a property value, which is referred to as the object of the relation. Figure 6 describes a triple using a directed graph, which is a representation often used when viewing RDF-based ontologies.

Figure 6: Triplet as a directed graph.

Consider the example of a person having a name and a gender. This piece of knowledge can be represented by two triples, both with person as its subject:

( Person, gender, Male )
( Person, name, “Homer” )

Again this can be represented using a directed graph (see Figure 7).

Figure 7: Example triplets represented as directed graph.

All RDF identifiable resources, like Person, Male, gender and name should be identified using URIs, but for readability this is not done in the examples above.
Reading an RDF-based ontology using a directed labeled graph is quite easy. This is one of the reasons why the RDF Core Working Group decided on a directed labeled graph as the default representation of the RDF data model.

In the RDF data model the triple subject, predicate, object can be represented in a generic way as a node-arc-node pattern. Additionally, a node comes in three flavors:

- **URIRef** nodes are identified by a unique URI.
- **Blank** nodes are unidentified, or anonymous, nodes.
- **Literals** represent objects only, not subject or predicate. A literal consist of three parts: a character string, an optional language part and a data type.

The arcs are directional and labeled with the predicate of the triple, and are directed from the subject to the object, like illustrated in Figure 6 and Figure 7.

### 2.2.1 The RDF/XML Syntax

Processing RDF ontologies as graphs may work out fine on paper, but the usual serialization of RDF is to XML, often represented as RDF/XML or plain RDF. An RDF-based ontology consist of a rdf:RDF root node, where relevant namespaces are defined, and with the actual ontology content contained within:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY sensors "http://www.ffi.no/ontology/sensors.rdf">
]>
.rdf:RDF
  xmlns:rdf="&rdf;#
  xmlns:sensors="&sensors;#">
...
</rdf:RDF>
```

The root node defines the default RDF syntax namespace as well as additional application specific namespaces. Using this namespace mechanism it is possible to construct ontologies that reuse concepts that are defined elsewhere.

By using DTD entity definitions, the ontologies can become less bloated with lengthy namespace references. Consider, for example, how the lengthy rdf and sensors namespaces are referenced in the above example using the entity references &rdf; and &sensors;.

For describing a subject, or resource, the RDF element *Description* is used. It uses the attribute rdf:ID to identify the specific resource that is described. The rdf:ID attribute uses an URI to identify the specific resource, for example, http://www.ffi.no/ontology/sensors.rdf#AcousticSensor.

```xml
<rdf:RDF xmlns:rdf="&rdf;#"
  xmlns:sensors="&sensors;#">
  <rdf:Description rdf:ID="&sensors;#AcousticSensor">
    <sensors:range>3000</sensors:range>
    <sensors:state>active</sensors:state>
  </rdf:Description>
</rdf:RDF>
```

The *Description* element contains two predicate/object pairs, defined as elements from the sensors vocabulary. The example above describes how Figure 8 is serialized into XML.
A Peer-to-Peer Distributed Semantic Service Registry

Figure 8: RDF graph representation of XML serialization.

The objects of both predicates in Figure 8 are serialized as RDF Literals. It is also possible for objects to be another RDF resource. Consider the introduction of a property manages that relates a sensor with a person that manages the sensor:

```
<rdf:RDF xmlns:rdf="&rdf;#"
  xmlns:sensors="&sensors;#">
  <rdf:Description rdf:about="&sensors;#AcousticSensor">
    <sensors:manages rdf:resource="#HomerSimpson">
  </rdf:Description>
  <rdf:Description rdf:ID="HomerSimpson">
    <sensors:firstname>Homer</sensors:firstname>
    <sensors:lastname>Simpson</sensors:lastname>
  </rdf:Description>
</rdf:RDF>
```

Using the `rdf:about` attribute it is possible to add properties to an existing identifiable resource. In the example above this feature is used to add the `sensor:manages` property to the existing `#AcousticSensor` definition. The `sensor:manages` predicate uses the `rdf:resource` attribute to identify the object of the triple.

RDF provides us with a language for expressing data models using an XML-based syntax. It provides the basic constructs for defining identifiable pieces of knowledge, and relations. It is, however, not sufficient enough to express the meaning of concepts. For expressing meaning, RDF schema’s or vocabularies is one popular option.

2.3 Expressing meaning using RDF Schema

The RDF syntax specification show how RDF constructs relate and how they can be serialized to XML. Another part of the RDF specification is the RDF Vocabulary Description Language; RDF Schema specification [09]. An RDF schema, or vocabulary, is a dictionary that defines elements of importance to a given domain. RDF Schema provides a type system and powerful constructs for defining relations between concepts. If RDF is a way of describing data, then the RDF Schema can be
considered a domain-neutral way of describing metadata that can be used to
describe the data for a domain-specific vocabulary [25].

The RDF Schema elements can be used in an ontology by introducing the RDF/S
specific namespace in the ontology's rdf:RDF root element (or at another suitable
location):

xmlns:rdfs="http://www.w3.org/2000/rdf-schema#"

All resources within RDF are implicitly members of the RDF/S class rdfs:Resource,
but the element used when introducing new concepts to a RDF/S-based ontology is
rdfs:Class.

<rdfs:Class rdf:ID="Sensor"/>

The rdfs:Class element is used to introduce a new concept identified by the rdf:ID
attribute's URI value. This resembles how new concepts are introduced in object-
oriented programming languages. And like in object-oriented programming
languages, RDF/S provides the means for creating specialized classes through class
extensions or sub-classing. Consider the following example:

<rdfs:Class rdf:ID="AcousticSensor">
  <rdfs:subClassOf rdf:resource="#Sensor"/>
</rdfs:Class>

Using the property rdfs:subClassOf it is possible to define the nature of the relation
between AcousticSensor and Sensor as an hierarchical one. This makes it possible to
group concepts in a hierarchic system, something that makes basic reasoning
possible. Consider, for example, the tanker example. By defining a tanker within
hierarchic system it is easy to reason about the difference between an oil tanker and
a military tank operator.

Analogue with the definition of concepts as classes, it is possible to define
predicates, or properties in RDF/S notation. The definition of properties is actually an
RDF syntax construct, but by combining the property with the RDF/S schema
element rdfs:subPropertyOf it is possible to define complex relationships between
resources:

<rdf:Property rdf:ID="hasClassificationCapability">
  <rdfs:domain rdf:resource="#Sensor"/>
</rdf:Property>

<rdf:Property rdf:ID="classifiesType">
  <rdfs:subPropertyOf rdf:resource="#hasClassificationCapability"/>
</rdf:Property>

The example above defines a property hasClassificationCapability that can be applied
to instances of the class Sensor. This application restriction is quite powerful and is
introduced using the rdfs:domain property, which is introduced by RDF/S. Further, a
property classifiesType is defined that is an extension of the
hasClassificationCapability property. This means that the property inherits the
application restriction of its super-property and can only be applied to instances of
class Sensor.

Analogue with the possibility to restrict the domain of a property relation, it is also
possible to restrict the possible object values of a relation:

<rdf:Property rdf:about="#hasClassificationCapability">
  <rdfs:range rdf:resource="#ClassificationCapability"/>
</rdf:Property>
This adds a value restriction to the hasClassificationCapability property using the rdfs:range element, introduced by RDF/S. All instances of hasClassificationCapability can now only be applied to instances of Sensor, and can only hold values that are instances of class ClassificationCapability, or an extension thereof.

Using RDF/S it is possible to define complex relationships between concepts in a way that can allow software to do some reasoning about the meaning of concepts. For example, it is possible to reason about the difference between a military tank operator and an oil tanker. Both can be called a tanker, but they possess both different properties and extend different classes.

RDF provides mechanisms that enable resource referencing, whereas RDF/S provides the tools for describing vocabularies. The vision is that when describing ontologies, there should be a vocabulary repository available of known concepts, known as Upper Level Ontologies. Thus, instead of defining everything from scratch, it should be possible to extend already defined concepts, and relate new concepts with existing ones. There are, however, some major shortcomings of RDF/S, and because of this the Web Ontology Language (OWL) [08] was developed.

2.4 The Web Ontology Language

With RDF/S it is possible to reason about ontologies using intelligent software. There are, however, several limitations when it comes to defining semantic constraints and semantic relationships using RDF/S. The Web Ontology Language (OWL) is a RDF/S-based language that goes beyond RDF/S when it comes to expressing semantic constraints and ontology relationships.

OWL is divided into three sub-languages with increasingly expressive characteristics. This division was done because of the intended users of OWL have different requirements concerning expressiveness, formal soundness and processing efficiency.

- **OWL Lite** provides support for simple classification and some limited constraint concepts.
- **OWL DL (Description Logic)** contains all the language constructs of OWL, but there are some restrictions on how these constructs may be used. These restrictions ensure that OWL DL ontologies are formally sound.
- **OWL Full** gives the user maximum expressiveness without any computational guarantees. An OWL Full ontology resides in an open-world environment, meaning that an ontology is considered unfinished because it can be extended elsewhere. Complete reasoning support for OWL Full is considered very unlikely.

2.4.1 The structure of an OWL-based ontology

OWL is based on RDF, thus an OWL ontology contains an rdf:RDF root node where, typically, the used namespaces are defined. Consider, for example, the following root node where the often-used RDF, RDF/S, XML Schema and OWL namespaces are defined:

```xml
<rdf:RDF
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#">```
In addition to the RDF compulsory rdf:RDF root node, an OWL-based ontology includes an owl:Ontology header element. The header element contains some metadata properties about the ontology, which includes comments, labeling, version control and inclusion of external ontologies. Consider, for example:

```xml
<owl:Ontology rdf:about=""
  <rdfs:comment>
    This is the ontology about military sensors.
  </rdfs:comment>
  <owl:imports rdf:resource="http://www.mil.no/geography#"/>
  <rdfs:label>Sensor Ontology</rdfs:label>
</owl:Ontology>
```

The empty rdf:about attribute of the header element is a standard way of defining that the current XML document’s base URI, is the URI of the ontology. The real content of an OWL-based ontology consists of its defined concepts and their properties and relationships. OWL uses an object-oriented approach, like with RDF/S, defining classes and properties. In addition, OWL has several constructs for stating constrains and similarity.

The possibility to include existing ontologies using the owl:import element, provides the mechanism that can realize repositories of well-known concept ontologies. Several such Upper Level Ontologies exist, and can be included in new ones, preventing redundant definitions of concepts.

### 2.4.2 OWL Class Definitions

An OWL ontology consists, mostly, of classes with their properties, instances and instance relationships. Each user-defined class is implicitly a subclass of owl:Thing, much like in the Java programming language [26] where every class is a subclass of java.lang.Object. A class in OWL is declared using the owl:Class element in the same way as with RDF/S. The introduction of a new OWL class construct, instead of reusing the RDF/S class construct, is done to make a clear distinction between OWL- and RDF/S-based ontologies:

```xml
<owl:Class rdf:ID="Sensor"/>
```

For defining sub-classes the RDF/S property rdfs:subClassOf is reused. In this way the OWL-language itself serves as a nice example of the ontology-reuse vision in realization:

```xml
<owl:Class rdf:ID="AcousticSensor">
  <rdfs:subClassOf rdf:resource="#Sensor"/>
</owl:Class>
```

With the class definition in place, it is possible to construct the instances of a defined class using standard XML constructs. Consider, for example:

```xml
<AcousticSensor rdf:ID="Sensor_Alpha"/>
```

The Sensor class instance with the unique ID Sensor_Alpha is defined. This ID can be left out, but in this case the software handling the ontology generates an ID.

### 2.4.3 OWL Property Definitions

If the ontology is entirely made up by empty classes with their instances, it would be more than sufficient to use RDF, or even a taxonomy, to define the ontology. Like in most object-oriented programming languages, OWL allows us to define class properties that assert facts about the members of a specific class. An OWL property is an extension of an RDF/S property, and provides two possible specializations:
Datatype properties are relations between instances of classes and RDF literals or XML Schema datatypes (XSD).

Object properties are relations between instances of two classes.

The OWL construct `owl:ObjectProperty` is used to introduce a new object property to the ontology. Consider, for example, the following ontology fragment:

```
<owl:ObjectProperty rdf:ID="hasOutput">
  <rdfs:domain rdf:resource="#Sensor"/>
  <rdfs:range rdf:resource="#SensorOutput"/>
</owl:ObjectProperty>
```

The `hasOutput` property defines a binary relation between two OWL class instances. The additional domain and range restrictions, which are reused from RDF/S, defines the subject of the relation to be an instance of a `Sensor` class, and the object to be an instance of the `SensorOutput` class. In addition to the `rdfs:range` and `rdfs:domain` constructs, the `rdfs:subPropertyOf` construct can also be reused from RDF/S in order to define more specialized object properties.

With datatype properties there are restrictions on the possible value range of a property. Only XML Schema datatypes or RDF literals are valid value-types of such a property. The OWL element `owl:DatatypeProperty` is used for defining such properties, much in the same way as with object properties:

```
<owl:DatatypeProperty rdf:ID="sensorRange">
  <rdfs:domain rdf:resource="#Sensor"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#long"/>
</owl:DatatypeProperty>
```

Like with object properties, the nature of the property defines a restriction on the possible relation objects. In addition the RDF/S domain and range restriction constructs can be used to restrict the relation further. In the fragment above the `sensorRange` property relates instances of the `Sensor` class with XSD `long` values.

In addition to the Object- and Datatype properties, OWL also defines the following property characteristics that are vital for formal validation of ontologies:

- TransitiveProperty
- SymmetricProperty
- FunctionalProperty
- InverseFunctionalProperty

These property characteristics can actually be instantiated directly, leaving the domain and range restriction to define the property as an object or datatype property:

```
<owl:FunctionalProperty rdf:ID="detectsDomain">
  <rdfs:domain rdf:resource="#Sensor"/>
  <rdfs:range rdf:resource="#SensorDomain"/>
</owl:FunctionalProperty>
```

The functional property `detectsDomain` is an object property, because the range restriction is applied to `SensorDomain` instances, and not XSD datatypes or literals. The notation used in the OWL Language Guide [27], utilizes the RDF `type` facility:

```
<owl:ObjectProperty rdf:ID="detectsDomain">
  <rdfs:domain rdf:resource="#Sensor"/>
  <rdfs:range rdf:resource="#SensorDomain"/>
</owl:ObjectProperty>
```
The above fragment defines the `detectsDomain` property in a way that both defines the property as a functional one, and restricts the range restriction to refer to an actual OWL class. The fragment is equivalent to the preceding definition of `detectsDomain`.

2.4.4 OWL Class Restrictions

One of the main advantages of OWL over RDF/S is its support for cardinality restrictions. This means that where RDF/S provides the mechanisms for restricting the possible subject and values of a defined property, OWL provides the ability to restrict the number of instances of that property within a class. Consider, for example, the following class re-definition:

```xml
<owl:Class rdf:about="#Sensor">
  <rdfs:subClassOf>
    <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="#hasOutput"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

An instance of the `Sensor` class is only valid if there is exactly one `hasOutput` property defined. The following cardinality restrictions are defined in OWL:

- `owl:cardinality`
- `owl:maxCardinality`
- `owl:minCardinality`

Whereas OWL DL and Full have no restrictions on how to use this constructs, OWL Lite can only restrict the cardinality value to being zero or one. This is done to make OWL Lite easier to process by machines with limited processing power.

Consider, for example, the following invalid sensor instance:

```xml
<Sensor rdf:ID="Sensor_Alpha">
  <hasOutput rdf:resource="#Track"/>
  <hasOutput rdf:resource="#Plot"/>
</Sensor>
```

By defining two `hasOutput` property instances, the class instance violates the cardinality restriction defined for the `Sensor` class. This is a powerful feature of OWL that makes it far more semantically expressive when compared to RDF/S.

2.5 Summary

Depending on the required expressiveness of an ontology language, there exist several alternatives. For Internet-based ontologies RDF seems to be a favourable solution. This is because RDF provides the ability to reference Internet-accessible resources within an ontology, making it possible to relate different concepts in different ontologies with each other.

In cases where the ontology language is required to provide constructs for defining class hierarchies of concepts, and for defining complex relationships, RDF Schema introduces several powerful features.

OWL goes beyond RDF and RDF/S by introducing powerful restriction constructs. This makes it possible to define ontologies that are logically sound, and can be
validated mathematically. In cases where cardinality restrictions are required to make ontology instances adhere to certain rules, OWL provides this, and several other interesting features.

In a semantic-based registry it is important that the stored information is valid. This is to ensure that query hits can actually be of some use to the querying party. For this reason the cardinality restriction features of OWL is a convenient feature, and thus OWL is the ontology language of choice for this report.

In this chapter I described some current ontology-based languages for describing concepts. In the next chapter I discuss methods as technologies for querying these ontologies.
3 Querying Ontologies

3.1 Introduction

Being able to represent complex semantics using ontologies is a huge step forward when it comes to reasoning about defined concepts. This requires a reasoning agent to be able to query the ontology content. With taxonomy-based ontologies a query is as simple as looking up a value in a table, but with RDF-based ontologies this gets more complicated. Since ontologies can be interrelated, the information about a concept can span many domains, and the amount of information can be excessive.

There are several languages available for representing ontology queries. There are pure Prolog-style languages using Description Logic clauses when querying the ontologies, and there are languages resembling more general database query languages. In [28] a template ontology is used for querying a datamodel by simply matching the instances defined in the template with the knowledge base.

Consideration has to be taken when choosing a query language depending on the type of underlying ontology language used (see Figure 10).

![Figure 10: Several alternative RDF query languages.](image)

Independently if OWL or RDF/S is used when defining an ontology, this ontology is represented by a pure RDF graph. RDF has no notion of transitivity, class creation and extensions, thus a query language needs to understand these vocabulary-specific extensions in order to be able to retrieve sensible query results. If a query language supports the ontology language used, the choice really boils down to the required level of expressiveness and usability.

Prolog-based languages provide the power of Description Logic clauses for reasoning about an ontology, something that delegates full formal support to the user agent. This, however, does make it less usable and is considered by this report to be too expressive, and is not considered. For template-based querying systems, the range of queries is limited to the expressiveness of the template. This may not be a downside, but it does limit the expressiveness available to the querying agent, and is considered too limited for this report. This report argues that RDQL [29] and OWL-QL [30] are both suitable languages, and are discussed in further detail in the following chapters.

3.2 Using OWL-QL for Querying Ontologies

The OWL Query Language (OWL-QL) for querying OWL-based ontologies is an abstract specification of a query language, which can come in many surface forms. In contrast to RDQL, which is based on pure SQL-style content retrieval, OWL-QL specifies a query session as a series of answer dialogues between a client and a server. The idea behind this is that OWL-QL is designed for the Semantic Web, and the Semantic Web is a complex information network, not comparable to relational databases.

Since the Semantic Web spans several schools of computer science, it is not the intention of the OWL-QL specification to provide a syntax suitable for all schools, but
rather several surface syntaxes of the abstract specification. This may result in concrete OWL-QL specifications in XML, SQL-like and Formal Logic syntax.

The OWL-QL specification does not only specify the query language, but also states that it should be utilized for querying OWL-based ontologies. This means that an OWL-based ontology needs no adaptation of the underlying RDF data-model, like pre-processing or on-demand Inferencing.

3.2.1 Answer Dialogues

Instead of SQL-style information retrieval that follows a request-response choreography, OWL-QL initiates a continuous dialogue between the client and the OWL-QL server. The reason for this is that the Semantic Web is the arena where OWL-QL is intended to operate. This means that a vast amount of information can be queried, possibly without a finite result set. This makes it necessary to deliver partial result-sets back to the client, and let the client be able to influence the ongoing process (see Figure 11).

![Figure 11: OWL-QL Dialogue.](image)

The client initiates a dialogue by sending a Query object to the server. The server responds with a bundle of answers, as well as a set of Process Handles. The set of process handles defines the possible actions the client can take, and is thus used by the client to influence the continuation of the dialogue. If the server cannot be queried anymore it can return an answer bundle containing termination tokens. If the client is satisfied with the received answers it can initiate termination of the dialogue.

3.2.2 Query Patterns

Querying in OWL-QL is considered asking the underlying knowledge base (KB) for sentences that adheres to a query pattern. The KB contains a collection of triples, in RDF terms. A query pattern is thus a series of triples which need to be matched against the KB. The result of a query is then the collection of triples in the KB that matches all triples in the query pattern. This does resemble the template-based ontology matching algorithm used in [28].

By utilizing variables, OWL-QL goes beyond simple template-based matching, and makes it possible to define query chains. Consider, for example, the following example query chain:

```java
( ?sensor, rdf:type, sensor:Sensor ),
( ?sensor, sensor:hasOutput, sensor:Track )
```
The first query triple matches with all triples in the KB that have the predicate `rdf:type` and the object `sensor:Sensor`. For each matched triple found, the following triple in the query chain is processed, with the value of the variable `?sensor` being the subject of the matched triple. If the whole chain is processed, any matching triple in the KB is considered an answer to the query. In OWL-QL all variables are named with a preceding question mark.

### 3.2.3 Variable Binding

One of the main advantages of OWL-QL over RDQL is how it handles variables. Since OWL provides constructs for defining cardinality restrictions, one should take this into consideration when querying ontologies. An OWL-QL variable can have one of the following properties defined:

- **Must-Bind**: If a value is not found for the variable the query fails to produce a result.
- **May-Bind**: If a value is found for the variable it is bound, otherwise the query continues without binding a value to that variable.
- **Don’t-Bind**: If a value is found for the variable the query fails to produce a result.

With a RDQL all variables are *must-bind* variables. For an example of the *may-bind* variables, consider the following ontology fragment:

```xml
<owl:Class rdf:about="#Sensor">
  <rdfs:subClassOf>
    <owl:Restriction owl:minCardinality="0" owl:maxCardinality="1"/>
    <owl:onProperty rdf:resource="description"/>
  </rdfs:subClassOf>
</owl:Class>
```

An instance of a `Sensor` class may or may not contain a `description` property. This follows from the `minCardinality` and `maxCardinality` restrictions specified. Consider the following two instances:

```xml
<Sensor rdf:ID="Sensor_Alpha">
  <description>This is sensor Alpha</description>
</Sensor>
<Sensor rdf:ID="Sensor_Beta"/>
```

If a query is issued that has a *may-bind* description variable the query will be able to find both `Sensor_Alpha` and `Sensor_Beta`. If the variable is *must-bind*, or RDQL is used, only `Sensor_Alpha` is found. The existence of optional properties, like in the example above, is quite common; therefore support for *may-bind* variables is a great feature when querying OWL-based ontologies.

### 3.2.4 Assumption Support

OWL does not contain any constructs for defining logical implications. With OWL it is only possible to reason about the actual *triples* defined in the ontology. In some cases it is desirable to query the ontologies in an *if-then* fashion. For example, consider the following query:

If `?sensor` is an acoustic sensor, what would its range be?

This query requires reasoning about a triple that may or may not exist in the ontology.
OWL-QL supports the definition of a premise when querying an ontology. The premise is a series of triples that are temporarily included into the ontology during the course of a dialogue.

3.3 Using RDQL for Querying Ontologies

RDQL is a data-centric query language for querying RDF data models. Being a data-centric query language means that there are no built-in abilities that do any reasoning that extends the triples contained within the ontology. The effect of this is that RDQL cannot be used directly on an OWL-based ontology. Since it does not have the ability to reason about OWL constructs like subClassOf, subPropertyOf and cardinality constraints.

In order to make RDQL a viable query language it is not the language itself that needs to be adapted, but the underlying data model through inference. For example, consider the following RDF/S-based ontology:

```xml
<rdfs:Class rdf:ID="Sensor"/>
<rdfs:Class rdf:ID="ActiveSensor">
  <rdfs:subClassOf rdf:resource="#Sensor"/>
</rdfs:Class>
<rdfs:Class rdf:ID="AcousticSensor">
  <rdfs:subClassOf rdf:resource="#ActiveSensor"/>
</rdfs:Class>
```

This ontology represents the following RDF graph:

```
#Sensor
   ^
   |
  #ActiveSensor
   v
#AcousticSensor
```

**Figure 12: RDF graph of subclass relation.**

This ontology model is made up by the following (relevant) data model triples:

```xml
( #Sensor, rdf:type, rdfs:Class )
( #ActiveSensor, rdf:type, rdfs:Class )
( #ActiveSensor, rdfs:subClassOf, #Sensor )
( #AcousticSensor, rdf:type, rdfs:Class )
( #AcousticSensor, rdfs:subClassOf, #ActiveSensor )
```

According to the OWL rules, the rdfs:subClassOf relation is a transitive relation. This means that, like in object-oriented programming languages, there is an implied rdfs:subClassOf relation between #AcousticSensor and #Sensor. Since RDQL operates on the actual triples in the data-model, this implied relationship is not found when queried for. If RDQL is used to query the ontology about whether AcousticSensor is a subclass of Sensor, the implied triple needs to be included into the data-model before executing the query:

```xml
( #AcousticSensor, rdfs:subClassOf, #Sensor )
```

RDQL has no knowledge about the meaning of the rdfs:subClassOf construct, and cannot make the connection on its own. It is up to the application to modify the data model to accommodate RDQL queries for various RDF dialects, for example, OWL and RDF/S. This is the both the advantage and the disadvantage of RDQL.
3.3.1 The RDQL Syntax

The syntax of RDQL resembles that of SQL [31]. This fits quite nicely with the content retrieval nature of RDQL. An RDQL query consists of the following parts:

- **SELECT**: Specifies the variables that need to be bound in the result-set. Since ontologies can be quite large, it is important to consider the needed variables in order to reduce memory requirements.
- **FROM**: Specifies the URI of the ontology used in the query.
- **WHERE**: Specifies the triples that are sought in this query instance.
- **AND**: Specifies boolean tests on the found triples. The result-set found using the WHERE-triples can be further reduced by including some boolean constraints on the triples.
- **USING**: Since RDF-based ontologies uses URIs to identify resources, RDQL queries can become quite bloated and unreadable. The USING section enables us to specify shortcuts for certain namespaces.

A variable in an RDQL query is represented in the same way as with OWL-QL, by a variable name preceded by a question mark:

```sql
?var
```

URIs in an RDQL query are represented in the following way:

```sql
<uri#resource>
```

If the URI has a defined shortcut in the USING clause, the following representation is used:

```sql
alias:resource
```

As an example, consider the following query:

```sql
SELECT ?sensor
FROM http://www.mil.no/sensors
WHERE (?sensor, rdfs:subClassOf, #Sensor )
```

The result-set should contain all instances that has a `rdfs:subClassOf` relation with `#Sensor`. In the above example the USING clause is not needed, since `rdfs` and `rdf` are built-in shortcuts. RDQL does not know how to resolve the `rdfs:subClassOf` relation, so its up to the application to modify the data-model.

3.3.2 Data-Model modification for OWL Queries

Using RDQL to query OWL-based ontologies requires modification of the underlying RDF data-model. This involves the insertion of additional triples in order to conform to the OWL rule-set. This process is often referred to as the *inference* process.

Consider the graph illustrated in Figure 13. The goal of the inference process is to insert extra triples to satisfy the implied `rdfs:subClassOf` relation between `#AcousticSensor` and `#Sensor` (see Figure 13).
A Peer-to-Peer Distributed Semantic Service Registry

Figure 13: RDF Graph with implied OWL Inference step.

The dotted line represents a triple that is inserted into the data-model as a result of inference, and is not directly defined in the ontology document itself.

There are two Inference-modes identified, namely:

- **Full Inference**: This involves performing all relevant inference steps before processing any queries. This can be done upon submittal to the ontology repository.
- **On-Demand Inference**: This involves tight integration between the query-engine, and results in temporary insertions of triples that are required for the current query.

**Full Inference Mode**

Using Full Inference can result in extremely large data-models. All possible rule-based triple insertions are performed for both the ontology itself, and the imported ones. The advantage of this approach is that the inference step is only performed once, during initial submittal of the ontology to the repository, and thus are the performance penalties of future queries eliminated. Figure 14 illustrates how the query-engine, regardless of the query language, is layered right on top of the data-model.

**On-Demand Inference Mode**

Using On-Demand inference requires a tight integration between the query engine and the data-model. In contrast to Full Inference mode, there are no inference steps being performed upon submittal of the ontology to the repository. Instead, triples are inserted when the nature of the query itself requires it.

By layering an Inference Engine between the Query Engine and the Data-Model, the query is not resolved directly on the data-model, but is first after the insertion of required temporary triples (see Figure 15).
3.4 Summary

When choosing a suitable query language or system several aspects have to be considered, depending on the context. For ease of use RDQL is a popular choice. RDQL has been designed for querying RDF ontologies, and does not support RDF/S or OWL constructs directly. This is a matter of implementation details, and the underlying data-model can be modified using inference in order for RDQL to be a viable solution for these ontology-languages as well. OWL-QL introduces a sophisticated dialogue-based query process that is very powerful, but may be too advanced for content matching applications. It does, however, possess an advanced variable binding feature that is very handy when querying OWL-based ontologies utilizing cardinality constraints.

For this report OWL is the ontology language of choice, thus OWL-QL is the most suitable query language.

In this chapter I discussed methods for querying or reasoning about RDF-based ontologies- In the next chapter I discuss how ontologies can be used to describe services based on their capabilities.
4 Representing Services using Ontologies

4.1 Introduction

Being able to represent service interfaces in terms of messages and datatypes is something that is already possible through technologies such as CORBA Interface Definition Language (IDL) [32] and the Web Services Description Language (WSDL) [04]. These technologies provide us with rich languages for describing message exchange and protocol bindings, but are limited to the actual interface, and provide no features for describing service capabilities.

WSDL defines a clear distinction between interface description and protocol bindings. This is a sound idea, since it abstracts the public interface of a service from its various protocol bindings. Thus, a service can have several methods of access, but only one public interface description (see Figure 16).

![Interface Description](image)

**Figure 16: Separation between Interface description and protocol bindings.**

The advantage of this approach is that using the WSDL document, a service can be invoked, regardless of its implementation, as long as there exists a binding for the given access protocol. This approach is worth considering when defining an ontology-based service description language.

The lack of capability definition support in WSDL is somewhat remedied when looking at the initial intention of WSDL document as a service description accessible via a UDDI [02] registry. UDDI has some support for relating a service description with some meta-data. This support is, however, limited and very business oriented. UDDI provides the ability to associate a service with a taxonomy code, but does not enable this association to be used with an ontology reasoner.

Using ontologies, for example based on OWL, to describe a service, enables us to define service capabilities that go beyond the limited taxonomy codes utilized in UDDI, and to describe services using detailed concepts, with logic restrictions and constraints. This makes it possible to reason about the meaning of ontology instances, thus making it possible to reason about whether a service's capabilities fit the requirements present. OWL-S is as the name suggests an OWL-based ontology for services, and allows one to define services in a semantically sound way using OWL. This chapter provides an introduction to using OWL-S for describing services, and in addition a series of guidelines are given of good practise with this regard.

4.2 Using OWL-S for Service Representation

OWL-S [06] makes it possible to represent services in a way that allows both easy reasoning on the capabilities of a service, as well as the ability to invoke methods based on the information contained within the service representation. Like WSDL, OWL-S has a clear distinction between the interface specific information and the protocol bindings. In addition it provides constructs for defining the process behaviour of services, much like BPEL4WS [33]. Three parts make up an OWL-S service representation:
• **Profile**: The profile provides meta-data concerning the service in question. The profile describes what the service does, and should provide enough information to confirm to a service seeking party whether the service in question is suitable. The Profile encapsulates the **capability-specific** information about the service.

• **Process model**: The process model describes how the service works. This involves what messages need to be conveyed, the sequence of actions, pre- and post-conditions and several other aspects. The process model can also be inspected by a service seeking party if a more in-depth analysis is required.

• **Grounding**: The grounding specifies how a user can access the service. This involves description of the communication protocol used, location and other context information needed to locate the service. The Grounding is actually **abstract**, and requires a WSDL document to provide the concrete protocol bindings.

### 4.3 OWL-S Document Structure

An OWL-S service description is an OWL ontology. Although not a requirement, it is common practice to divide an OWL-S service description into four separate documents. These are the profile, process model, grounding and a **master service** document that ties them all together (see Figure 17).

![Figure 17: Structure of OWL-S Service Description.](image)

The **grounding** consists of the abstract grounding OWL-S document part and a concrete link to a WSDL interface description. OWL-S only provides the WSDL grounding by default, but it is open to other concrete groundings. Since a WSDL document is not a valid OWL document, it does not serve as a direct member of the OWL-S ontology, but can be referenced from the grounding part.

**Guideline 1:**

Always separate an OWL-S ontology into four separate documents, namely Service, Profile, Process Model and Grounding.
4.4 The Service Ontology

The Service ontology part of an OWL-S service description should not contain much information. With OWL there is a lot of freedom when it comes to defining ontologies, but since these ontologies need to be searchable, some ground rules have to be followed. The Service part should only contain OWL-S service instances:

```xml
<service rdf:ID="Radar-78_SERVICE">
    <presents rdf:resource="&profile;Radar-78_PROFILE"/>
    <describedBy rdf:resource="&process;Radar-78_PROCESS"/>
    <supports rdf:resource="&grounding;Radar-78_GROUNDING"/>
</service>
```

The service instance initializes the service, and relates the service description with its profile, process model and grounding. This is done with the logically named object properties; presents, describedBy and supports, respectively. These are functional properties and tie the parts together.

It is possible to define sub-classes of the service class and create a hierarchy of service types. This report, however, argues that it is a good idea to limit the responsibility of the Service part to only tie the service description together. All other responsibilities should be delegated to the profile, process model and grounding part.

**Guideline II:**

The Service part of an OWL-S service representation should only be responsible for tying the ontology parts together. All other responsibilities should be delegated to the Profile, Process Model and Grounding part.

4.5 Service Profile

The service Profile part of an OWL-S service description ontology is designed to be searchable. This means that if an agent needs to determine whether a service fits certain requirements, the profile part is processed. All information concerning a service's capabilities and context should be included in the profile. Much like the way UDDI registries are queried for suitable WSDL descriptions of services, an OWL-S profile is queried when searching for suitable OWL-S described service.

A service profile itself is created by instantiating a Profile class, from the OWL-S Profile ontology namespace:

```xml
<profile rdf:ID="Radar-78_PROFILE">
    <service:presentedBy rdf:resource="&service;#Radar-78_SERVICE"/>
    ...
</profile>
```

The presentedBy property, from the OWL-S Service namespace, relates the profile instance with the service instance in the Service part of the OWL-S service representation. The property is isomorphic, meaning that a service can only have one profile and a profile can only relate to one service.

It is possible to directly instantiate the profile class, like in the example above, but if a hierarchy of service types is needed, sub-classing the profile class is the way to go. This makes it possible to define groups of service types, with their own properties and relations. Taking considerable care when defining the profile hierarchy, searching for services may be easier.
Guideline III:
Define, or reuse existing, sub-classes of the Profile class to enable class based service queries.

Consider the following ontology fragment:

```xml
<owl:Class rdf:about="#Sensor">
  <rdfs:subClassOf rdf:resource="&profile;#Profile"/>
</owl:Class>
<owl:Class rdf:ID="Radar">
  <rdfs:subClassOf rdf:resource="#Sensor"/>
</owl:Class>
```

Every instance of #Sensor or derivatives thereof, are OWL-S service profile constructs. This makes it possible to design the service ontologies as a typical object-oriented structure. Consider, for example, the following instance of a Radar OWL-S profile:

```xml
<Radar rdf:ID="Radar-78_PROFILE">
  <service:presentedBy rdf:resource="&service;#Radar-78_SERVICE"/>
...
</Radar>
```

By instantiating the profile as a Radar sub-class, a querying agent that specifically needs a Radar can find it by searching for rdf:type properties with value #Radar. This is a direct result of the RDF/S subclassing feature.

4.5.1 Non-Functional Profile Properties

An OWL-S Profile instance provides, much like UDDI, several standard properties for describing non-functional aspects of the service. These properties include:

- `serviceName`, which is a required datatype property relating the profile with an XSD string defining the service's name.
- `textDescription`, which is a required datatype property relating the profile with an XSD string providing a textual description of the service.
- `contactInformation`, which is an optional object property relating the profile with a contact information object. The type of contact information is totally up to the author.
- `serviceCategory`, which relates the profile with well-known taxonomy codes.

These properties fit in nicely with the fixed meta-data field defined for UDDI registries. In [12] a mapping is actually provided between the OWL-S properties and the UDDI meta-data fields. Thus, if an OWL-S service description is required to comply with the defined framework of OWL-S, this mapped UDDI registry can actually be used to host OWL-S service descriptions.

4.5.2 Parameter support

There is support for defining parameters in the service profile. These definitions are simply links to the parameter definitions in the process model, but may be of some help to searching agents. For template-based query systems [28] the OWL-S Profile alone is queried when searching for suitable services. By providing links to the parameters defined in the process model, such systems are able to query for services that provide certain input and output value types.
There are four concrete parameter types defined within the OWL-S Profile ontology, namely:

- hasInput;
- hasOutput;
- hasCondition;
- hasEffect.

Consider the following example:

```xml
<Radar rdf:ID="Radar-78_PROFILE">
  <service:presentedBy rdf:resource="&service;#Radar-78_SERVICE"/>
  <profile:serviceName>Radar-78 Service</profile:serviceName>
  <profile:textDescription>
    This is the control service of the Radar with identification name "Radar-78".
  </profile:textDescription>
  <profile:hasInput rdf:resource="&process;GetTarget_INPUT"/>
  <profile:hasOutput rdf:resource="&process;GetTarget_OUTPUT"/>
</Radar>
```

The hasInput and hasOutput properties reference the parameters defined in the process model ontology.

If the process ontology defines several processes, the parameter links in the profile ontology are not directly able to discriminate between the parameters belonging to a single process. This can lead to a positive query match if the requested parameters can be found on several distinct processes. This is not an ideal situation.

### 4.6 Process Model

The Process Model describes the processes that the service supports. The process ontology defines constructs that help coordinate the execution of the service methods. The process ontology is highly influenced by the Business Process Execution Language for Web Services (BPEL4WS) [33]. The ontology defines which service methods are available as processes. A process model is instantiated in the following way:

```xml
<process:ProcessModel rdf:ID="Radar-78_PROCESS">
  <service:describes rdf:resource="&service;#Radar-78_SERVICE"/>
  <process:hasProcess rdf:resource="#GetTrack"/>
</process:ProcessModel>
```

The service:describes property completes the symmetric behaviour of the service:describedBy property used in the Service definition. The process:hasProcess associates the process model with one atomic or composite process.

If the process:hasProcess points to an atomic process, this means that the service in question only has a single service primitive defined. If there are several service methods available, the service model must point to a composite process that controls how a user can choose between the methods.

#### 4.6.1 Atomic Processes

An Atomic Process is the most basic type of service primitive. It defines a method that may take inputs and deliver outputs upon completion. An atomic process is also
subject to defined pre-conditions and effects. In order to be invoked, an atomic process needs to be associated with a process grounding. This association is not mentioned in the process model ontology itself, but is defined in the service instance through the service:supports property. An atomic process is instantiated in the following way:

```xml
<process:AtomicProcess rdf:ID="GetEnemyTargets">
  <process:name>GetEnemyTargets</process:name>
  <process:hasOutput>
    <process:UnConditionalOutput rdf:ID="GetEnemyTargets_OUTPUT">
      <process:parameterType rdf:resource="&sensors;#TargetList"/>
    </process:UnConditionalOutput>
  </process:hasOutput>
</process:AtomicProcess>
```

For information purposes an atomic process can have a defined name, specified by the process:name property. In addition, the parameters associated with the atomic process are defined in terms of the following four properties, which can be referenced from the service profile (see section 4.5.2):

- hasInput
- hasOutput
- hasCondition
- hasEffect

The hasInput and hasOutput properties describe instances of a process:Parameter sub-class. The sub-classes defined in the process model ontology are Input, ConditionalOutput and UnConditionalOutput. The use of conditions and effects is not covered in this report, thus only Input and UnConditionalOutput will be considered.

### 4.6.2 Composite Processes

Composite processes are a collection of sub-processes that can be executed as defined by the process:composedOf property. The process:composedOf property is required in a composite process instance, and associates the process with a control structure containing zero or more sub-processes. Consider the following example:

```xml
<process:CompositeProcess rdf:ID="ServicePrimitives">
  <process:composedOf>
    <process:Choice>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess rdf:about="#GetEnemyTargets"/>
        <process:AtomicProcess rdf:about="#GetFriendlyTargets"/>
        <process:AtomicProcess rdf:about="#GetUnidentifiedTargets"/>
      </process:components>
    </process:Choice>
  </process:composedOf>
</process:CompositeProcess>
```

The process:composedOf property has a control structure of type process:Choice as its value. This control structure defines how a user can invoke the composite process. For a Choice control structure the user can choose from any process within the process:components property collection, and execute them. Thus for a service that provides several methods, such a construct should be associated with the process model.
In addition to the *process:Choice* control structure, there are several other structures defined in the process ontology:

- **Sequence**: A series of sub-processes are executed in sequence, where the first process in the sequence handles the input, and the last one serves the output. The output of one sub-process becomes the input of the next one in the sequence.

- **Split**: A collection of sub-processes that can be executed concurrently.

- **Split-Join**: Partly concurrent execution.

- **Unordered**: The sub-processes can be executed in an un-orderly fashion, or even concurrently.

- **If-Then-Else**: Typical conditional execution of two sub-processes.

- **Iterate**: Iterates over the collection of sub-processes. It is not clear what the difference with sequence is. The difference is probably that with iterate, the input/output handling is done at each sub-process.

- **Repeat-While**: Conditional repeated execution of a process.

- **Repeat-Until**: Conditional repeated execution of a process.

For an illustration of the *Sequence* control structure, consider Figure 18.

**Figure 18: Sequence control structure.**

Figure 18 shows an OWL-S process instance that defines a composite process that can determine the origin of an enemy flight using its ID. This process can be defined as a sequence of existing atomic processes. The atomic process *GetTrack* returns a target-track given an enemy identifier. This track is supplied as the input to the *CreatePlot* process, which creates a plot-map of the targets location history. This plot is supplied as the input to the *DetermineOrigin* process, which analyzes the plot to determine the targets origin.

### 4.7 Service Grounding

The *Service Grounding* ontology is used to define the concrete process groundings. This follows the approach taken by WSDL [04] where the service interface is abstracted form the, possibly several, concrete protocol bindings. The grounding ontology can use a WSDL document to define the concrete protocol bindings (see Figure 19).
A WSDL document is an XML-based document, and not an RDF-based ontology, thus it is not, directly, possible to reason about the content of this WSDL document itself. This is, in my opinion, a shortcoming of the OWL-S specification. Consider, for example, a scenario where a user needs to locate a service that is reachable by a specified protocol. This information can only be obtained by searching for services that provide the required functionality, and then examining WSDL groundings in order to locate the ones with the desired protocol bindings.

The rationale behind the usage of WSDL as the concrete protocol binding lies in WSDL does that very well. It is no use inventing the wheel all over again. There is already a lot of tool-support for WSDL, and thus the advantages outweigh the disadvantages. However, the OWL-S community could provide an RDF/S [09] version of WSDL, which maps perfectly on the existing XML version. This would make the WSDL grounding fully integrated into the OWL-S ontology.

The OWL-S ontology is open for other grounding alternatives through the use of sub-classing. In the current version of the OWL-S ontology there is only one sub-class of the service:ServiceGrounding defined, namely grounding:WsdlGrounding. This grounding is instantiated in the following way:

```xml
<grounding:WsdlGrounding rdf:ID="Radar-78_GROUNDING">
  <service:supportedBy rdf:resource="&sensor_s;#Radar-78_SERVICE"/>
  <grounding:hasAtomicProcessGrounding rdf:resource="#GetEnemyTargets"/>
  <grounding:hasAtomicProcessGrounding rdf:resource="#GetFriendlyTargets"/>
  ...
</grounding:WsdlGrounding>
```

The service:supportedBy property makes up the symmetric nature of the service:supports property defined in the service ontology. For every atomic process defined in the process ontology, the grounding ontology may support a concrete binding by including a grounding:hasAtomicProcessGrounding property instance.

### 4.7.1 Process Binding

The grounding:hasAtomicProcessGrounding property initializes the grounding for a specified atomic process by referencing, for example, an instance of the grounding:WsdlAtomicProcessGrounding class:

```xml
<grounding:WsdlAtomicProcessGrounding rdf:ID="GetEnemyTargets">
  ...
</grounding:WsdlAtomicProcessGrounding>
```

The grounding:WsdlAtomicProcessGrounding relates, or mediates, between the atomic process definition of the process ontology, and the related WSDL document.
Without going into too much detail regarding the WSDL specification, an atomic process is related to an operation of a given *PortType* (interface), as defined in the referenced WSDL document:

```xml
<grounding:WsdlAtomicProcessGrounding rdf:ID="GetEnemyTargets">
  <grounding:owlsProcess rdf:resource="&sensor_pm;#GetEnemyTargets"/>
  <grounding:wsdlOperation>
    <grounding:WsdlOperationRef>
      <grounding:portType>&sensor_wsdl;#SEN1_PortType</grounding:portType>
      <grounding:operation>&sensor_wsdl;#getEnemyTargets</grounding:operation>
    </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>
</grounding:WsdlAtomicProcessGrounding>
```

The property pair `grounding:owlsProcess` and `grounding:wsdlOperation` is responsible for associating the abstract atomic process definition in the process ontology with the abstract operation definition in the WSDL document. The `grounding:WsdlOperationRef` is necessary to define a unique association through the pair `grounding:portType` and `grounding:operation`. The actual concrete protocol binding of the operation is handled within the scope of the WSDL document, and does not involve OWL-S inference. This results in two degrees of freedom:

1. The grounding ontology can be bound to WSDL and other description languages, for example *Universal Plug and Play* (UPnP) [70].
2. The WSDL document can have several protocol bindings.

### 4.7.2 Parameter Binding

The main problem when it comes to concrete grounding of processes, is the type system. OWL-S is built on OWL and therefore a process definition may contain input and output parameters that rely on OWL objects and datatypes. In contrast, the described concrete services most likely do not support semantically rich datatypes, like OWL. This problem is by no means inherent only to OWL-S. SOAP [34], which is widely used in Web Services [03], was specifically designed to make type conversion effortless. SOAP, and to a certain degree WSDL, relies on XML Schema datatypes (XSD). This means that, using SOAP, parameters are translated between some native form, for example OWL, and its XML Schema equivalent (see Figure 20).

![Figure 20: SOAP-based datatype conversion.](image)

With OWL-S-based service descriptions the problem gets even more complicated, since there has to be two rounds of translation on both sides of the wire. First the
native type has to be converted into OWL constructs. Then if SOAP is used as the wire protocol, these constructs have to be translated into XML Schema equivalents.

Where OWL represents structures using owl:Class instances, SOAP-bound services will support XML Schema complexType constructs. This problem manifests itself when binding OWL-S parameters to WSDL message parts.

For situations where the service supports OWL datatypes, the WSDL extension mechanism can be used to specify the type of a message part, and only one round of translation is needed:

```xml
<message name="GetEnemyTargetsOutputMessage">
  <part name="targetList" owl-s-wsdl:owl-s-parameter="process:#GetEnemyTargets_OUTPUT"/>
</message>
```

Instead of determining the type of the message part through the usual type or element attribute, the WSDL specification does provide the ability to define extensions, like the owl-s-wsdl extension. The grounding:wsdlInputMessage and grounding:wsdlOutputMessage properties associates a WSDL message with the current grounding:WsdlAtomicProcessGrounding instance:

```xml
<grounding:WsdlAtomicProcessGrounding rdf:ID="GetDomain_Grounding">
  ... 
  <grounding:wsdlOutputMessage>
    &sensor_wsdl;#GetEnemyTargetsOutputMessage
  </grounding:wsdlOutputMessage>
  ... 
</grounding:WsdlAtomicProcessGrounding>
```

In most cases, however, services are not designed to support OWL datatypes by default. In these cases RDF-based OWL datatypes need to be translated into XML Schema complexType instances. The OWL-S grounding ontology does support this through XSLT transformation mapping.

### 4.7.3 WSDL Message Mapping

Every OWL-S parameter of a given atomic process needs to be bound to a message part of the message defined using the grounding:wsdlInputMessage or grounding:wsdlOutputMessage properties. This binding is done using the grounding:wsdlInputs and grounding:wsdlOutputs properties:

```xml
<grounding:wsdlOutputs rdf:parseType="Collection">
  <grounding:WsdlOutputMessageMap>
    <grounding:wsdlMessagePart>
      &sensor_wsdl;#trackList
    </grounding:wsdlMessagePart>
  </grounding:WsdlOutputMessageMap>
  ... 
</grounding:wsdlOutputs>
```

In the example above the grounding:wsdlOutputs property contains a grounding:WsdlOutputMessageMap instance, which is used to map a single WSDL message part with an OWL-S parameter from the process model. This mapping can be done directly if the service supports OWL datatypes. If this is not the case, XSLT translations can be used to translate between OWL and its XML Schema equivalent.

For each WSDL message part, a mapping has to be provided through the grounding:wsdlMessagePart property. When mapping to an XSLT resource, there are two possible solutions: An external XSL document is referenced using the
**Guideline IV:**

WSDL message part mapping using XSLT should avoid using external stylesheets in order to reduce the transformation time.

If the XSL stylesheet is included in the grounding instance, using the `grounding:xsltTransformationString` property, this should be done within a CDATA field to avoid interfering with the RDF-aspects of the ontology.

### 4.8 Summary

OWL-S as a service description language provides a process model that is heavily influenced by BPEL4WS. In addition it uses WSDL for concrete process grounding, thus so far nothing new. What makes it stand out is its profile ontology, which goes beyond UDDI registries when it comes to describing service capabilities. The profile ontology is responsible for defining service capabilities using the expressiveness of OWL.

Using OWL it is possible to define profile hierarchies which can make distributed searching for services far more structured. Searching an OWL-S service description is not limited to the profile itself, but the user has access to the profile, process model and the service grounding. This makes OWL-S a powerful, fully integrated service description language.

In this chapter I discussed how OWL-S can be used to describe services in terms of their capabilities and processes. In the next chapter the scope is shifted towards the actual registry platform, and various strategies for distributing registries are discussed.
5 Distributed Registries

5.1 Introduction

In the proposed distributed semantic service registry one of the main design goals is to avoid centralization. This goal is in contrast with the general notion of a registry, which is that of a centralized source of information. Based on the requirements of this report, the distribution requirement follows from security and robustness considerations, rather than load-balancing and other performance-enhancing features that registry distribution brings. Thus, although there are several solutions to distributing a registry for performance reasons, there appears to be little work done in that regard when it comes to security and robustness.

Consider, for example, the UDDI version 3 [35], registry specification, which introduces the Multi-Registry Environment. This approach makes it possible to distribute a UDDI registry across several hosts. This seems like a great idea, and even more when considering that there is work in progress to integrate OWL-S service descriptions with UDDI [36], but there is a severe drawback. The UDDI Multi-Registry Environment is based on the idea of a root registry with several affiliate registries. This root registry is a major centralized vulnerability, or point-of-attack, and does not fulfill the requirements of this report.

Another interesting approach is that taken by the Jini Network Technology [37]. Here a host on the network initially has no reference to the registry, or Lookup Service. These references are obtained by sending out a multicast-based message on the network, to which available registries may answer. This is interesting since as long as there are registries available on the network, the system keeps on working. There are, however, two main drawbacks in the approach taken by Jini. The first is that Jini is Java- [26] and TCP/IP [38] based, something that seriously impacts interoperability at the level required by this report. The second drawback is that even though there are several registries that need to be taken out in order to corrupt the system, they still stand out as centralized points of intelligence, although not as much as with UDDI.

Another emerging technology is Peer-to-Peer networking. A popular technology when it comes to file sharing, but it has features that are usable for many more purposes. This report argues that Peer-to-Peer networking is a suitable tool for distributing a registry.

5.2 Peer-to-Peer Networking

Sean Fanning designed Napster [39] for file sharing. Instead of downloading files from a centralized web server, Napster was designed to make computers exchange files directly with each other. Instead of having several clients being dependent of a couple of powerful centralized servers, Napster introduced a network of hosts, or peers, that were all servers.
A network of equal servers sharing resources resembles the initial design of the Internet, where mainframes where interconnected with equal responsibilities. Thus the idea of Peer-to-Peer networking is far from new, and not invented by Napster, although it was the first major application in the modern Internet applying this paradigm.

Webopedia [40] defines a Peer-to-Peer network as:

“...A type of network in which each workstation has equivalent capabilities and responsibilities. This differs from client/server architectures, in which some computers are dedicated to serving the others."

Napster cannot be considered as a pure Peer-to-Peer network, because the network included several special purpose peers. This is also the case with most current Peer-to-Peer network applications. Perhaps the most valuable feature of a Peer-to-Peer network is that a dynamic network is implied. Peers can come and go as they please, without affecting the network. This is a shift from normal DNS [41] and IP-based [42] networks, which are rather of static nature.

5.3 Jxta-based Peer-to-Peer Networking

With Peer-to-Peer applications like Napster [39], Gnutella [17], ICQ [43] and Kazaa [44] there is a major problem. They all follow the Peer-to-Peer programming model, but they use their own custom protocols for communications, something that makes cross-application interoperability only possible by means of protocol translators (see Figure 22).

Figure 22: Cross-application interoperability using translators.

With the rapid expansion of Peer-to-Peer based applications, writing translators between them can become a time-consuming task. Project Jxta [18] tries to solve interoperability issues by supplying an open, XML-based protocol. It is designed to be both platform- and network independent, and maybe more importantly, generic.
While Napster is a music sharing application, Gnutella a generic file sharing application and ICQ an instant messaging service, Jxta is designed to be a universal generic Peer-to-Peer technology, which can be used for all purposes and run on devices ranging from mainframes to light switches.

5.3.1 Jxta Concepts

Identifiers

In Jxta concepts like Peers, PeerGroups, Modules and others are identified using UDDI identifiers [47]. Since a Jxta connected Peer can be reached via several network protocols, a network specific addressing scheme is not desirable. A network independent addressing scheme is utilized by Jxta, which involves using UDDI identifiers in a URN [46]. Consider, for example, the following identifier of a Jxta Peer:

urn:jxta:uuid-69D604A7E39D43ABB3628A25BF58B3A002

Peers

A Jxta peer is any networked device implementing the core Jxta protocols. A unique UDDI ID identifies each peer, and each peer all may publish services and resources on the network. Dependencies between peers depend entirely on the peers themselves, but peers are autonomous by nature. In order for each peer to be able to communicate, a peer endpoint is identified for every peer, representing its Network Interface Card (NIC).

Peer Groups

“Peers self-organize into Peer Groups. A peer group is a collection of peers that have a common set of interests.” [45] Each group is identified by a unique UUID group ID, and can be published and discovered on the Jxta network. The physical division of a network does not dictate the forming of a peer group, but it may prove useful to keep the underlying topology in mind when forming a group. Jxta recognized three common motivations for creating a group:

- Creating a secure environment.
- Creating a scoping environment. Peers with similar interest are logically grouped together.
- Creating a monitoring environment.

Jxta defines several PeerGroup services. These are services that apply to members of a peer group, or peers that wish to join a group. In order to belong to a group, a peer has to apply to the group’s membership service. The membership service can be tailored to suit the specific joining rules of a group.

Network Services

Jxta can be used by peers to publish, discover and invoke network services. However, Jxta does not dictate how these services are defined. This is maybe one of the most valuable features of Jxta, namely its flexibility. A peer can discover a network service using the Discovery Protocol. Once found, the specification of the service can also be obtained, proxies can be generated and then the peers may invoke methods on that service. The format of the service specification may come in
many forms, i.e., WSDL, ebXML, IDL, Java interface, etc. Networked services can be devided into two separate groups:

1. PeerGroup Services are services provided by a specific PeerGroup. These services can in general only be used by PeerGroup members.

2. Peer Services are services provided for the whole Jxta network.

Advertisements

All network resources are represented in Jxta with a language neutral meta-data structure, called an advertisement. The Jxta specification specifies several advertisements, but there is nothing stopping one from adding custom ones, either from scratch or by sub-classing existing ones.

Because of the extensive use of advertisements, considerable care has to be taken when specifying one. By reducing the supplied infoset to the absolute required minimum, and resorting to caching, serious performance gains can be realized.

Pipes

Pipes are virtual communication channels used to send and receive messages between endpoints. A pipe uses virtual mailboxes in order to create the illusion of a location and topology-independent link. The quality of service of a pipe depends on the implementation, for example:

- Uni-directional asynchronous pipe (required)
- Synchronous request/response (required)
- Bulk transfer
- Streaming
- Secure

The binding of a pipe to two endpoints is carried out by the Pipe Binding Protocol.

Messages

The information transferred using pipes is represented using XML. A message contains an envelope element as its root, containing several other XML elements. This approach is compliant with the W3C XML Protocol standards [22], and is similar to that of SOAP [34].

5.3.2 The Jxta Architecture

In the Jxta architecture each participant is referred to as a peer, something that follows typical Peer-to-Peer terminology. Each peer is identified by a unique UUID [47] identifier, which is the Jxta substitute for a DNS [41] domain name. Peers sharing common interest, for example as participants of the same Jxta-based application, may be grouped together in a unique PeerGroup.

Jxta is designed to be network-independent, a functionality that is provided by the Endpoint Service. The endpoint service makes it possible for a message to be sent from one peer to another, across multiple network bearers spanning several sub-networks, without revealing this complexity to the service user (see Figure 23).
A Peer-to-Peer Distributed Semantic Service Registry

Figure 23: Endpoint Service.

Using the endpoint service a Jxta-peer is able to send a message to a UUID identified peer, across multiple network subnets, without knowing the network address of its target. The UUID identifier is the target address of the message, and as long as its temporary host is reachable, it can receive the message. This is quality is quite appealing in highly dynamic network situations where a host may change network bearer without warning.

In addition to the endpoint service Jxta provides several XML-based protocols, where some are compulsory and others optional. The protocols that are important for this report are:

- **Endpoint Routing Protocol**, which is responsible for routing a message between peers.
- **Rendezvous Protocol**, which is responsible for propagating messages within a Jxta PeerGroup.
- **Peer Resolver Protocol**, which provides a generic query service.

The **Peer Resolver Protocol** provides the basic functionality for realizing a Jxta-based distributed registry. The Peer Resolver Protocol depends on the **Endpoint Routing Protocol** and the **Rendezvous Protocol** to function.

### 5.3.3 The Peer Resolver Protocol

The **Peer Resolver Protocol** provides a basic request-response communication model for resolving queries on a peer-to-peer network. The protocol is designed to be application specific, thus it uses the **Rendezvous Protocol** to limit the propagation of a query to stay within a PeerGroup.

A query is initiated by propagating a **ResolverQuery** message using the **Resolver Service** of the target PeerGroup. A ResolverQuery message is an XML document that is validated against the following XML Schema definition:

```xml
<xs:element name="ResolverQuery">
  <xs:complexType>
    <xs:element name="Credential" type="xs:anyType" minOccurs="0"/>
    <xs:element name="SrcPeerID" type="xs:anyURI"/>
    <xs:element name="HandlerName" type="xs:string"/>
    <xs:element name="QueryID" type="xs:string"/>
    <xs:element name="Query" type="xs:anyType"/>
  </xs:complexType>
</xs:element>
```

The **Resolver Service** does not guarantee that query results arrives in the same order as the query requests. For this reason it is up to the query application to supply each ResolverQuery message with an interpretable value for the **QueryID** element, in addition to the **Query** element itself.

A ResolverQuery message does not need to have a target Peer address, since it is propagated to all peers within a PeerGroup, but instead, it is directed at an application specific query handler (see Figure 24).
Figure 24: ResolverQuery directed at query handler "OWL-QL".

When a ResolverQuery message is received at a Peer, the query is forwarded to a registered handler for the specified handler name. If the peer does not provide a handler for the specified handler name, the query is ignored. If the query handler requires some sort of authentication, the ResolverQuery message may include a Credential element with the required authentication credentials. When a query handler is able to produce a response to the query, the response is sent directly back to the requesting peer, using the SrcPeerID element to identify it.

5.4 Summary

When designing distributed solutions without any source of centralization, few available solutions can be found. The question is to what extent centralization must be avoided. Jini is fully decentralized, but requires dynamic allocated Lookup Services, which do form temporary centralized entities. In addition Jini relies on Java, which is a drawback since Java is not yet available for all types of devices.

Peer-to-Peer networking is similar to Jini, in that it provides a dynamic network of peers. By using Jxta as the realizing Peer-to-Peer technology, all centralization can be avoided. The protocol is XML-based and thus does not rely on any specific technologies. Jxta provides a generic query resolver service and in addition makes it possible to group peers into communities. These are features that provide the building blocks for a distributed registry.

In this chapter I introduced several strategies for distributing a registry application. In the next chapter, the discussed technology solution of this and previous chapters are used in order to define a proposed conceptual architecture of a semantic, distributed service registry.
6 A Proposed Semantic, Distributed Service Registry

6.1 Introduction

In the design of a semantic distributed service registry there are two main problem domains that have to be defined: the first is the semantic representation of services, and the second is registry distribution. During the initial research for this thesis, the Web Ontology Language (OWL) and its service ontology [06] were considered as a suitable solution for the semantic representation of services. For the distribution of registries this thesis has been influenced by [16], where the Gnutella [17] Peer-to-Peer technology is suggested as an enabling technology for creating a distributed service registry. This chapter gives an overview of a conceptual architecture of a semantic, distributed service registry, which uses OWL-S [50] for semantic representation of services and Peer-to-Peer networks for distribution.

In [12] a UDDI [02] registry is modified by mapping standard UDDI meta-data fields to properties in the OWL-S ontology. In this way it is possible to use a UDDI registry in order to host semantic OWL-S service descriptions. There are, however, two main disadvantages when considering a UDDI-based solution for providing a semantic, distributed service registry. First, the distribution support that UDDI provides is based on a Multi-Registry Environment where an upper-level root registry delegates responsibility to several affiliate registries. The root registry serves as a synchronization server and is a centralized element within the architecture, something that does not fulfill the requirements of this thesis. The second shortcoming follows from the mapping of UDDI meta-data fields to OWL-S profile ontology properties. The rationale behind this mapping is that a service should be searchable via the standard UDDI search interface. Doing this, however, limits the expressiveness of service queries to stay within the boundaries of the standard UDDI meta-data fields.

6.2 The Architecture

For the proposed semantic distributed service registry a peer-to-peer based architecture is considered, where each service provider hosts its own OWL-S service description (see Figure 25). This approach is also taken in [16], but with DAML-S [07] and Gnutella as the enabling technologies.

![Figure 25: Peer-to-Peer distributed service registry.](image)

Every host that provides a service can join the Peer-to-Peer network as a peer, and provide an OWL-S service description of all its hosted services to be queried via the
network. A user that needs to find a service using the proposed architecture can post a request on the peer-to-peer network and wait for a possible response. Based on this the proposed architecture can be divided into two separate services:

- The *Query Distribution Service*, which is responsible for providing the peer-to-peer network through which the service descriptions are reachable.
- The *Query Service*, which is responsible for hosting OWL-S service descriptions and responding to service queries.

Whereas the *Query Service* is only used by the hosting peers, meaning the peers that provide an OWL-S service description, the *Query Distribution Service* is required by both hosting and querying peers. Querying peers use the query distribution service to send queries, while hosting peers receive queries and return results using the service (see Figure 26).

![Figure 26: Interaction between querying peer and hosting peer.](image)

Typically, both the *Distribution Service* and the *Query Service* should be provided by a local application server. In this way the application server is constantly connected to the peer-to-peer network, and thus startup-related costs are avoided. Whenever a user needs to find a service it has to obtain a reference to the *Distribution Service* from the application server, and post, for example, an OWL-QL query (see Figure 27):

![Figure 27: Sequence Diagram of Distribution Service usage.](image)

In the diagram above a client application contacts the local application server’s *Naming Service* and looks up the distribution service using its well-known naming service name. If a reference to the distribution service is returned, the client invokes the *getServiceURI* method supplying a query string as its only argument. The distribution service creates a *ResponseHandler* thread that is responsible for...
A Peer-to-Peer Distributed Semantic Service Registry

distributing the query on the peer-to-peer network, and to wait (asynchronously) for a response within a given time frame.

Whenever a query request results in a positive response, this response is a list of URI references to the OWL-S service description of the found service. Using this reference the user can invoke methods, described as processes in the OWL-S service description, or generate a specialized proxy that can be used for service invocation. The invocation of service primitives based on an OWL-S service reference is handled in a broker-architecture fashion in [15], whereas in [48] the ability to search and invoke a service is integrated into a Virtual Machine. This thesis only provides the search-functionality, and leaves the invocation aspects up to other applications.

6.2.1 Garbage Collection

Ignoring the semantic description capabilities of the proposed registry, the registries contents can be viewed as a table, where each column represents a fixed meta-data field. Whereas in a UDDI registry a single server keeps the whole table, in the proposed architecture every hosting peer contributes to the rows (see Figure 28).

Figure 28: Difference between a Peer-to-Peer distributed registry and UDDI registries.

Whenever a hosting peer is no longer available in the peer-to-peer distributed registry, the table rows it contributed to are removed. This garbage-collection is a nice side-effect of the proposed architecture and solves one of the major problems with UDDI-like registries, namely dead links.

In UDDI Version 3 [35] the support for distributed registries does allow distribution of rows between affiliate registries that are coordinated using a root registry. This forms a source of centralization, something that does not fulfill the requirements of this report.

6.3 The Query Service

The Query Service is responsible for hosting OWL-S service descriptions in an associated Knowledge base, and exposing methods for querying these descriptions. The query service is designed to handle queries received and passed on by the Query Distribution Service, and is not designed to be invoked directly by the user (see Figure 29).
Figure 29: Sequence Diagram of interaction between Distribution Service and Query Service.

Figure 29 describes the situation at a hosting peer. The distribution service and the query service are both running as services within an application server context. The distribution is connected to the peer-to-peer network, and upon reception of a service query the query is passed on to the query service. After that the query service connects to the local Knowledge base implementation and queries the knowledge base for matching query bindings. If the query yields results, these results are processed by the query service, which returns a list of service URIs.

In [28] an OWL-S service description is queried by comparing the hosted service description with a query template. This, however, forces queries to stay within the scope of the fixed properties of the OWL-S Profile ontology. In the proposed architecture an ontology query language, rather than profile templates, is used to query the hosted service descriptions. This allows a query to search for required properties in the profile, process model and grounding of the OWL-S service description. The only information that is not searchable is the concrete process grounding, for example WSDL, which is associated with the abstract OWL-S grounding.

One problem with querying OWL-based service descriptions is that it normally requires a lot of computer resources. In [12] a UDDI registry is modified to host OWL-S service descriptions. Since UDDI registries are designed to hold vast amount of service descriptions, often in heavy-load centralised servers, the amount of computer resources required to run such a system is extensive. By proposing a peer-to-peer based architecture, peers only host as many service descriptions as they host services, the load is distributed amongst all hosting peers on the network.

6.3.1 Administration

As a service running inside an application server, the query service should not be invoked directly by user applications, but rather via the distribution service. For the host, however, there is the need to administrate the knowledge base whenever services are added, removed or modified. Depending on the type of application server used, this administration can be handled by Simple Network Management Protocol (SNMP) agents or via Java Management Extension clients (see Figure 30).
Figure 30: Administration of QueryService using SNMP agent.

For performance reasons it is quite likely that the knowledge base will keep its OWL-S service description in a database rather than loaded into memory. This means that if an OWL-S service description changes, the database needs to be reloaded with the updated service description. For such reasons having an administration interface to the query service is a required feature.

This does, however, present a problem in many cases. Consider, for example, an OWL-S service description that contains dynamic contextual information in the profile ontology. This is information that needs to be updated frequently, something that requires software automation. This, however, is outside the scope of this work.

6.4 The Query Distribution Service

The Query Distribution Service is responsible for providing a peer-to-peer network and the service primitives for distributing service queries and receiving query results. The distribution service, unlike the query service, is required for both hosting peers and querying peers, and has to be available in order to make use of the proposed architecture. The distribution service serves as the public interface to the proposed semantic distributed service registry.

As mentioned before, the rationale behind the distribution service’s use of a peer-to-peer network is to avoid the centralization that typical static registries, like UDDI, expose (see Figure 31).

Figure 31: Typical static UDDI Registry configuration.

Centralized registries form obvious points-of-attack and need to be avoided. In a centralized system a client opens a connection to the registry in order to find a service reference. In the proposed architecture the client’s application server is
constantly connected to a peer-to-peer network. While a UDDI registry keeps its information in a table with a column for each meta-data field, a peer-to-peer distributed version of such a registry is a networked table where each hosting peer provides one or more rows.

**Figure 32: The proposed Peer-2-Peer distributed registry.**

In Figure 32 the concept of a single registry is absent. The registry function is now an integral part of the peer-to-peer network. Since the network exists as long as there are peers available, all peers have to be rendered useless in order to remove the registry.

There may be situations where a centralized peer is introduced into the registry. Consider, for example, a group of service providers that cannot join the peer-to-peer network. In order for their service descriptions to be found using the proposed architecture a *delegate* peer can be chosen to host the OWL-S service description on their behalf. A proprietary protocol needs to be shared between the hosting peer and the services in question (see Figure 33).

**Figure 33: Hosting OWL-S descriptions on behalf of other hosts.**

With the cost of some limited centralization introduced by the *delegate* peer, the descriptions of the services in question can be found. If, however, the amount of delegate peers grows, a different architecture may be more suitable, such as, a Jini-based solution.

### 6.5 Usage Scenarios

The proposed architecture provides a decentralized registry for OWL-based service descriptions. Queries are not limited to template-matching but allow *description logic* reasoning, and the registry has no centralization. What are the possible usage scenarios that require such an architecture? This report argues that the functionality
provided by the proposed architecture should be utilized by a client using a pre-compiled proxy object (see Figure 34).

![Figure 34: A proxy-base usage of the proposed registry.](image)

In a proxy-based system a proxy object encapsulates the query of the registry for a service, as well as the invocation of the service. To the user the proxy provides a simple interface that hides the complexity all together. A proxy, in this context, can be thought of as a broker that mediates between the service user, service description and the service itself. Such a broker-based system is described in [15]. There are, however, several strategies available when designing proxies.

### 6.5.1 Runtime Dynamic Proxy

In a Runtime Dynamic Proxy system the client needs to pass the actual service query as an argument to the proxy when invoking a service. Using the supplied query the proxy queries the Distribution Service, and invokes a service if the query yields a response (see Figure 35).

![Figure 35: Runtime Dynamic Proxy.](image)

The use of runtime dynamic proxies requires enough knowledge about the service ontologies to provide a query argument that is capable of finding the required service. In addition the found service’s process model can refer to a composite process with a Choice control structure. This requires the user, or the proxy, to specify what process to invoke, most likely through the use of a call-back feature.

Although the runtime dynamic proxies work, they require too much knowledge from the user. This is why such a proxy-system should only be used in exceptional situations where the ability to tweak the system is required to produce the needed result.

### 6.5.2 Service Type specific Proxy

For each OWL-S Profile sub-class representing a service type, it is likely that a software object class is created for the application context. Consider, for example, the Sensor OWL-S profile sub-class:

```xml
<owl:Class rdf:about="%Sensor">
  <rdfs:subClassOf>
    <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="%detectsDomain" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```
The Sensor class defines three compulsory properties, which can be thought of as possible methods on a service instance, or simple value fields. Thus, class-specific information can be extracted from its OWL ontology and used to create an object-oriented class-structure, for example a service-type specific proxy.

Consider the following instance of the sensor profile:

```xml
<Sensor rdf:ID="Sensor_Alpha">
  <detectsDomain rdf:resource="#Ground"/>
  <hasOutput rdf:resource="#Track"/>
  <hasClassificationCapability rdf:resource="#Unique"/>
</Sensor>
```

A proxy for a service with hasOutput value of Track would have another interface than a proxy for the value Plot. This way the profile sub-class and eventually the found instance itself determine the interface-methods available to the client (see Figure 36).

Figure 36: Sequence Diagram describing service type proxy.

A client uses a ProxyFactory to find and create a ServiceProxy representing a Sensor OWL-S described service. The Sensor class defines three compulsory properties, where hasOutput is one. The getOutputHandler method on the sensor proxy returns a proper specialized output handler for the instance value of hasOutput. In the example above this value is Track, thus a TrackHandler is returned. If the ontology is standardized to some extent, it could be feasible to assume that a service that has an output value of Track provides the service primitive getTrack.

This just serves as an example of how specialized proxies can be made to match well-known service types. This makes the system less dynamic, but it is not likely that ontology modifications will render the previous versions useless.
6.6 Summary

The driving force behind the design of the semantic distributed service registry is availability. By avoiding centralization and distributing the registry across a peer-to-peer network, the registry is available as long as there are services available. By distributing the registry content to the actual hosting peers, the registry is always clean, meaning that garbage collection happens as a side-effect of the peer-to-peer distribution.

Describing services using a semantically rich language, paired with an expressive query language, a service user is able to find suitable services by specifying queries that search for service capabilities rather than service types. Having found a service, it is up to other applications to handle actual invocation on it.

The proposed architecture defines a semantic query service, which is responsible for hosting OWL-S service descriptions and processing queries, much in the same way UDDI registries hosts WSDL documents. The proposed architecture, however, avoids centralization by placing a query service at each hosting peer. Clients queries the registry using the proposed query distribution service, which is responsible for distributing a query to each hosting peer using a Peer-to-Peer network. This design removes centralization from a typically centralized registry architecture, and makes it possible to find services using semantic properties without considering any centralized points of intelligence.

This ends the overview of the proposed conceptual architecture of a semantic, distributed service registry. In the following chapters, the Query- and Distribution Service is described in more detail. Finally a demonstrator application is presented that provides a reference implementation of the proposed registry.
7 The Conceptual Query Service

7.1 Introduction

When looking at a registry as a black-box, the protocol is quite simple. A query is given as input and a result is received as output (see Figure 37). For the proposed query architecture the response of a supplied query is either nothing, if the query is not matched, or a collection of URI references to matched OWL-S service descriptions.

Upon reception of a query, the proposed Query Service tries to match it against a collection of OWL-S service descriptions contained within an associated Knowledge base (KB). Depending on the query language supported by the query service, the query is analyzed and if needed modified, and then passed on to the KB.

The query service should provide functionality for administrating the KB, as well as processing, for example, OWL-QL queries. Since it is likely that the proposed registry will be contained within an application server, the administration of the KB can be done via a Simple Network Management Protocol (SNMP) agent or a Java Management Extension (JMX) client. Querying the KB is strictly handled by the Distribution Service (see Figure 38).

The Query Service provides two sets of interfaces: to the Distribution Service the Query Service provides methods for querying the hosted OWL-S service descriptions; to be able to manage the hosted service descriptions, the Query Service provides an administration interface to, for example, SNMP agents.

The Administration interface provides methods for adding, updating and removing OWL-S service descriptions to/from the KB database. In addition, methods for diagnostics and reporting could be provided.

The interface provided to the Distribution Service is quite simple. In a query system, it normally is possible to define the maximum number of hits a query should return. The proposed query service is integrated into a peer-to-peer network, thus limiting
the number of results one peer returns gives no guarantee to the overall size of the result-set. This is why the query service’s protocol, regardless of its importance to the proposed architecture, is very simple. A query string is submitted, and the results are returned.

7.2 OWL-S Service Description Administration

In order for the Query Service to handle queries it first needs to possess a repository of OWL-S service descriptions. This means that the Knowledge base (KB) needs to be configured to load those service descriptions. In addition, services may change from time to time, thus a mechanism for updating service descriptions needs to be provided in the architecture. There also needs to be support for removing a service description, either by requesting this directly or providing a validity timer. All these administration methods should be provided by the query service’s administration interface, which is, as mentioned, manageable via an SNMP agent or JMX client.

The configuration of the KB is handled via the import mechanism that OWL provides:

```xml
...<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="&radar-78;" />
  <owl:imports rdf:resource="&isar-13;" />
</owl:Ontology>
...
```

By specifying a configuration OWL ontology that imports the OWL-S service descriptions that the query service needs to host, one consistent ontology model is created. Thus instead of creating a custom solution for hosting multiple ontologies like in [49], existing RDF data model solutions can be utilized.

How the initial configuration, update and removal of service descriptions are handled depends on the implementation details of the KB. For example, if a memory stored solution is chosen, updating is as trivial as restarting the KB using an updated configuration file. If the KB is using a database backend to avoid network roundtrip delays, this process involves clearing the database and restarting the KB.

When starting the query service, a referenced configuration ontology is loaded into the KB. If during the course of the service’s uptime a reconfiguration is needed, the administration interface provides the following two methods:

- `reconfigure()`
- `reconfigure(String uri)`

The first method restarts the KB using the initial referenced configuration ontology, which may or may not have been modified. The second method reconfigures the KB using the specified URI reference argument to identify a configuration ontology.

Before loading a configuration file into the KB, this file should be verified to ensure that the proposed registry stays consistent. This is of great importance since queries can be expressive, and ontologies that do not validate against the OWL rules can produce unwanted results.

Adding and removing OWL-S service descriptions can be done by modifying the configuration file, clearing the database and reconfiguring the KB. It is also possible for the query service to provide more advanced means of administration.
The OWL-S Verifier

The OWL-S Verifier carries a lot of responsibility when it comes to keeping the KB consistent. This means that all service descriptions contained within the KB need to be well-formed and valid OWL ontologies. In addition the verifier needs to make sure that the individual service descriptions have not expired.

Validating the OWL-S service descriptions would normally involve checking the ontology for OWL DL conformance. With the current version of OWL-S (version 1.0), however, the ontology itself is only valid as an OWL Full ontology. This means that since all service descriptions are based on OWL-S, they only conform to OWL Full and cannot be validated directly. OWL-S version 1.1 [50] is planned to be the first OWL DL conformant OWL-S ontology, thus the verifier can be designed by introducing a custom OWL-S verifier, or the verification effort should be delayed until the release of OWL-S version 1.1.

To illustrate the importance of validating the OWL-S service descriptions, consider the following ontology fragment:

```xml
<Radar rdf:ID="Radar-78">
</Radar>
```

The Radar instance does not include the compulsory service:presentedBy, profile:serviceName or profile:textDescription properties. Since the ontology states, by using cardinality restrictions, that these properties are required, the applications working with these ontologies may take their availability for granted. If, for example, an application is given a reference to the OWL-S profile ontology instance of a service description, and this application needs to navigate to the process model (via the service ontology), this is not possible because the service:presentedBy property is absent. Since OWL-S is currently OWL Full conformant, the validator can assume that the Radar instance is extended elsewhere on the Internet, thus is the instance not invalid. This is because of the open world context that applies to OWL Full.

OWL-S Document Version Control

Whenever an OWL-S document is submitted to the query service, and passes the validity check, its timestamp is checked against the database. The timestamp is added to an OWL-S document by augmenting the owl:Ontology element with an owl:versionInfo element:

```xml
<owl:Ontology rdf:about="">
  ...
  <owl:versionInfo>1066643833110</owl:versionInfo>
  ...
</owl:Ontology>
```

The content of the owl:versionInfo has no range restrictions, and can contain both datatypes and objects, such as, instances from the OWL-S sponsored Time ontology [51] or a millisecond value.

In order for the KB to determine if an ontology is actually expired the expires property is introduced. analogue with the versionInfo property the expires property is added to the owl:Ontology header element:

```xml
<owl:Ontology rdf:about="">
  ...
  <uls:expires>1066643833110</uls:expires>
  ...
</owl:Ontology>
```
The proposed architecture defines the *uls* namespace. This namespace contains the `owl:AnnotationProperty uls:expires`. The content of the `uls:expires` element is the same as the `owl:versionInfo` element. There is one restriction, namely that the value of the `uls:expires` element must represent a time later than the value of the `owl:versionInfo`.

7.3 The Query Processor

The *Query Processor* module is responsible for processing a received query, and returning the result to the user. The query processor accepts queries that follow some basic requirements, and returns a collection of valid OWL-S Service instance URI references. This is done by passing the received query on to the *Knowledge base* (KB) for processing, or retrieving the results directly from the *cache*, if available (see Figure 39).

![Figure 39: Query Processor.](image)

In the proposed architecture the query service is only invoked by the administration interface or the *Distribution Service*. Depending on the query language used in the supplied query, the query service can augment a query with some default *triple* filters. Both RDQL [29] and OWL-QL [30] use query chains to filter out matches within the underlying data model. This makes it possible to assume that certain variables and triples are included in the queries by default, and augment the incoming queries with these default facts (see Figure 40).

![Figure 40: Augmentation of RDQL query by the Query Modifier.](image)

The *Query Modifier* is responsible for analyzing incoming queries and changing them if needed. By accepting sub-queries, which are queries that are yet to be completed, the users of the proposed registry can create much simpler queries.
The Knowledge base

The Knowledge base (KB) is responsible for hosting OWL-S service descriptions, and accepting queries to process on them. It has been considered a separate module because both administration agents and the distribution service use it.

Most projects under the Semantic Web umbrella build their data model on the Jena framework [52]. The Jena framework provides the basic building blocks for representing pure RDF-based ontologies. One of several advantages of the Jena data model is its support for both in-memory representation and persistent database storage of models. It does have a reasoning architecture that makes it possible for applications to reason about RDF/S and OWL ontologies represented within the framework, this support is, however, very experimental (see Figure 41).

Figure 41: Jena Reasoning architecture.

Since the Jena reasoning architecture implementation suffers from very poor performance, several other optional reasoning systems have been suggested, for example, Java Theorem Prover (JTP) [53] or Pellet [54]. Where the Jena reasoning system inserts inferenced facts into the data model when the ontology is loaded, JTP and Pellet are able to do this on-demand, something that greatly improves performance. However, reasoning support is evolving rapidly in the Jena framework, such that basic support for on-demand inference is already present.

The KB is designed to work with different implementations of the data model and reasoning architecture, but with a consistent API to the application user. There are two compulsory methods provided by a KB, namely:

- **Query**: This takes a supported query, for example in RDQL, as a single argument. The KB processes this query on the hosted OWL-S service descriptions, and returns a collection of service instance URI references.

- **Configure**: Takes a valid URI of an OWL configuration ontology as a single argument, and loads it. Depending on the reasoning architecture used, the data model undergoes inferencing.

In addition to these simple primitives, specific KB implementations may provide more specialized ones. Consider, for example, a KB utilizing a database backend. This requires some administration methods for analyzing the database connection amongst others.

7.4 Summary

The Query Service would normally provide the core functionality of a registry, namely hosting service descriptions and accepting and processing queries. In the proposed architecture each query service is designed to be a small part of a larger architecture. This is why the query service has a rather simple design. It provides
two interfaces; namely one to the *Distribution Service*, which simply involves querying the knowledge base, and one to administration interfaces, accessible via SNMP or JMX.

The query service is designed to be open, meaning that support for several query languages can be provided, as well as support for different kinds of knowledge base implementations.

In this chapter the Query Service was described on a conceptual level. In the following chapter, the last architectural module, namely the Distribution Service is described.
8 The Conceptual Distribution Service

8.1 Introduction

In the proposed registry architecture the *distribution service* is responsible for providing a peer-to-peer network to both *hosting* peers and *client* peers. Using the distribution service a client is able to distribute service queries and receive the results. Since services may be based on various software, hardware and network solutions, it is a requirement that the peer-to-peer technology is available for the platform used by the hosting peers (see Figure 42).

![Figure 42: A possible environment that the peer-to-peer network operates in.](image)

With hosts potentially ranging from mainframe computers to HF radio terminals, and with several network and transport technologies, the peer-to-peer solution needs to be open. Unlike *Gnutella* [17], which requires HTTP for transportation, *Jxta* [18] provides a set of XML-based peer-to-peer protocols for transport independent peer-to-peer networking. This report argues that given the requirements for the proposed registry, the Gnutella-based approach taken by [16] is not open enough. Therefore the proposed distribution service utilizes *Jxta* as the peer-to-peer technology of choice.

In situations where there are large amounts of participating peers on the network, distributing queries to every peer may introduce serious congestion into the network, something that can cripple the whole system. In peer-to-peer environments this is often remedied by introducing *super-peers* that keep routing tables [19, 20]. By distributing queries via a super-peer, queries are only propagated to peers on the network that are relevant. These super-peers, however, are also *points-of-attacks*. The proposed distribution service introduces a *distributed query router*, which provides routing functionality at each peer.

8.2 The Distribution Service Architecture

The distribution service is responsible for providing a peer-to-peer network on which peers can distribute queries, and receive query results. When being used by a querying agent, for example a dynamic stub or a query application, the distribution service is responsible for analyzing the submitted query, and deciding how to route the query to suitable service providers. When being used by a service provider, the
distribution service is responsible for listening for incoming queries, forwarding these to the *Query Service* and finally returning the query result back to the originator. Thus the distribution service is designed to service both service providers and querying agents, or both at the same time. The distribution service can be considered the network layer through which a client application and the provider server communicate (see Figure 43).

![Figure 43: Distribution Service Layering.](image)

The Distribution Service is responsible for connecting the client peer with peers that provide a Query Service. This means that when the registry is queried, service providers may receive the submitted query and choose to respond.

The actual distribution of queries is handled by the *Query Router*, which is an integral part of the distribution service. Both the query service (when present) and the query router need to access ontologies via their respective URIs. If the ontologies are not available at every host, the ontologies are downloaded when processed. This introduces *points-of-attacks* into the system, namely the hosts where the ontologies are provided. The proposed distribution service provides a *Namespace Resolver*, which can both resolve Jxta-based URI references, as well as resolve normal *HTTP* references on the peer-to-peer network (see Figure 44).

![Figure 44: Distribution Service.](image)

The distribution service can be accessed through three distinct interfaces:

- Registry interface;
- QueryService interface;
- Administration interface.

The *Registry* interface is used by a client to search the registry. The *QueryService* interface is used by a service provider to associate the distribution service with a query service. The *Administration* interface is used by a client to monitor and manage the distribution service (see Figure 45).
A client can access the proposed service registry through the Registry interface provided by the distribution service. The distribution service itself is contained within an application server, which can be contacted by the client in order to retrieve a remote reference to the service, for Remote Method Invocation (RMI).

### 8.2.1 Registry Interface

The Registry interface provides the registry functionality to the client. Unlike registry technologies like Java Naming and Directory Service (JNDI) the registry interface only provides methods for querying and not for binding. Whereas the query service’s interface provides functionality local to the service, the registry interface provides access to the whole distributed registry. The interface provides the following basic service primitives:

- `getService(Query query)`
- `getService(Query query, long timeframe)`
- `getAllServices(Query query)`
- `getAllServices(Query query, long timeframe)`

The service primitive, `getService`, distributes the supplied query argument on the network, and waits for the first response to arrive. Since the registry is distributed it can take a while before a query result arrives. The `timeframe` argument specifies how long the method waits for a response before generating an exception. If the time frame argument is not specified, a default value is used as a time frame. This default value is entirely up to the implementation of the distribution service.

The `getAllServices` service primitive returns all query results that arrive within the specified or default time frame.

Consider the following sequence diagram that describes a typical client-registry interaction:
A Peer-to-Peer Distributed Semantic Service Registry

Figure 46: Client-Registry interaction.

The client looks up the distribution service’s Registry interface using the local application server’s naming service. Then the client tries to search for a service using the getService service primitive.

8.2.2 QueryService Interface

When the distribution service is being used by a service provider, it needs to be associated with a Query Service to which the distribution service will forward incoming queries. The Query Service interface provides a single method, namely

- handleGroup(String groupId)

which takes a peer-to-peer group ID as a single argument. This method tells the distribution service that if it receives a query via peer-to-peer group groupId, the application server’s Query Service is ready to process it.

It can be argued that the QueryService interface should be left open, and the method definition up to the individual implementations. Not every peer-to-peer technology provides grouping functionality. In the proposed architecture Jxta peer-to-peer [18] technology is the technology of choice, but is not an absolute requirement. In [16] Gnutella [17] peer-to-peer technology was used to provide a distributed registry. Gnutella, however, does not provide grouping and only provides HTTP [59] networking.

8.2.3 Administration Interface

In addition to the Registry and QueryService interfaces, the distribution service provides an administration interface as well. This interface can be accessed by Simple Network Management Protocol (SNMP) agents or Java Management Extension (JMX) clients. The methods provided by the administration interface are not defined, but can be left up to the implementation details of the distribution service. For example, it can provide methods that return statistical information concerning the use of the service.

8.3 The Query Router

When there are several peers located on the network, propagating each query to all peers can become a performance. If, for example, a peer only hosts weather-related services, it will never be able to service a query for a book buying agent, and will be better off not ever receiving such queries at all. Edutella [19] and JxtaSearch [20], introduces super peers or hubs on the network, which are responsible for relaying queries to peers that may actually be able to service the request. By analyzing
incoming queries, the super peer is able to determine, based on a dynamic routing table, which peers are able to handle the queries (see Figure 47).

**Figure 47: Super Peer-based routing.**

The distribution service of a peer will always distribute a query via its nearest *super peer*, which is dynamically elected, and who builds its routing table dynamically. The super peer analyzes the received query, and determines which peers on the network are able to provide a response to the query, and forwards the query to those peers.

Even though the super peers are dynamically created, they do form a centralized point of intelligence, and thus impose vulnerability to the system, even though the system is able to recover if a super peer is removed. In the proposed architecture the use of *peer group*-routing is utilized instead.

### 8.3.1 Peer Group Routing

The concept of *peer groups*, like in project *Jxta*, was introduced to limit peer-to-peer broadcasts to only propagate within a given peer group. A Peer is able to join several peer groups at the same time. This has resulted in that most applications provide their own peer groups, and all participating peers are required to join that group (see Figure 48).

**Figure 48: Jxta Application-based Peer Groups.**

Figure 48 shows that, a Jxta Peer is able to join several groups at the same time, and all peers are members of the *top-group NetPeerGroup*.

The proposed *Query Resolver* utilizes peer groups to facilitate distributed query routing. Since all service providers need to share ontologies of OWL-S service profiles, these ontologies serve as a suitable place for sharing routing information.
The idea is to relate an OWL-S Profile sub-class with a unique peer group identifier, through the use of the introduced OWL property peerGroup. Consider, for example, the following ontology fragment, which describes a Jxta PeerGroupAdvertisement using OWL-constructs:

```xml
<owl:Class rdf:ID="#PeerGroupAdvertisement"/>
<owl:DatatypeProperty rdf:ID="GID">
   <rdfs:domain rdf:resource="#PeerGroupAdvertisement"/>
   <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="MSID">
   <rdfs:domain rdf:resource="#PeerGroupAdvertisement"/>
   <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>
<owl:Class rdf:about="#PeerGroupAdvertisement"/>
   <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1">
         <owl:onProperty rdf:resource="#GID"/>
      </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1">
         <owl:onProperty rdf:resource="#MSID"/>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

Sticking with the minimal, only the required PeerGroup information is defined, namely Group ID (GID) and Module Service ID (MSID). An instance of the PeerGroupAdvertisement class looks like the following fragment:

```xml
<PeerGroupAdvertisement rdf:ID="RadarGroup">
   <GID>urn:jxta:uuid-69D604A7E39D43ABB3628A25BF58B3A02</GID>
   <MSID>urn:jxta:uuid-DEADBEEFDEAFBABEFEEDBABA000000010306</MSID>
</PeerGroupAdvertisement>
```

By introducing a property jxtaGroup that relates instances of OWL-S Profiles with instances of PeerGroupAdvertisement, it is possible to enforce Jxta peer group participation based on the OWL-S Profile class of a service description. The query router assumes that the jxtaGroup property is required for all OWL-S Profile subclasses. Consider the following ontology fragment:

```xml
<owl:ObjectProperty rdf:ID="jxtaGroup"/>
   <rdfs:subPropertyOf rdf:resource="#peerGroup"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#jxtaGroup">
   <rdfs:domain rdf:resource="&profile;#Profile"/>
   <rdfs:range rdf:resource="#PeerGroupAdvertisement"/>
</owl:ObjectProperty>
<owl:Class rdf:about="#Radar">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#jxtaGroup"/>
         <owl:hasValue rdf:resource="#RadarGroup"/>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```
By using the OWL Restriction class to restrict class membership, every instance of #Radar, from the above example, automatically contains a property jxtaGroup with an object value of #RadarGroup (see Figure 49). The jxtaGroup is a sub-class of the more general property peerGroup. This allows other peer-to-peer technologies than Jxta to be used instead.

Figure 49: Relation between Profile-class and Jxta-Group.

Whenever a query is to be distributed, the query router analyzes the query for the presence of a certain filter-chain, which can be a RDQL where-clause or an OWL-QL query pattern. If this filter-chain includes a triple with the subject ?profile and the property rdf:type or rdfs:subClassOf, the object of the triple should provide the concrete OWL-S Profile sub-class reference. If such a triple is not present, this means that the query does not consider the OWL-S Profile hierarchy, and the query router is not able to do anything else than distributing the query to all peers.

Consider, for example the following triple:

( ?profile, rdf:type, sensors:Radar )

The object part of the clause, namely sensors:Radar, is the OWL-S Profile sub-class of the service description. Now it is possible to follow the jxtaGroup property and extract the GID and MSID values of the related PeerGroupAdvertisement (see Figure 49).

The Query Resolver uses the GID value to instantiate and join a Jxta peer group, with GID as its unique identifier. After joining the peer group it distributes the query in the group using its Resolver Service. This means that every Peer has its own ontology-based router. The routing-table is fully integrated into the ontologies used, and all centralization is avoided, as long as the ontologies themselves are resolvable via the Peer-to-Peer network.

A service provider’s Distribution Service needs to forward queries to the query service for all peer groups it supports. This is handled by PeerGroup Handlers, which are instantiated for each distinct peer group.

8.3.2 Provider PeerGroup Handler

Since queries are only routed via specific peer groups, the provider needs to handle incoming queries for the peer groups of all OWL-S Profile sub-classes contained within its Query Service knowledge base. In Jxta this is done by joining each associated peer group.

This process involves analyzing all OWL-S service descriptions contained within the knowledge base, and extracting the relevant group information. For a Jxta-based solution, this means extracting the GID value of all related PeerGroupAdvertisement instances. For each distinct GID the distribution service joins the PeerGroup, and registers a Peer Group Handler, which is responsible for forwarding incoming queries to the Query Service, and returning the result back to the querying agent (see Figure 50).
Figure 50: PeerGroup Handlers.

With one PeerGroupHandler for each peer group, each query distributed by a client is resolved by one PeerGroupHandler for each suitable peer. The PeerGroupHandler forwards the query to the Query Service, which executes it and returns the result. Since every query has a source peer, the query result is sent directly to that peer.

8.4 The Ontology Resolver

The OWL-S ontologies stored in the Query Service, and the ontology references used in queries, all rely on external ontologies that need to be resolved. This means that the ontologies are actually downloaded from its concrete URI location, which in many cases may be a centralized host. Consider, for example, the following ontology fragment:

```html
<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.w3.org/2002/07/owl" />
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.0/Service.owl" />
    ...
</owl:Ontology>
```

When loading this ontology the imported ontologies, specified using the `owl:import` properties, are downloaded and integrated into the RDF model. Both properties reference static hosts, which do provide single points-of-attack. Ontology references used in the proposed distributed service registry should either contain a peer-to-peer-based namespace, or be resolvable using the proposed Ontology Resolver. Consider, for example, the following ontology fragment:

```html
<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.w3.org/2002/07/owl" />
    <owl:imports rdf:resource="urn:jxta:uuid-69D604A7E39D43ABB3628A25BF58B3A002/owl-s/1.0/Service.owl" />
    ...
</owl:Ontology>
```

The first import references the standard OWL namespace, and does not need to be downloaded, as there is normally built-in support for the ontology. The second ontology is resolvable via a Jxta-based resource identifier. This means that all peers, within a Jxta PeerGroup, are asked to deliver the ontology, and again, centralization is avoided.

Using Jxta-based resource identifiers provides an elegant way of avoiding centralization, but it does introduce some problems. For example, the inclusion of Jxta-based ontology references results in cluttered ontologies. The chance of introducing errors into an ontology is also much more plausible. The main problem, however, is that the whole Semantic Web [05] initiative is based on ontology sharing. Having Jxta-based ontology references means that the ontologies can only
A Peer-to-Peer Distributed Semantic Service Registry

be used in applications that have access to the Jxta-network technology. If that is not considered a drawback for a given application domain, using Jxta-based ontology references is still an option. The same problem also applies for peer-to-peer technologies other than Jxta.

In the proposed distribution service the Ontology Resolver is introduced. It is responsible for resolving ontology references whether they are located on the peer-to-peer network, or at some external centralized host. Whenever an ontology is referenced, the Ontology Resolver is asked to deliver it. This can be done by retrieving the ontology from the cache, downloading the ontology from its URI location or resolving the ontology on the peer-to-peer network. Resolving the ontology in the peer-to-peer network is done by distributing an ontology query. Consider, for example, the following query:

<?xml version="1.0" encoding="UTF-8"?>
<ontology>
  http://home.student.utwente.nl/r.p.bjornstad/owl/ontology.xml
</ontology>

If there are peers on the network that have the specified ontology in its cache, a response message may be returned to the queries originating peer. Consider, for example, the following response:

<?xml version="1.0" encoding="UTF-8"?>
<ontology
  uri="http://home.student.utwente.nl/r.p.bjornstad/owl/ontology.xml">
  <![CDATA[
    <?xml version="1.0" encoding="UTF-8"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns">
      <owl:Ontology rdf:about="">
        ...
      </owl:Ontology>
    </rdf:RDF>
  ]]>}
</ontology>

The ontology that belongs to the URI specified in the originating query is the querying peer stores this ontology in its cache, such that future references to the same ontology can be resolved locally.

Given the possible size of an ontology, considerable care has to be put into how the query resolver is implemented. Consider, for example, that a peer tries to resolve a large ontology within a PeerGroup. If a large amount of peers within that PeerGroup is able to return that ontology, the network traffic can be intensive. An implementation of the ontology resolver needs to limit the total amount responses to an ontology resolver query.

8.5 Summary

The proposed Distribution Service was designed to remove any form of centralization from the distributed service registry. By distributing the registry content on a peer-to-peer network, dead-link problems, which UDDI suffers from, is avoided. This is a huge advantage when it comes to keeping the registry consistent. However, service providers need to make sure that the service they represent using OWL-S is actually present. This report argues that by hosting a service description at the service host itself, enforcing this integrity constraint is far more feasible.
Even though peer-to-peer networks bring some interesting advantages, there are several problems as well. Without proper routing mechanisms the network could easily be flooded. Routing tables, dynamic or static, do form centralized points of intelligence, something that should be avoided in the proposed architecture. Instead of using routing tables, the proposed distribution service uses peer groups to limit propagation of query messages. By embedding routing information into the service ontologies, it is possible to create a peer-to-peer distributed router, where each peer is responsible for selecting the peer groups to which a query message should be propagated. This does, however, limit the choice of possible peer-to-peer technologies to those providing grouping functionality.

Another problem follows from the nature of ontologies. They are supposed to be shared, and thus should be downloadable via its resource identifier. This means that the host, or hosts, where the ontology is located forms a centralized threat, something that is unacceptable in the proposed architecture. The solution is to either use peer-to-peer based resource identifiers to resolve ontologies, or use the proposed Ontology Resolver, which is a part of the distribution service. By making it possible to resolve an ontology via the peer-to-peer network, centralized dependencies are removed and ontologies can still be used outside a peer-to-peer network.

In this chapter the Distribution Service was described. This concludes the conceptual design of a proposed semantic, distributed service registry. In the next chapter, a demonstrator application, which contains a prototype implementation of the proposed registry architecture is described.
9 The Demonstrator

9.1 Introduction

In the previous chapters a semantic distributed service registry has been proposed. The architecture suggests that given the pre-existing requirements, a peer-to-peer distributed OWL-S-based [06] solution should produce a centralization-free, highly semantic service registry. This chapter describes the demonstrator that was developed to demonstrate the core functionality of the proposed architecture.

The proposed Distribution Service and Query Service were implemented using Java [26] and designed to be contained inside the JBoss [55] application server. This approach has two main advantages:

- A client interacts with the registry in a manner similar to current registry technologies, for example, UDDI [02] registries. In order to use the registry the client first needs to obtain a reference to it from the local application server.
- A service provider can host the provided services inside the same application server as the query service and the distribution service.

Both clients and service providers requires a running JBoss application server in order to make use of the registry. For a client host that does not provide any services, this requires JBoss configured with only the Distribution Service (see Figure 51).

Both clients and service providers requires a running JBoss application server in order to make use of the registry. For a client host that does not provide any services, this requires JBoss configured with only the Distribution Service (see Figure 51).

![Figure 51: Client host JBoss configuration.](image1)

For a service provider the application server is configured with more responsibility. In addition to the Distribution Service, which is needed by both querying clients and service providers, the Query Service is configured for a service provider (see Figure 52).

![Figure 52: Service Provider JBoss configuration.](image2)

In addition to the proposed registry components, the application server contains the services that are provided by the host. In order to demonstrate the functionality of the registry prototype, several dummy services were created using different implementation solutions.

The demonstrator was created to show that a client can use the proposed registry to locate a semantically described service, using Jxta [18] peer-to-peer networking, and invoke that service (see Figure 53).

![Figure 53: Demonstrator setup.](image3)
A Peer-to-Peer Distributed Semantic Service Registry

Figure 53: The basic demonstrator functionality.

A client application queries the registry for a SAR Sensor service. This is done by invoking the getService method on the Distribution Service, after first obtaining a reference to it from the local application server. The distribution service propagates the query argument on the Jxta peer-to-peer network, and any listening provider passes it on to its Query Service to check for a match. If a match is found, the service URI is returned to the client, who then can choose to invoke the service.

In order for this scenario to be realizable, several features were added to the demonstrator, in addition to the Distribution- and Query Service:

- A military sensor OWL ontology of OWL-S Profile sub-classes was created. Using this ontology it is possible to place a service within a sensor hierarchy. This follows the guidelines suggested in chapter 4.

- A Java class-library was created with Java representations of the sensor ontology classes. In addition, several proxies were created for easy interaction with the proposed registry.

- Actual services were created, using different implementation techniques. This was necessary in order to demonstrate that a client can find a suitable service, regardless of its implementation details.

- A distributed environment of hosts was created to test the actual usability of the proposed registry.

- An OWL-S Service Invoker was created. This invoker is able to invoke a service based on the information contained in its OWL-S service description. The invoker goes beyond the capabilities of the Mindswap OWL-S API [56] in that it is not limited to Document/Literal SOAP services.

The scope of this report was to provide an architecture for a distributed, semantic service registry. The demonstrator, however, goes beyond this and provides a framework for describing, finding and finally invoking a service. This is similar to the functionality provided by the OWL-S Virtual Machine [48], but in a peer-to-peer distributed environment, with the registry functionality at its core.

9.2 Registry Configuration

The registry prototype was designed to run within the JBoss application server. There are four main reasons for this:
- Registry functionality is normally provided by an application server. This demonstrates that the proposed registry is no different with that regard.

- The JBoss application server provides a highly configurable Micro-Kernel that makes it possible to define the required operation environment. One of the major features of this is that dependencies can be defined. The Query Service cannot operate if the Distribution Service is not present. This is something that can be defined in the server configuration.

- The JBoss application server provides a Java Naming and Directory Interface (JNDI) server. This is needed by clients and inside the distribution- and query service.

- The whole Micro-Kernel architecture is loaded as Java Management Extension (JMX) MBeans. This makes configuration and management easy, but also provides a direct solution to the Administration interfaces of both the distribution- and the query service.

Since JBoss is built on a JMX Micro-Kernel, the server can be configured to fit to the requirements present. Consider, for example, that the application server is used by a client application that only needs to query the registry. Such an environment would only need to be configured with the Distribution Service, which provides the Registry interface, and a JNDI server that is used to lookup the distribution service (see Figure 54).

Figure 54: JBoss client configuration.

If, alternatively, the application server operates on behalf of a service provider, the server needs to provide both the distribution- and query service. In addition the server may host the services that are described in the registry, something that requires databases, transaction managers, message queues and a range of other services normally required by middleware containers (see Figure 55).

Figure 55: JBoss service provider configuration.

Being able to tailor the application server to mirror the actual requirements of the user gives us the best of both worlds: the registry runs within an application server but does not need to be bloated when used by a client.
9.3 Query Service Implementation

The Query Service implementation provided by the demonstrator is designed as a remote accessible object according to Java RMI. As suggested by the conceptual architecture of the query service, it is designed with two prospective users in mind, namely the Distribution Service and a Java Management Extension (JMX) client. The Distribution Service invokes the Query Service through the remote interface QueryService. The Administration interface is invoked by a JMX client through a JMX MBean (see Figure 56).

![Interface and implementation of Query Service.](image)

The RMIQueryServiceImpl is the query service implementation provided by the demonstrator. It provides a remote interface QueryService to the Distribution Service. The implementation is loaded into the JBoss application service by wrapping it inside a JMX Mbean, which is loaded automatically by the JBoss kernel (see Figure 57).

![Wrapping QueryService inside JMX MBean.](image)

Once loaded the Query Service is configured according to a JBoss configuration file and bound to the application server's JNDI [57] registry. The MBean QueryServiceMBean serves as an adapter or mediator between the JBoss MBean architecture and the Query Service's Administration interface, allowing administration and management of the service using a JMX client.
9.3.1 **The QueryService remote interface**

The *QueryService* interface is designed to be used by the associated Distribution Service whenever an incoming service query arrives (see Figure 58).

![Diagram of QueryService usage](image)

**Figure 58: DistributionService usage of QueryService.**

The Distribution Service registers a *GroupHandler* object for every Jxta PeerGroup. The GroupHandler is responsible for listening for incoming queries distributed within the associated Jxta PeerGroup. When a query arrives it is passed on as a single argument to the `query` method on the Query Service. The method returns a `java.util.List` of OWL-S service description URIs, which are passed on to the querying peer.

Even though a *String* representation of the query argument is chosen to allow multiple query languages, the demonstrator only accepts RDQL [29] queries. The incoming query is passed on to the associated *Knowledge base*, which processes it against the hosted OWL-S service descriptions. If the knowledge base is able to find any matches for the query among the hosted service descriptions, the results are sent back to the originating peer.

The *QueryService* remote interface provides only one method `query`, which takes a single string argument representing a RDQL query:

```java
public interface QueryService {
    public List query(String query);
}
```

The method returns a `java.util.List` of OWL-S service description URIs, or an empty list if no matches are found among the hosted service descriptions.

9.3.2 **The Administration interface**

The *Administration* interface of the Query Service implementation provides methods for configuring the *Knowledge base*. The interface provides the following methods:

```java
public interface Administration {
    public void setConfigurationURL(URL config);
    public void setKnowledgebase(String knowledgebase);
    public void reconfigure() throws QueryServiceException;
    public void reconfigure(URL config) throws QueryServiceException;
}
```
The setKnowledgebase method provides a single argument representing the class name of the Knowledge base implementation used by the Query Service. The runtime configuration option is provided to enable experimentation. The content of the knowledge base is configured using a configuration ontology that is specified using the setConfigurationURL method. The method takes a single URL argument representing the location of the configuration ontology.

If either the setKnowledgebase method or the setConfigurationURL method is called during runtime using a JMX client, the Query Service needs to be reconfigured. This is handled by the reconfigure method. This method simply reloads the knowledge base using the updated configuration options. A convenient version of the method is also provided where a configuration ontology location can be specified as a single URL argument.

9.3.3 The Knowledge base

The main functionality of the Query Service is handled by the associated Knowledge base implementation. This decoupling is provided to enable experimentation with different ontology representation and query technologies. The implementation provided by the demonstrator provided a solution based on the Jena [52] framework. Jena is a popular solution when it comes to storing RDF-based ontologies, which includes OWL and OWL-S. In addition Jena provides a querying system that supports RDQL [29] as a querying language. Even though OWL-QL [30] is considered the better choice when it comes to querying OWL-based ontologies, the demonstrator only supports RDQL for now.

Jena supports database storage of ontology models. This means that ontologies do not need to be downloaded every time the knowledge base is initialized. This is not always desired, especially during testing of an OWL-S service description. Since different knowledge base implementations can be used by the Query Service, the demonstrator provides both a database-based and a memory-based Jena knowledge base (see Figure 59).

![Diagram of the association between Query Service and Knowledge base](image)

**Figure 59: Association between Query Service and Knowledge base.**

The knowledge base content is loaded by invoking the load method, which takes a single URL argument specifying a configuration ontology. This method is called by
the Query Service implementation using the configuration ontology URL specified using the Administration interface.

The main functionality provided by the knowledge base is accessed through the query method. The method takes a single string argument representing a valid RDQL query. When invoked, the knowledge base executes the query on its ontology data-model, and all matches, if found, are returned.

9.3.4 Distribution Service Association

The Query Service is not a stand-alone service, and requires the Distribution Service to be running in order to function. To take advantage of this dependency, the content of the query service’s knowledge base is used to configure the distribution service’s Query Router. The distribution service should only intercept incoming queries that are distributed within PeerGroups that are associated with the OWL-S Profile sub-classes of the OWL-S service descriptions hosted in the knowledge base.

During initialization of the query service, the PeerGroups associated with the hosted OWL-S service descriptions are registered with the distribution service using the method handleGroup, which is provided by the distribution service.

9.4 Distribution Service Implementation

The Distribution Service was implemented using Java and designed to provide three interfaces to its intended context. The service can be accessed remotely, using RMI, via its Registry interface by a client application. The Administration interface is used by a JMX client application to administer the service. When the Distribution Service operates on behalf of a service provider, it needs to be able to resolve incoming service queries. This is done by forwarding the queries to the Query Service, if the Query Service has notified its interest in queries. The Provider interface is provided by the Distribution Service to facilitate the establishment of an association between the Distribution- and Query Service (see Figure 60).

Figure 60: Implementation of Distribution Service interfaces.

The demonstrator provides one implementation of the Distribution Service, namely the DistributionServiceImpl class. It implements all interfaces defined in the conceptual architecture of the Distribution Service, and enables extensive administration through the Administration interface. The service is loaded into the JBoss application server by wrapping an instance of the class within a JMX MBean (see Figure 61).
Figure 61: Loading DistributionService into JBoss using MBean.

The DistributionServiceMBean serves two purposes: first it delegates the lifecycle management of the Distribution Service to the JBoss Micro Kernel architecture. This makes it possible to start, stop and pause the service using a JMX client; second the MBean mediates between the JBoss MBeanServer and the Distribution Service instance’s Administration interface. This allows invocation of the methods provided by the Administration interface through a JMX client. Thus, from outside the container there are two methods for invoking the Distribution Service. Either by Remote Method Invocation through the remote Registry interface, or using a JMX client (see Figure 62).

Figure 62: Distribution Service implementation environment.

In order for a client application to obtain access to the distribution service, it first looks up the distribution service, from the JBoss provided JNDI naming service, using a default name. If a reference is found, the client can make use of the provided registry functionality through the Registry interface.

Since JBoss provides a HTTP-client [59] to its JMX MBean server, the client has two methods of access to the Administration interface of the distribution service: through a web-browser and using a JMX client application.

9.4.1 The Query Router

The main functionality of the Distribution Service is related to the Jxta-based Query Router. The Query Router uses information contained in concrete OWL-S Profile ontology sub-classes to determine how to route a query on the Jxta network. In
order to support experimentation with different implementations for the Query Router, it was implemented as an *abstract* class with one implementation provided by the demonstrator (see Figure 63).

**Figure 63: Class-diagram of Query Router.**

In the demonstrator's *military ontology* several service profiles were defined using OWL. These definitions where also augmented with associations to Jxta *PeerGroups*. Consider, for example, the following sub-graph of the military ontology:

**Figure 64: Sub-graph of the military ontology illustrating Jxta relation.**

If a client is searching for an *ISAR* service, it invokes for example, the `getService` method on the *Registry* interface, with the following RDQL query:

```
SELECT ?service
WHERE {?service, rdf:type, service:Service),
   (?service, service:presents, ?profile),
   (?profile, rdf:type, nbd:ISAR)
USING service FOR <http://www.daml.org/services/owl-s/1.0/Service.owl#>,
nbd FOR <http://home.student.utwente.nl/r.p.bjornstad/owl/nbd.owl#>
```

By searching the query for the occurrence of the triple

```
(?profile, rdf:type, ?profileType)
```

the `profileType` variable should hold the value of the OWL-S *Profile* sub-class *ISAR*. Using this knowledge, the Query Router Knowledge base is consulted for the
associated Jxta PeerGroup, by extracting the jxtaGroup property value of the ISAR class. Consider the following sequence diagram illustrating the distribution of an incoming query:

![Sequence Diagram](image)

**Figure 65: Sequence Diagram illustrating interaction with Query Router.**

A client invokes the `getService` method using the `Registry` interface supplying a RDQL query as a single argument. The query is distributed on the Jxta network using the `Query Router`, which analyzes the query and consults the router `Knowledge base` for the Jxta PeerGroups to distribute to. The demonstrator only allows a query to be distributed within a single Jxta PeerGroup. This limitation does not respect the subclassing provided by OWL, and should be removed in a future implementation. The router initiates a `ResponseHandler` instance that is responsible for joining the destination Jxta PeerGroup, sending the query using its `Resolver Service` and waiting for a specified number of seconds. If a service provider is querying the registry, the local `Query Service` is also queried.

**The Router Knowledge base**

The Query Router is configured with a `Knowledge base` of OWL-S Profile sub-classes that have associated Jxta PeerGroups. The knowledge base is a Jena-based OWL model that is configured by a `configuration ontology` and store within a database. Consider, for example, the following configuration ontology fragment:

```xml
<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="&nbd;" />
</owl:Ontology>
```

The OWL `Ontology` header imports the `Network Based Defense` (NBD) ontology, which is the military example ontology provided by the demonstrator. The ontologies imported in this configuration ontology provide the `routing table` used by the Query Router. If a query is issued for a type of service that is not defined within the configuration ontology, the Query Router is not able to select a suitable Jxta PeerGroup to distribute in, and has to distribute the query to all peers using the `NetPeerGroup`.

The configuration ontology is configured by invoking the `setRouterConfigurationURL` on the Distribution Service’s `Administration` interface. The method takes as a single argument the URL of a valid OWL ontology, which uses the OWL import mechanism to populate the knowledge base.
9.4.2 The Provider Interface

The Provider interface is used by the Query Service to register interest in queries for a certain Jxta PeerGroup. The interface provides a single method:

```java
public void handleGroup(String peerGroupID)
```

To limit the amount of processing carried out by the Distribution and Query Service, the Distribution Service registers QueryHandler's for the Jxta PeerGroups that are associated with the OWL-S profiles of the hosted services. The interface lends its name to the fact that it is only used by the Query Service, and thus only by a service provider.

When the Query Service registers interest in a certain Jxta PeerGroup, the Distribution Service is responsible for joining that group, and listening for incoming queries. Thus a provider peer is a member of both the top-most Jxta PeerGroup NetPeerGroup, but also a member of all PeerGroups that the Query Service has registered interest in (see Figure 66).

![Figure 66: Association between Distribution Service and Query Service using PeerGroupHandler.](image)

When the Query Service invokes the `handleGroup` method on the Distribution Service’s Provider interface, a PeerGroupHandler is created that is responsible for receiving queries on the Jxta network and forwarding it to the Query Service. Consider, for example, the following sequence diagram that illustrates the establishment of an association between the Query Service and the Distribution Service when invoking the `handleGroup` method on the Provider interface (see Figure 67).

![Figure 67: Sequency Diagram illustrating handleGroup method.](image)

The Query Service initiates the association by invoking the `handleGroup` method on the Distribution Service’s Provider interface, supplying a single string argument representing the Jxta PeerGroup ID of the Jxta PeerGroup to register interest in. The Distribution Service instantiates the Jxta PeerGroup class using the `createGroup`
method of the GroupKit utility class. Then a GroupHandler is created to handle the group. This involves joining the Jxta PeerGroup and listening for incoming queries on that group.

A special Group Handler is created for handling the top-most Jxta groups NetPeerGroup and WorldPeerGroup. This makes it possible to receive and process queries that are sent by clients that do not have an up-to-date Query Router, and thus need to distribute their queries in these groups.

9.4.3 The Registry Interface

The Registry is, as the conceptual architecture of the Distribution Service suggests, a remote interface, and serves as the entry-point to the proposed registry for the user. In order for a client to use the registry, it first needs to obtain a reference to the Registry interface from JBoss’s JNDI [57] Naming service (see Figure 68).

![Figure 68: Client invocation of the Distribution Service.](image)

The client first queries the JNDI registry for the Distribution Service. When found, the Registry interface is used to query the proposed registry, for example, using the getService method. The Registry interface contains the following methods:

```java
public String getService(String query)
public String getService(String query, long timeout)
public String getAllServices(String query)
public String getAllServices(String query, long timeout)
```

Since the queries are distributed on the Jxta network, there is no way of telling how long it takes before a result is available. The demonstrator can be configured with default waiting times for single and multiple service queries using the Distribution Service’s Administration interface. If more control is needed, the timeout value can be supplied as a second argument to the registry methods.

9.4.4 The Administration Interface

The Distribution Service’s Administration interface provides methods for configuring and managing the service. The following methods are provided by the interface:

```java
public void setJxtaHome(String home);
public void setJxtaUsername(String username);
public void setJxtaPassword(String password);
public void setSingleServiceTimeout(long timeout);
public void setMultipleServiceTimeout(long timeout);
public void setRouterConfigurationURL(URL config);
public List listGroupsHandled();
```

The Jxta runtime environment creates a directory of configuration files, which includes a cache of advertisements. The location of this directory can be configured using the setJxtaHome method. The setJxtaUsername and setJxtaPassword methods configure
A Peer-to-Peer Distributed Semantic Service Registry

the security principals required by the Jxta configuration. Being able to configure these aspects of the Jxta runtime environment makes it easy to run multiple instances of the proposed registry on the same host.

The `setSingleServiceTimeout` method is used to configure the default timeout value of the Registry interface methods `getService`. The `setMultipleServiceTimeout` method does the same for the `getAllService` method.

The `setRouterConfigurationURL` provides the URL of a configuration ontology used by the Query Router to initialize its knowledge base. Currently, this can only be configured during startup.

The `listGroupsHandled` is a convenience method provided to support diagnostics. The method returns a list of all Jxta PeerGroup identifiers.

As mentioned, the demonstrator presents the Administration interface through a JMX MBean. Since JBoss's Micro Kernel architecture loads all components as MBeans, these configuration methods are called by JBoss during startup. The argument values are extracted from the MBean configuration file. Consider, for example, the following MBean configuration:

```xml
<mbean
code="sdsr.distribution.jmx.DistributionService"
name="jboss.sdsr:service=DistributionService">
<attribute name="JxtaUsername">password</attribute>
<attribute name="JxtaPassword">password</attribute>
<attribute name="JxtaHome">.provider</attribute>
<attribute name="RouterConfigurationURL">
  resource:ontologies/router.owl
</attribute>
<depends>jboss:service=Naming</depends>
</mbean>
```

The JBoss MBean architecture uses reflection to match configuration attributes with MBean methods. This way the MBean attribute `JxtaUsername`'s value is used as an argument to the MBean method `setJxtaUsername`.

9.5 Sample Ontology of Concepts

In order to demonstrate the description of services using OWL-S, as well as facilitating the query of those services, a sample-ontology was created for the demonstrator. This ontology is concentrated around different kinds of military sensor services (see Figure 69).
The arrows in Figure 69 represent rdfs:subClassOf relations between OWL classes. The Component class is the top-most class of the example ontology. This class is not directly used within the demonstrator, but serves to emphasize that all services can be described using OWL-S, not only sensors. The Component class is a sub-class of the OWL-S Profile class, which allows one to describe services using the provided sample ontology. In the Demonstrator, the Sensor class, or one of its derivatives is used to describe services.

9.5.1 The Sensor class

The Sensor OWL class serves as the super-class of all specific sensor concepts. In the sample ontology the sensor requires that three properties are defined by all instances. These are:

- detectsDomain;
- hasOutput;
- hasClassificationCapability.

Several more required properties could be included, but these serve as good examples of how the OWL Restrictions can be used to impose certain requirements on the class instance. In the sample ontology the availability of these properties are handled by the sub-classing of the Restriction class:

```xml
<owl:Class rdf:about="#Sensor">
  <rdfs:subClassOf>
    <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="#detectsDomain" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This restriction requires all instances of the Sensor class, or one of its derivatives to provide a detectsDomain property. The same approach is used for the other sensor methods.

If a client needs a sensor service capable of detecting objects on the ground, it may not be interested in what type of service it is. Consider, for example, the following RDQL query:

```rdql
SELECT ?service
```

---

Figure 69: Graph of military sample ontology (OWL-version of [71]).
The above query can return any sensor service that detects objects on the ground. Since OWL provides an object-oriented hierarchy of classes, an instance of an \texttt{ActiveHullMountedSonar} is also an instance of a \texttt{Sensor} and thus needs to provide the \texttt{detectsDomain} property.

### 9.5.2 The Domain class

The \texttt{detectsDomain} property relates a \texttt{Sensor} class instance with a \texttt{Domain} class. These are ontology concepts representing the different types of domains in which a sensor detects objects. The \texttt{detectsDomain} property is defined as follows:

\begin{verbatim}
<owl:ObjectProperty rdf:ID="detectsDomain">
  <rdfs:domain rdf:resource="#Sensor"/>
  <rdfs:range rdf:resource="#Domain"/>
</owl:ObjectProperty>
\end{verbatim}

The following \texttt{Domain} sub-classes have been defined in the sample ontology, and can be used as the value of the \texttt{detectsDomain} property:

\begin{verbatim}
&nbd;#Domain  
&nbd;#Ground  
&nbd;#Air  
&nbd;#Space  
&nbd;#CyberSpace  
&nbd;#MaritimeSurface
\end{verbatim}

\textbf{Figure 70: The Domain hierarchy of OWL classes.}

Consider, for example, a sensor service that covers the ground. The following ontology fragment describes this situation for an ISAR sensor:

\begin{verbatim}
<nbd:ISAR rdf:ID="ISAR-13_PROFILE">
  ...
  <nbd:detectsDomain rdf:resource="#Ground"/>
  ...
</nbd:ISAR>
\end{verbatim}

It can be argued that the \texttt{detectsDomain} property should not be required for a \texttt{Sensor} instance. After finding a service a \texttt{getDetectsDomain} method could produce this information. The property, however, is actually provided to support service queries, and not to impose meta-data information on a service.

### 9.5.3 The SensorOutput class

The \texttt{hasOutput} property relates a \texttt{Sensor} class instance with a \texttt{SensorOutput} class. The sample ontology provides several sub-classes of the \texttt{SensorOutput} class, which represent the type of output a sensor produces (see Figure 71).

\begin{verbatim}
&nbd;#SensorOutput  
&nbd;#Plot  
&nbd;#Bearing  
&nbd;#Track
\end{verbatim}

\textbf{Figure 71: OWL graph of SensorOutput hierarchy.}
Consider, for example, a sensor service that produces a movement track of a detected object. The following ontology fragment describes this situation for a SAR sensor:

```xml
<nbd:SAR rdf:ID="SAR-24">
  ...
  <nbd:hasOutput rdf:resource="&nbd;#Track"/>
  ...
</nbd:SAR>
```

If a client application aims to display a movement track of an identified object, it is not interested in services that cannot provide the Track output. The hasOutput helps the client providing more specific service queries.

### 9.5.4 The ClassificationCapability class

The hasClassificationCapability property relates a Sensor class instance with a ClassificationCapability class. The sample application provides several sub-classes of the ClassificationCapability class, which represent the object classifier provided by the sensor (see Figure 72).

![Figure 72: OWL graph of ClassificationCapability hierarchy.](image)

Consider, for example, a sensor service that is able to classify the type of a detected object. The following ontology fragment describes this situation for a PassiveRangingSonar sensor:

```xml
<nbd:PassiveRangingSonar rdf:ID="">
  ...
  <nbd:hasClassificationCapability rdf:resource="&nbd;#Type"/>
  ...
</nbd:PassiveRangingSonar>
```

If a situation emerges where the nearest AWAC plane needs to be detected, being able to separate the type of an air born target is a requirement. The hasClassificationCapability property helps the client specifying a query that is able to find a sensor capable of determining the type of a detected object.

### 9.5.5 The Jxta Ontology

In addition to the classifying different types of sensors, the sample ontology provides properties and values to help the Distribution Service’s Query Router. This is done by relating a sensor class with an OWL-based Jxta PeerGroup Advertisement. Consider, for example, the following ontology fragment:

```xml
<owl:Class rdf:about="#ISAR">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&jxta;#jxtaGroup"/>
      <owl:hasValue rdf:resource="#ISARGroup"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```
The above fragment restricts all instances of the ISAR sensor class to provide the `jxtaGroup` property with the resource `ISARGroup` as its value. The `ISARGroup` resource is an instance of the OWL-based PeerGroup advertisement:

```xml
<jxta:PeerGroupAdvertisement rdf:ID="ISARGroup">
  <jxta:GID>urn:jxta:uuid-3CAA9CE2D8C440579E24DBBAC1D1F39302</jxta:GID>
  <jxta:MSID>Unspecified</jxta:MSID>
  <jxta:groupName>ISAR JxtaGroup</jxta:groupName>
</jxta:PeerGroupAdvertisement>
```

By relating all instances of the ISAR sensor class with a specific Jxta PeerGroup, the Query Router is able to distribute queries within that group, and thus only peers that host ISAR services will receive the query.

### 9.6 Java Class Library of Concepts

To make working with the sample ontology easier, a client-side Java class library version of the sample ontology is provided by the demonstrator. This class library includes Java classes for the OWL Object- and Datatypes defined in the sample ontology, Sensor-specific proxy classes and handlers for the various `SensorOutput`, `ClassificationCapability` and `Domain` classes. The class library is organized into the following packages (see Figure 73).

![Package hierarchy of sample ontology class library.](image)

Figure 73: Package hierarchy of sample ontology class library.

The class library was organized into a hierarchy similar to the hierarchy of the OWL-based military sample ontology. The library is not complete. Only the constructs actually used by the demonstrator application are fully functional.

#### 9.6.1 nbd.datatypes

The `nbd.datatypes` package contains the Java classes of the OWL Object- and Datatypes defined in the military sensor ontology. These classes are used as argument types when demonstrating the functionality of the `JavaInvoker` and the `EJBIInvoker` of the OWL-S Invocation Framework provided by the Demonstrator. The classes shown in Figure 74 are provided in the package.

![Datatypes classes.](image)

Figure 74: Datatypes classes.

When serialized into XML the classes in Figure 74 produce valid OWL instances of its associated OWL class from the military sample application. Consider, for example, the following serialization of a `Coordinate` instance):
By using the class library for development, tedious XML processing is avoided, but with the same result.

### 9.6.2 `nbd.component.sensor`

The `nbd.component.sensor` package provides Java classes of the OWL `Sensor` classes from the military sensor ontology. The classes shown in Figure 75 are provided by the package.

**Figure 75: Sensor classes.**

For the classes coloured grey in Figure 75 the demonstrator provides *registry proxies*. These proxies are used to invoke methods on services found using the proposed registry. Consider, for example, the sequence diagram in Figure 76, which illustrates how a client obtains a reference to an ISAR service using the registry.

**Figure 76: Sequence diagram illustrating ISAR proxying.**

The *ISARProxy* handles all interaction with the *registry*. This involves creating a query designed to find an ISAR service on the peer-to-peer network. The proxy should return an ISAR interface to the client if a service URI can be found using the registry. The proxies shown in Figure 77 are provided by the demonstrator.
9.6.3 nbd.output

The nbd.output package contains Java classes of the OWL SensorOutput classes from the military sensor ontology. These classes represent the actual output produced by a sensor. For each type of sensor output defined by the sample ontology, the class library provides an output handler (see Figure 78):

Figure 78: Sensor Output classes.

The demonstrator application provides a scenario where an airplane is tracked. For this scenario, only sensors with Track output are considered. For this reason only the TrackHandler has been implemented with useful functionality.

When a client obtains a sensor proxy for a sensor that provides track output, the client can use its associated TrackHandler to invoke the single method getTrack. The method takes a component identifier as input and returns an instance of the Track class from the nbd.datatypes package (see Figure 79).
A Peer-to-Peer Distributed Semantic Service Registry

Figure 79: Sequence diagram of TrackHandler invocation.

The TrackHandler utilizes the OWL-S Invocation Framework class OWLSInvoker (see section 9.8) in order to invoke a service. The OWLSInvoker is used to invoke methods on a SOAP, EJB or local Java service, using the information contained in the service’s OWL-S description. This means that the ISARProxy can invoke methods on a service, regardless of the service's implementation details.

9.6.4 nbd.domain

The nbd.domain package provides Java classes of the OWL Domain classes from the military sensor ontology. The classes shown in Figure 80 are provided in the package.

Figure 80: Sensor Domain classes.

None of the classes in Figure 80 are actually used in the demonstrator. For this reason they provide no working functionality.

9.6.5 nbd.classification

The nbd.classification package provides Java classes of the OWL ClassificationCapability classes from the military sensor ontology. The classes in Figure 81 are provided in the package:

Figure 81: Classification Capability classes.

The demonstrator application makes no use of the classes in Figure 81. For this reason they provide no working functionality.
9.7 The Sample Services

The demonstrator provides four sample services with an associated OWL-S service description. Each service provides a different Sensor sub-class service, but all four provide Track as sensor output. In the demonstrator a scenario is created where the registry is used to locate a sensor capable of providing the movement track of an identified enemy airplane. Since all sample services are capable of delivering this information, it should not matter to the client which one it uses. The demonstrator provides the following sample services:

- **ISAR-13**
  - Provides an ISAR sensor service as a SOAP [34] accessible Web Service [03].

- **SAR-24**
  - Provides a SAR sensor service as an Enterprise JavaBean bound to a specified JNDI [57] registry.

- **Radar-78**
  - Provides a Radar sensor service as a SOAP accessible Web Service.

- **PRS-1970**
  - Provides a Passive Ranging Sonar sensor service accessible as a Java class in the local classpath.

All services come with an associated OWL-S service description, which uses WSDL [04] with Web Services Invocation Framework (WSIF) [60] extensions for concrete grounding. The service providers provide a dynamic WSDL document describing the service. This makes it possible to experiment with different distributions of the services without updating addressing information by hand.

To simulate the movement of the enemy airplane, a fifth service is created which emits the airplane position information. Whenever a sensor service is asked to deliver tracking information, it forwards the request to the position service and returns the result (see Figure 82).

![Figure 82: Simulation of enemy airplane position.](image)

When the `getEnemyTrack` method is invoked on the SAR-24 service, the service’s implementation invokes the `getPosition` method on the Airplane Position Service, and returns the result.
9.7.1 ISAR-13

The ISAR-13 service is implemented as a simple Java class, which is accessible as a SOAP Web Service using Apache Axis [61]. In order to demonstrate that the implementation details of the service should not make an impact on the ability to find and use it, the service is not designed to share any common datatypes with other systems.

The service is simple and provides only a single method:

```java
public Track getEnemyTrack(String targetId)
```

By supplying a valid target identifier as a single argument, the service returns the current tracking information of the simulated enemy airplane. The service's WSDL description makes use of an ISAR-13 specific XML Schema [62] for describing the `Track` datatype. The rationale behind not using any shared XML Schema for all services is that this gives a more accurate representation of a realistic situation.

Since the WSDL document for this service is accessible to the client, a XML Schema compiler can be used to provide client side stubs for the service's datatypes. The WSDL document is again referenced by the service's OWL-S description. This description, however, describes the service in terms of common OWL-based class concepts, which are used to describe all sensor services.

**WSDL Description**

The WSDL description of ISAR-13 provides an interface, or `portType`, which contains a single operation `GetEnemyTrack`:

```xml
<wsdl:portType name="ISARInterface">
  <wsdl:operation name="GetEnemyTrack" parameterOrder="componentId">
    <wsdl:input message="isar:GetTrackRequest" name="GetTrackRequest"/>
    <wsdl:output message="isar:GetTrackResponse" name="GetTrackResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

The operation takes a single-part message as an input, and provides a single-part message as output:

```xml
<wsdl:message name="GetTrackRequest">
  <wsdl:part element="typens:componentId" name="componentId"/>
</wsdl:message>

<wsdl:message name="GetTrackResponse">
  <wsdl:part element="typens:track" name="track"/>
</wsdl:message>
```

As mentioned, the ISAR-13 service provides its own distinct XML Schema for representing datatypes. The same XML Schema also defines the `componentId` and `track` message part element:

```xml
<xsd:element name="componentId" type="xsd:string"/>
<xsd:element name="track" type="typens:Track"/>
```

The `componentId` operation argument is a simple XML Schema `string` representing the target identification of the component that is to be tracked. The `track` argument, however, is of a complex XML Schema type, customized for the ISAR-13 service:

```xml
<xsd:complexType name="Track">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:string" minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```
A Peer-to-Peer Distributed Semantic Service Registry

The Track datatype provides a list of coordinates of an identified target. The service implementation makes use of Java classes for representing the Target-related concepts. These classes, however, are generated from the XML Schema using the Axis schema compiler, thus Axis generates Java class instances based on the supplied XML arguments provided by a SOAP request.

The WSDL description of ISAR-13 provides a single binding to Document/Literal SOAP:

The specified binding can be invoked, when running at address 192.168.2.2, at the specified service port:

The service is packed as an Apache Axis J2EE [63] Web Application running inside JBoss`s Web container at port 8080.

**OWL-S Description**

The OWL-S service description of the ISAR-13 service provides a single atomic process that maps on to the GetEnemyTarget SOAP operation. The OWL-S description, as opposed to the WSDL description, defines process arguments in terms
of OWL concepts from the military sensor ontology, and does not define any custom class concepts.

The OWL-S Profile instance of the service is defined using the sample ontology within the Sensor class hierarchy:

```xml
<nbd:ISAR rdf:ID="ISAR-13_PROFILE">
  <service:presentedBy rdf:resource="&s;#ISAR-13_SERVICE"/>
  <profile:serviceName>ISAR-13 Operation Service</profile:serviceName>
  <profile:textDescription/>
  
  <nbd:detectsDomain rdf:resource="&nbd;#Ground"/>
  <nbd:hasOutput rdf:resource="&nbd;#Track"/>
  <nbd:hasClassificationCapability rdf:resource="&nbd;#Unique"/>

  ... 
</nbd:ISAR>
```

By defining the service profile as an instance of the ISAR class, a client looking for an ISAR service is able to find it using the proposed registry. Since the demonstrator scenario involves the tracking of an enemy airplane, the hasOutput property refers to the Track sub-class of the SensorOutput class defined in the sample ontology. Thus, a client looking for a sensor capable of delivering tracking information is able to find ISAR-13 based on this information.

As mentioned above, the OWL-S Process Model defines a single atomic process for the service:

```xml
<process:AtomicProcess rdf:ID="GetTrack">
  <process:hasInput>
    <process:Input rdf:ID="GetTrack_componentId_INPUT">
      <process:parameterType rdf:resource="&xsd;#string"/>
    </process:Input>
  </process:hasInput>
  
  <process:hasOutput>
    <process:UnConditionalOutput rdf:ID="GetTrack_track_OUTPUT">
      <process:parameterType rdf:resource="&nbd;#Track"/>
    </process:UnConditionalOutput>
  </process:hasOutput>
</process:AtomicProcess>
```

The process GetTrack takes a single input argument of type XML Schema string, and a single unconditional output value of type Track from the sample ontology. When invoking the service, the invocation framework needs to mediate between the OWL-based Track concept from the sample ontology, with the custom XML Schema Track datatype used by ISAR-13s SOAP access point. How this is done is entirely up to the OWL-S Grounding ontology:

```xml
<grounding:wsdlOutputMessageParts rdf:parseType="Collection">
  <grounding:WsdlOutputMessageMap>
    <grounding:owlsParameter rdf:resource="&pm;#GetTrack_track_OUTPUT"/>
    <grounding:wsdlMessagePart>&wsdl;#track</grounding:wsdlMessagePart>
  </grounding:WsdlOutputMessageMap>
</grounding:wsdlOutputMessageParts>
```

```
<!--[CDATA[
```
The OWL-S Grounding of ISAR-13 includes an XSL [64] stylesheet that defines how to translate between the Track concept of the sample ontology, and the Track concept used by ISAR-13's SOAP access point.

9.7.2 Radar-78

The Radar-78 service is implemented as a single Java class, accessible as an Apache Axis SOAP service. The service is almost identical to ISAR-13, but to demonstrate how OWL-S described services can interoperate, the service provides its own XML Schema describing its datatypes.

Another difference between Radar-78 and ISAR-13 is the OWL-S Profile sub-class used to instantiate the profile ontology:

```
<nbd:Radar rdf:ID="Radar-78_PROFILE">
  <service:presentedBy rdf:resource="&s;#Radar-78_SERVICE"/>
  <profile:serviceName>Radar-78 Operation Service</profile:serviceName>
  <profile:textDescription/>
  ...
</nbd:Radar>
```

The nbd:Radar is actually the super-class of both nbd:SAR and nbd:ISAR. This means that a client looking for a service with Radar functionality can expect to receive a reference to SAR-24, ISAR-13 and Radar-78. Which one is chosen should be totally transparent to the client.

9.7.3 SAR-24

The SAR-24 service is implemented as an Enterprise JavaBean (EJB) [65] service. In a typical EJB-based system, the service client possesses the client-side classes required to invoke methods on this service. However, in order to integrate an EJB service into the proposed registry, the service needs to be described using OWL-S, and simply by its Java-based interfaces. OWL-S makes no distinction between a SOAP service and an EJB. OWL-S describes a service in terms of transparent processes, which are grounded according to an associated WSDL service description. The SAR-24 OWL-S service description is much like the description of ISAR-13. The main difference lies in the classification of the OWL-S Profile instance used by the profile ontology:

```
<nbd:SAR rdf:ID="SAR-24_PROFILE">
  <service:presentedBy rdf:resource="&s;#SAR-24_SERVICE"/>
  <profile:serviceName>SAR-24 Operation Service</profile:serviceName>
  <profile:textDescription/>
  ...
</nbd:SAR>
```

The nbd:SAR OWL-S Profile sub-class instance makes it possible for clients to find SAR-24 when looking for a SAR service.

In order to describe the SAR-24 EJB using WSDL, the WSIF [60] extensions to WSDL are used. In addition an XML Schema version of the argument types used by SAR-24 needs to be defined. Like with ISAR-13, the XML Schema is embedded within the
Types element of the WSDL description. EJB-related extensions to WSDL are first encountered in the PortType binding of the WSDL description:

```xml
<wsl:binding name="SAR-24EJBBinding" type="sar:PRSInterface">
  <ejb:binding/>
  ...
</wsl:binding>
```

The above fragment states that the PRSInterface PortType of the SAR-24 service description is bound to a concrete EJB. Since WSDL described method invocations in terms of input, output and fault messages, WSIF also provides extensions for defining mappings between message part types and Java classes:

```xml
<format:typeMapping encoding="Java" style="Java">
  <format:typeMap
typeName="typens:Track"
  formatType="brylex.service.Track" />

  <format:typeMap
typeName="xsd:string"
  formatType="java.lang.String" />
  ...
</format:typeMapping>
```

The above fragment states that input message parts of type xsd:string are marshalled into instances of java.lang.String when invoked. Analogue, an EJB output of type brylex.service.Track is unmarshalled into an XML element of type typens:Track. The input and output message parts are mapped to EJB method arguments as defined in the following fragment:

```xml
<wsl:operation name="GetTrack">
  <ejb:operation
    methodName="getTrack"
    parameterOrder="componentId"
    interface="remote"
    returnPart="track" />
  <input name="GetTrackRequest" />
  <output name="GetTrackResponse" />
</wsl:operation>
```

The above fragment binds the WSDL PortType operation GetTrack to the EJB operation getTrack as defined by the methodName attribute of the ejb:operation element. The method takes a single argument componentId, as defined by the parameterOrder attribute, which refers to the input message GetTrackRequest part with the same name. Analogue, the track part of the output message GetTrackResponse is defined as the result of the operation.

The WSDL service binding is, as with a SOAP service, accessible from a defined service port:

```xml
<wsl:port name="SAR-24" binding="sar:SAR-24EJBBinding">
  <ejb:address
className="brylex.service.interfaces.SAR24RemoteHome"
jdniName="ejb/component.sensor.radar.sar.SAR-24"
initialContextFactory="org.jnp.interfaces.NamingContextFactory"
jdniProviderURL="jnp://192.168.2.2" />
</wsl:port>
```

The ejb:address element provides the information needed to lookup and invoke methods on the specified EJB service.
Even though WSDL can be used to describe an EJB service, the availability of the client-side EJB classes needs to be in the \textit{invoker} classpath. This means that invocation is not totally transparent, but by using this approach it is possible to integrate the service description into the proposed registry.

9.7.4 \textit{PRS-1970}

The \textit{PRS-1970} service is actually a simple Java class instance, which is described using OWL-S and WSIF extensions to WSDL. Having little practical use, it serves as an example of how the proposed registry is able to find almost any types of services.

Like with the other three services, the OWL-S service description is transparent to the implementation details of the service. The concrete binding to a Java class in the client’s classpath is handled in the associated WSDL document.

Like with the SAR-24 EJB service, the WSDL description utilizes WSIF extensions for providing a PortType binding and a service port. The binding is specified almost identically to the EJB binding extension used by SAR-24. The service is accessible by a local Java class bound service port:

\begin{verbatim}
<wsdl:port name="PRS-1940" binding="prs:PRS-1940JavaBinding">
  <java:address className="brylex.service.PRS1940"/>
</wsdl:port>
\end{verbatim}

The Java class \texttt{brylex.service.PRS1940} needs to be in the client’s classpath.

9.8 OWL-S Invocation Framework

The demonstrator provides an \textit{OWL-S Invocation Framework} that makes it possible to invoke services using their OWL-S description. The framework was provided to demonstrate similar functionality as the \textit{DAML-S Virtual Machine} [48], but also because it enables highly dynamic semantics systems to be built (see Figure 83).

![Figure 83: The place of the OWL-S Invoker in a complete architecture.](image)

A client uses the proposed registry to find the OWL-S service description of an \textit{ISAR} service. Using the \textit{OWL-S Invoker} the client is able to invoke methods on the found service, by simply using its OWL-S service description. Even though this functionality is not a part of the overall architecture of the proposed registry, it serves as a good example of possible applications of the architecture, and is an integral part of the demonstrator application.

The OWL-S Invocation Framework is built upon the \textit{Web Services Invocation Framework} (WSIF) [60] by Apache [66]. This technology makes it possible to invoke a \textit{Web Services} [03] method, directly, only using its WSDL description. This differs from popular Web Services technologies like \textit{Java Web Services Developer Pack} [67].
and Apache Axis [61]. The approach of these two technologies is to generate Java classes based on a services WSDL document. These classes are then used to create a service specific application.

WSIF also provides extensions to WSDL for describing concrete protocol bindings to, among others, Enterprise JavaBeans (EJB) and local Java classes. Thus, using WSIF it is possible to describe an EJB using WSDL, and then invoke the EJB directly using the content contained in the WSDL description.

9.8.1 The Architecture

During the initial stages of the OWL-S Invocation Framework development, a custom Java API was created for manipulating OWL-S service descriptions. Later, however, Mindswap provided an OWL-S API [56], which I adopted for OWL-S Invocation Framework. The idea was to provide the user of the framework with an OWL-S centric interface, and use WSIF internally to actually invoke a service (see Figure 84).

Figure 84: Usage of the Web Service Invocation Framework.

The problem with this approach is that OWL-S describes process parameters, in most cases, in terms of OWL object- and datatypes. WSDL, however, relies on XML Schema Datatypes (XSD) [62]. When a client invokes a service process, it supplies OWL instances as its parameters. The OWL-S Invocation Framework is responsible for translating these parameters to XSD instances, and passing them on to a WSIF Invoker.

The OWL-S Invocation Framework provides the classes shown in Figure 85, which are responsible for making it possible to invoke a service based on an OWL-S service.

Figure 85: The main classes provided by the OWL-S Invocation Framework.
The **OWLInvoker** class is the main entry point for a client. This class is responsible for translating all provided parameters to XSD, and delegating further invocation on to the WSIF-based **DynamicInvoker** class.

### 9.8.2 OWL-S Invoker

As mentioned, the **OWLInvoker** class serves as the entry point for a client when invoking a method on a service based on its OWL-S service description. The class provides a single method **invoke**, which takes the following two arguments:

- **serviceURI**, which takes a `java.net.URI` instance referencing the OWL-S Service instance of the described service.
- **Parameters**, which takes an OWL-S API `ValueMap` instance containing `org.jdom.Element` instance values for each process parameter.

The OWL-S Invoker does not support **composite** processes. This means that it only supports OWL-S service description with a single **Atomic Process**, thus defining which process to invoke is not needed. The Atomic Process referenced in the **ProcessModel** of an OWL-S service description is automatically invoked. Consider, for example, the following ontology fragment:

```xml
<process:ProcessModel rdf:ID="ISAR-13_PROCESS">
  <service:describes rdf:resource="&s;#ISAR-13_SERVICE"/>
  <process:hasProcess rdf:resource="#GetTrack"/>
</process:ProcessModel>
```

The **process:hasProcess** property references the **#GetTrack** Atomic Process as the services only process. This is a serious limitation of the prototype registry, since a service can only provide a single method. Thus, for the demonstration purposes, the ISAR-13 service provides a single method **GetTrack**, which can be invoked using the OWL-S Invoker:

```xml
<process:AtomicProcess rdf:ID="GetTrack">
  <process:hasInput>
    <process:Input rdf:ID="GetTrack_componentId_INPUT">
      <process:parameterType rdf:resource="&xsd;#string"/>
    </process:Input>
  </process:hasInput>
  ...
</process:AtomicProcess>
```

The **#GetTrack** atomic process defines a single input parameter of type XSD string. If a client wishes to get the current track of the object **Flight-78**, it supplies the following RDF instance as the input parameter value in the **ValueMap**:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:nbd="http://home.student.utwente.nl/r.p.bjornstad/owl/nbd.owl#">
  <Flight-78>
</rdf:RDF>
```

This RDF-centric encoding of parameters needs to be translated into a format suitable for WSIF invocation. This is handled by means of XSLT transformations. Since the above parameter value is of a well-known XSD string value, XSLT transformation is not really needed. For the demonstration purposes, however, all parameter types undergo XSLT transformation. Consider the following OWL-S **Grounding** fragment:
The grounding:Wsd1InputMessageMap states that the OWL-S process parameter GetTrack_componentId_INPUT maps on the componentId message part of the WSDL description. The mapping is by translating the supplied RDF instance value using the XSL [64] stylesheet defined in the grounding:xsltTransformation property.

The OWL-S Invoker translates all process parameters using its associated XSL stylesheet. The translated values are included into an XML input document, which is supplied as an argument to the WSIF-based Dynamic Invoker. Each translated value is included as a child of an element with the same name as its associated WSDL message part:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <componentId>
    Flight-78
  </componentId>
</input>
```

Using XSLT, OWL-based process parameters are translated into values suitable for WSDL message parts. The same translation is carried out on the response value received from the Dynamic Invoker (see Figure 86).
The DynamicInvoker invokes a service method based on its WSDL description. This means that the OWL-S service description is not needed by the DynamicInvoker, but a reference to the service’s WSDL document is sufficient.

9.8.3 WSIF Dynamic Invoker

The WSIF Dynamic Invoker is based on the DynamicInvoker client supplied by WSIF. This client, however, is only capable of handling simple parameter types. The implementation of the DynamicInvoker provided by the OWL-S Invocation Framework is far more robust. The OWLSInvoker class has a single method invoke with the following arguments:

- `wsdlLocation` that takes a `java.net.URL` instance values referencing an accessible WSDL document describing the service to invoke a method on.
- `portType` that takes a `javax.xml.namespace.QName` instance value referencing the `portType` of the WSDL description.
- `operation` that takes a `java.lang.String` instance value of the operation name to invoke.
- `parameters` that takes a `org.jdom.Document` instance value containing the WSDL message part values for the operation in question.

The `wsdlLocation`, `portType` and `operation` values are normally extracted from the WSDL description of the service being invoked. The `parameters` value is an `org.jdom.Document` representing the XSLT translated process parameter values.

Using WSIF the DynamicInvoker analyzes the WSDL document, specified through the `wsdlLocation` argument, in order to determine if the WSDL document describes a SOAP, EJB or local Java class binding of the service. These are the only bindings supported by the OWL-S Invocation Framework, even though WSIF supports several other bindings. Depending on the type of binding used in the WSDL document, the DynamicInvoker forwards the request to one of the following concrete invokers:

- SoapInvoker;
- EJBInvoker;
- JavaInvoker.

SoapInvoker

The SoapInvoker handles, as the name suggests, invocations of SOAP services. Even though there are several styles of SOAP services available, the SoapInvoker only supports Document/Literal/style invocation, often supported by .NET Web Services [68]. The SoapInvoker follows the approach taken by the OWL-S API, where an Apache Axis [61] SOAP request is populated without any type checking of the message parts.
The SoapInvoker provides a single method `invoke` with the following arguments:

- `operation` that takes a `WSIFOperation_ApacheAxis` instance value representing the SOAP operation to invoke on the service.
- `inputMessageName` that takes a `java.lang.String` instance value representing the name of the operation's input message.
- `outputMessageName` that takes a `java.lang.String` instance value representing the name of the operation's output message.
- `parameters` that takes a `org.jdom.Document` instance that describes the value of each message part as valid XML instances.

When invoked the SoapInvoker uses the `operation`, `inputMessageName`, `outputMessageName` and `parameters` argument values in order to create an outgoing SOAP envelope. The `parameters` XML argument contains a root element `input`. The SoapInvoker simply includes each child of this root element in the body of the outgoing SOAP envelope.

When the response SOAP envelope is received, the process is reversed. An XML result document is created with a root element `output`. The child elements of the response SOAP envelope are extracted and inserted as children of the `output` element of the result document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<output>
  <track>
    <Track>
      <id>Flight-78</id>
      <position>
        <longitude>-30.34554</longitude>
        <latitude>30.54323</latitude>
        <altitude>23</altitude>
      </position>
    </Track>
  </track>
</output>
```

The result document is parsed as an `org.jdom.Document` instance and returned.

**EJBInvoker**

The `EJBInvoker` handles, as its name suggests, invocations of `Enterprise JavaBean` (EJB) services. The WSIF framework provides extensions to WSDL for defining the location of an EJB, as well as its class. This requires the client to have access to client-side stubs as well as JNDI drivers in its classpath. Consider, for example, the following WSDL port definition:

```xml
<wsdl:port name="SAR-24" binding="sar:SAR-24EJBBinding">
  <ejb:address
    className="brylex.service.interfaces.SAR24RemoteHome"
    jndiName="ejb/component.sensor.radar.sar.SAR-24"
    initialContextFactory="org.jnp.interfaces.NamingContextFactory"
    jndiProviderURL="jnp://192.168.2.2" />
</wsdl:port>
```

The `ejb:address` provides enough information to enable invocation of an EJB, as long as the client-side classes are available.
The main problem with WSDL-based EJB invocation is how to translate between XML argument values and Java class instances. WSIF provides extensions to WSDL for this as well:

```xml
<format:typeMapping encoding="Java" style="Java">
  <format:typeMap
    typeName="typens:Track"
    formatType="brylex.service.Track" />
  <format:typeMap typeName="xsd:string" formatType="java.lang.String" />
  ...
</format:typeMapping>
```

The `format:typeMap` defines the a mapping between a certain XSD datatype and a Java class, which need to be available in the local classpath. This mapping is important when invoking a EJB service. Consider the following WSDL binding operation:

```xml
<wSDL:operation name="GetTrack">
  <ejb:operation
    methodName="getTrack"
    parameterOrder="componentId"
    interface="remote"
    returnPart="track" />
  <input name="GetTrackRequest" />  
  <output name="GetTrackResponse" />
</wSDL:operation>
```

The above fragment specifies that the `portType` operation `GetTrack` is invokable as an EJB using the method `getTrack`. The method takes one argument, which maps on the `componentId` message part of the `GetTrackRequest` message. Thus, when invoking the `getTrack` method on the EJB, the value of the `componentId` message part needs to be translated into a Java instance of the correct class. The correct class is specified in the `format:typeMapping` WSDL extension.

The EJBInvoker provides a Marshaller for translating the XML-based input message part values into a Java class instance. It also provides an Unmarshaller for doing the reverse translation (see Figure 87).

**Figure 87: Type conversion using Marshaller and Unmarshaller.**

The Marshaller and Unmarshaller make use of the WSIF extensions to WSDL as well as the XML Schema contained in the service’s WSDL document to mediate between an XML instance document and Java class instances. The sequence diagram in Figure 88 illustrates how a client normally invokes an EJB method.
When a client needs to invoke the getTrack method on the EJB using the DynamicInvoker, it issues the invoke method on the DynamicInvoker. The XML argument contains the WSDL message part values for the request. These values are passed on to the Marshaller using the marshall method, which returns a Java class instances. These instances are then passed on as arguments when the getTrack method is issued on the EJB. The EJB returns a Java class instance, which is unmarshalled by the Unmarshaller, and returned to the client.

JavaInvoker

The JavaInvoker handles invocation of a WSDL described Java class located in the classpath. This has little practical use, but serves as an example of how all service types can be described and invoked using WSIF.

While a WSDL SOAP port delivers the network access point of SOAP service, the Java port simply describes the Java class of the service:

```xml
<wsdl:port name="PRS-1940" binding="prs:PRS-1940JavaBinding">
  <java:address className="brylex.service.PRS1940"/>
</wsdl:port>
```

The WSDL portType binding resembles the type used by the EJB extension to WSDL, but refers to methods on a local Java class:

```xml
<wsdl:operation name="GetTrack">
  <java:operation
    methodName="getTrack"
    parameterOrder="componentId"
    methodType="instance"
    returnPart="track" />
  <input name="GetTrackRequest"/>
  <output name="GetTrackResponse"/>
</wsdl:operation>
```

In the same way as with the EJBInvoker, the JavaInvoker needs to translate between XML message part values and Java class instance when invoking a method. This is handled in exactly the same way as with the EJBInvoker, using a Marshaller and an Unmarshaller.

9.9 The Demonstrator Application

The demonstrator application provides an environment for demonstrating the core functionality of the semantic distributed service registry. Making use of the sample ontology, Java class-library, example services and the OWL-S Invocation Framework,
the application demonstrates how the registry can be used to find a service based on its semantic capabilities and to invoke that service dynamically.

The following components participate in the demonstrator application:

- The **Client** application, which makes use of a **SensorProxy** object to find services on the network that are capable of delivering a sensor-based service. The definition of a sensor-based service is derived from the sample ontology’s OWL-S Profile sub-class **Sensor**.
- The **Airplane** service, which simulates an airplane. As described earlier, this service simply emits positioning information of an imaginary airplane.
- The **ISAR-13** sensor service, which is an ISAR service that is capable of delivering tracking information. The service is designed to fetch positioning information from the **Airplane** service.
- The **Radar-78** service, which is a Radar service capable of delivering tracking information. The service is designed to fetch positioning information from the **Airplane** service.
- The **SAR-24** service, which is a SAR service capable of delivering tracking information. The service is designed to fetch positioning information from the **Airplane** service.
- The **PRS-1940** service, which is a Passive Ranging Sonar service capable of delivering tracking information. The service is designed to fetch positioning information from the **Airplane** service.

All of the above mentioned components are *randomly* distributed on a network. The client application is designed to track an identified airplane, namely **Flight-78**, which is simulated by the **Airplane** service. The sequence diagram in Figure 89 illustrates the typical steps taken in order to retrieve the current position of **Flight-78**:  

![Sequence Diagram](image-url)  

**Figure 89: Sequence Diagram illustrating demonstrator application client.**
When the client application’s `getTrack` method is invoked, the `SensorProxyFactory` is asked to create an instance of the `SensorProxy` class. This involves using the proposed Registry to find a service, which provides sensor functionality and tracking capabilities. The ISAR-13 service is a possible result, as well as the other three OWL-S services provided by the demonstrator.

When found, the SensorProxyFactory returns a SensorProxy object that serves as a proxy to the ISAR-13 service. The ISAR-13 service provides tracking information as sensor output, thus a tracking specific output handler is returned when invoking the `getOutputHandler` method on the SensorProxy instance. The `TrackHandler` is responsible for invoking tracking methods on the associated service (ISAR-13) using the OWL-S Invocation Framework. When the TrackHandler invokes the `getTrack` method on the ISAR-13 service, using the OWL-S Invocation Framework, the `Airplane` service is queried for its current position, which is returned back to the client.

If the ISAR-13 service is taken off the network, the client should be able to find either SAR-24, Radar-78 or PRS-1940, which are all capable of delivering the same service to the client. This demonstrates the core functionality of the proposed registry, namely finding a service based on capabilities rather than implementation details.

### 9.10 Summary

The demonstrator is responsible for providing the building blocks needed to demonstrate the core functionality of the proposed semantic, distributed service registry. This involves not only providing an implementation for the registry itself, but also providing sample service and ontologies that makes it possible to test the registry.

The demonstrator provides an application that mimics a possible real-life scenario where the registry serves a valuable purpose. Without taking performance into consideration, this application demonstrates how a client can locate an OWL-S service located on the network, based on its semantic requirements. These semantic requirements are formulated as an RDQL query, and are used by the proposed registry to locate a service description on the Peer-to-Peer network that matches the query.

In order to demonstrate the possible gains in flexibility and dynamics, the OWL-S Invocation Framework is provided by the demonstrator. The framework enables dynamic invocation of OWL-S described service, transparent to their implementation details. The proposed semantic distributed registry’s functionality and usability is demonstrated by the demonstrator application, thus I conclude that the registry architecture is a feasible solution given the initial requirements.

In this chapter the various parts of the demonstrator was introduced. This included a prototype implementation of the proposed registry, as well as an advanced OWL-S invocation framework. In the next chapter important issues that were raised during this report are discussed, and finally a conclusion is given.
10 Discussion

10.1 Introduction

This thesis gives a conceptual architecture of a semantic distributed service registry. Two main requirements have been the driving forces for choosing the core technologies for the proposed registry solution:

1. Services should be found based on their semantic capabilities.
2. Centralization should be avoided in the solution.

These requirements imposed that service descriptions contained in the service registry should be described using a semantically rich language. Based on this, three languages for describing concepts using ontologies have been evaluated, as well as solutions for querying ontologies. I carried on by evaluating registry distribution technologies, with an emphasis on centralization avoiding solutions.

The Web Ontology Language (OWL) [08] has been considered as a sound choice for a semantically rich language for defining ontologies. The related OWL ontology for services (OWL-S) [06] has therefore been evaluated as a service description language. Peer-to-Peer networking using Jxta [18] has been considered as interesting technology for distributing registry functionality. I have used these technologies to describe a conceptual architecture of a service registry that is distributed on a Jxta-based Peer-to-Peer network, where services are described using OWL-S.

The purpose of this work has been to provide a proof-of-concept of a registry solution where centralization is avoided and services can be found based on their semantically defined capabilities. Several choices have been made, which are open for discussion. I have structured the discussion in the following way, according to the choices that were made:

- **OWL-S Service Descriptions**: this section discusses the choices that were made that led to choosing OWL-S as a language for describing services. This discussion includes issues with ontologies in general, as well as issues related to ontology query languages.
- **Jxta Peer-to-Peer Network Technology**: this section discusses the issues related to choosing Jxta Peer-to-Peer networking as a means for distributing a service registry.
- **The Proposed Conceptual Architecture**: this section discussed the issues related to the proposed architecture suggested in this thesis.
- **The Demonstrator**: this section discusses the demonstrator that was developed to prove some of the statements given in this thesis. The demonstrator provides an implementation of the core functionality of the proposed registry architecture, as well as an environment for demonstrating the usability of the registry.
- Finally a discussion of possible further work is carried out.

10.2 OWL-S Service Descriptions

OWL-S is an XML-based [22] language for describing services using OWL [08] constructs. OWL is an extension of RDF [23] and RDF Schema [09], which also are
languages for defining ontologies. The rationale behind choosing an OWL-based service description ontology over an RDF or RDF Schema-based solution, is that OWL provides a much richer mechanism for defining concept restrictions and concept relations. In the Semantic Web scenario described in [05], sharing of ontologies may become widespread in the future. Being able to define ontology concept equality and disjunction is a requirement that has to be met. OWL provides rich features for both sharing ontologies and expressing equality and disjunction between ontology concepts.

There are, however, disadvantages associated with using OWL in the proposed registry. OWL adopts RDF resource references using URIs [24]. On the Web this is an advantage, since all ontologies can be resolved from their resource address. In a Peer-to-Peer environment, however, static resource locations do not exist, or are avoided whenever possible. In a military context, and especially considering the requirements of the proposed registry, centralized resource identifiers form serious security risks. This thesis proposes a Jxta-based URI scheme, which makes it possible to resolve ontologies on the Peer-to-Peer network.

Using OWL-S for describing services is another major choice in the proposed registry. One of the major requirements for the registry was that services should be searchable based on their semantic properties. OWL-S describes a service in three ways. Firstly, it describes the service capabilities in a Profile ontology, which should contain enough information about a service to make it possible to determine if it suits the client’s requirements when it comes to capabilities. Secondly, it describes service primitives in terms of processes, which is a strategy adopted from BPML4WS [33]. This makes it possible to construct processes that cover several service primitives. Finally, service processes are grounded to a specific service. OWL-S provides a standard grounding to a WSDL [04] document, but the grounding can be customized to accommodate user needs.

Whereas the service profile description provides a searchable interface to a service, it does suffer from poor OWL-S specification choices. For example, the OWL-S Profile ontology provides the hasInput and hasOutput properties. These are convenience properties that make it possible to query for service inputs and outputs from the service profile description. In many cases, however, a service has several distinct processes, thus it is possibly better to classify process types rather than process parameter types.

There are two process types defined in the Process ontology, namely atomic processes and composite processes. For invocation of OWL-S described services without human interaction, services with a single atomic process is an easy case. However, if the service provides a choice of several processes, determining which one to invoke requires the invoking agent to analyze the Process ontology of the service, and thus the hasInput and hasOutput properties of the Profile ontology provides no added value.

The OWL-S Process ontology is still incomplete. For example, the specification defines constructs for describing process pre- and post conditions. How these conditions are defined, and how they should be processed, are not mentioned in the specification. There is, however, much work in progress in that area of the OWL-S specification.

One of the major advantages of OWL-S is that it is an OWL ontology. This makes it possible to reason about the content of a service description using powerful reasoning systems. The OWL-S Grounding ontology, however, provides a service
grounding that associates a service process with a WSDL document. This association hides a lot of grounding information contained in the WSDL document from the OWL reasoning software. For example, it is impossible to search for services that provide an Enterprise JavaBean (EJB) type service. This binding specific information is contained in the WSDL document, which is off-limits to the reasoning software. This thesis argues that an OWL-based version of WSDL should be created instead, which maps WSDL constructs directly to OWL constructs. This enables reasoning about the information contained in the WSDL document, for example, binding specific information.

I argue that the choice of a query technology should be up to the specific client. Depending on the need for semantic expressiveness, a query technology suitable for doing the job needs to be supported by the proposed registry. RDQL is a query language for querying RDF data models. By configuring the underlying data model, the support for simple RDF/S transitivity or OWL DL could be provided to the query reasoning architecture. RDQL, however, does not provide the semantic properties required for querying ontologies that make use of OWL cardinality restrictions. OWL-QL is a query language designed for OWL and the Semantic Web. With support for both variable binding schemes and asynchronous responses, OWL-QL is the most suitable choice for the proposed registry architecture.

10.3 Jxta Peer-to-Peer Network Technology

Creating a registry service does in most cases involve some kind of centralization. UDDI [02] provides an architecture for distributing multiple instances of the registry, while providing a uniform access point to the client. The solution, however, relies on a centralized administration service and is thus not always suitable. Choosing Peer-to-Peer networking as a means to distributing a registry introduces several issues. In the architecture provided in this thesis, communication is rather naïve. By avoiding all sources of centralization, methods for optimizing and controlling query distribution are difficult to implement. On systems with few peers the current solution is feasible. In case the network grows, so called super peers may help make communication more robust and more efficient. These super peers, however, introduce temporarily centralization in the system.

In the proposed architecture the Jxta resolver service is used to multicast (not IP multicast) query messages within a PeerGroup. The PeerGroup to which message are distributed, is information that is available at every peer, thus it can be considered a decentralized router mechanism. This allows focused distribution of queries, but does not make use of any bookkeeping functions at all. This means that a peer does not have information about its neighboring peers at any time. This is because such information might be stale, and should not be trusted. This is an approach that is rather naïve. In an operational system, being able to store local references to known peers should be allowed. The trick is to determine how long such references should be kept valid. This kind of bookkeeping is also something that can be carried out by local super peers.

10.4 The Proposed Conceptual Architecture

This thesis describes a conceptual architecture of a semantic, distributed service registry, based on OWL-S service description and Peer-to-Peer registry distribution. The OWL-S service descriptions contained in the registry are distributed to the peers providing the actual services. This approach results in a decentralized registry that is
always up-to-date because of automatic garbage collection. Whenever a peer goes offline, the information it provided to the registry disappears.

The architecture provides a service query router, which is based on Peer-to-Peer grouping features found in, for example, Jxta. The query router determines how to route a specific query based on information contained in the OWL-S service descriptions. This provides a decentralized router, but introduces router specific constructs to a service-specific ontology. This should be avoided in a production system by making use of concept sub-classing features provided by OWL.

The query router relies on ontology reasoning, a process that is normally computation intensive. It is rather naïve to think that all service providers possesses this kind of computation power. By allowing some centralization into the architecture, router super peers could provide this functionality. Extra robustness can be provided by allowing peer-based routing whenever a router super peer is not available.

The proposed registry consists of two major building blocks that provide the registry functionality, namely the Distribution Service and the Query Service. This thesis suggests that both services should run as application server services. This is a sound choice for service providing peers, but for simple clients this seems like overkill. Simple clients should be able to use the registry without the need for an application server.

10.5 The Demonstrator

The demonstrator has been developed to show the basic functionality of the proposed registry. This involved providing a basic implementation of the Distribution and the Query Service, as well as providing an environment for demonstrating these services.

The Distribution Service implementation provides the Peer-to-Peer infrastructure needed to distribute service queries on the network. Jxta, which is an XML-based generic Peer-to-Peer middleware technology, is the Peer-to-Peer technology used by the Distribution Service. One of the main reasons for choosing Jxta as the Peer-to-Peer technology of choice was its generic and flexible nature, as well as its grouping feature. The grouping feature makes it possible to limit message distribution to members of a specific Jxta PeerGroup. This feature was central in the design of the query router, which is an integral part of the Distribution Service.

The Distribution and Query Service are designed as Java RMI [69] objects, which are loaded by Java Management Extension (JMX) [58] MBeans into the JBoss application server [55]. This approach follows the suggestions provided in the conceptual architecture of the proposed registry. To demonstrate registry functionality alone, the dependency on JBoss is not really necessary, since standalone RMI object would suffice. This is, however, done to provide an environment, which resembles a production environment as close as possible.

The Query Service relies on a Knowledge base implementation that is based on the Jena RDF framework [52][29]. This framework provides an extensive Java API to RDF, RDF/S and OWL ontologies, with support for database storage and ontology reasoning. In addition, Jena provides a RDQL system that makes it possible to query Jena-based ontology data. Even though RDQL is not designed to query OWL ontologies, the Jena data-model built-in inference engine allows reasoning to be carried out on-the-fly, supporting RDQL queries on OWL ontologies. This is why RDQL is used as the only supported service query language in the Query Service implementation.
The built-in inference engine provides full support for both OWL Lite and OWL DL, but the amount of memory required for querying even modest sized ontologies is extensive. For this reason it was necessary to only support property transitivity as the only OWL rules supported. This is a serious limitation and has grave impact on the expressiveness of service queries. For the demonstration of core functionality, however, transitivity support is enough.

The Distribution Service Query Router analyses outgoing queries for the specific OWL-S Profile ontology sub-class of the service targeted by the query. This operation requires reasoning about ontologies stored in a Routing Knowledge base, which is Jena-based. The same performance problems apply to the Routing Knowledge base as the Query Service Knowledge base. In this case, however, simple transitivity support is enough since the type of queries used is deterministic. In the future, however, the Routing Knowledge base may contain more complex ontologies, and thus full OWL DL support is needed.

The query distribution approach taken by the Distribution Service implementation is rather naïve. Firstly, a query is analyzed to find the associated Jxta PeerGroup. Secondly, the query is distributed within the specific PeerGroup, and finally a response handler is initialized that listens for a possible response for a specified time period.

10.6 Further work

There are several issues raised by this thesis that are subject to further work. Issues related to incompleteness of the technologies used are straightforward; other issues are architectural choices taken that may need rethinking. The following are a few areas that need further work:

- Better support for services with multiple methods in the OWL-S Process ontology need to be provided. The parameter associations between the Profile and Process ontologies need to be rethought. OWL-S has the potential of allowing automatic invocation of service, but for this to happen with less effort from the application designer, this issue needs to be resolved at the OWL-S ontology level, rather than at an application-specific level.

- The OWL-S Grounding ontology provides an association to a WSDL document. The content of this document is outside the domain of an OWL reasoner, except from the grounding associations. This is a problem for applications that need to determine whether a service has a SOAP or EJB service port. Providing an OWL-based WSDL ontology may help eliminate this shortcoming and is something worth working on.

- The proposed architecture relies on a pure Peer-to-Peer based solution without any centralization. In an operational system such an approach may be naïve. Work should be carried out in the area of investigating how a super-peer based architecture may provide satisfactory results. This does also raise the question of whether Jini is a suitable technology for registry distribution after all.

- The registry prototype also needs further work. The current implementation suffers badly from performance issues introduced by the Jena framework. Especially the area of query reasoning performance is a major issue, and alternative solutions should be considered.
• The demonstrator provides an OWL-S invocation framework, which allows transparent invocation of an OWL-S described service process. This framework needs further work, especially in the area of OWL type mapping. In addition, the registry and the OWL-S invocation framework could be integrated into an OWL-S Virtual Machine based solution, similar to [48].
11 Conclusion

This thesis proposes an architecture of a service registry suitable for operation in the future Network-Based Defence (NDB). Centralization avoidance and support for most service types have been key requirements that needed to be addressed by the proposed architecture. I have proposed a solution that uses Peer-to-Peer networking to avoid centralization, as well as describing services using rich semantics in order to preserve transparency.

I started out by evaluating ontology-based languages for describing services in an abstract way. Because of the continuous need for cooperation between networked components in the future NBD, the need for describing services using rich semantics resulted in OWL-S as the technology of choice for describing services. Several OWL-S described example services are provided in order to demonstrate the usability of OWL-S as a service description language.

I continued by evaluating various solutions for distributing a service registry. Peer-to-Peer networks have been considered the most suitable solution for realizing a decentralized service registry. Instead of centralizing service descriptions at a central registry, using Peer-to-Peer networks a service provider can host its own service descriptions. In addition to avoiding centralization, a Peer-to-Peer distributed service registry does not suffer from bad links because of automatic garbage collection.

Finally, a conceptual architecture of a service registry has been proposed that is distributed using Peer-to-Peer networks and where services were described using OWL-S. A prototype of the registry was implemented in order to demonstrate the usability of the registry in a simulated real-life battlespace situation. Using the prototype registry a client is able to find services based on their semantic described capabilities, without querying any centralized resource.

When using OWL-S as a service description language, services can be described using rich semantics. Even though OWL-S provides the Profile ontology for capability matching, the whole service description ontology can be queried by a client. This means that a client may choose to query for services that possess certain process and grounding capabilities. The problem is that at the time of writing this report, the OWL-S Process ontology was not complete, and there were several issues without a solution. The OWL-S Grounding ontology provides a concrete grounding to a WSDL document describing the service. This seems like a good idea, but I consider it a bad one since the content of the WSDL document is not accessible by the OWL reasoner used to query the service description. I propose going the other way around by introducing an OWL-based WSDL ontology.

Using Peer-to-Peer networking in order to avoid centralization is an approach taken by popular file-sharing systems, like Kazaa and Napster. These systems do not provide a pure Peer-to-Peer network where all peers are equal, but rather delegate some extra responsibility to so called super-peers. A similar approach is taken by Jini. The proposed architecture does not rely on any super-peers, and operates in a pure Peer-to-Peer network. I consider this approach sound in a proof-of-concept for the architecture, but the prototype registry demonstrated that such an approach suffers from network performance problems.

As stated, the proposed architecture does provide the requested functionality, but suffers from performance issues and shortcoming of OWL-S. As stated in section 10.6, these are areas that need further work.
12 References


[02]. OASIS, “Universal Description, Discovery and Integration (UDDI)”, http://www.uddi.org/.


[06]. The OWL Services Coalition, “Semantic Markup for Web Services (OWL-S)”, http://www.daml.org/services/owl-s/1.0/


Robotics Institute, Carnegie Mellon University.

Carnegie Mellon University, USA.


[18]. L. Gong, “JXTA: A Network Programming Environment”,

[19]. W. Nejdl et al., “EDUTELLA: A P2P Networking Infrastructure Based on RDF”,

[20]. S. Waterhouse, “JXTA Search: Distributed Search for Distributed Networks”,
Sun Microsystems, Inc.

[21]. ECCMA, "Universal Standard Products and Services Classification (UNSPSC)",
http://eccma.org/unspsc

[22]. F. Yergeau, T. Bray, J. Paoli, C. M. Sperber-McQueen, E. Maler, “Extensible Markup Language (XML) 1.0 (Third Edition)",
W3C Recommendation 4th February 2004

W3C Recommendation, February 1999.

IETF Request For Comment 2396.

[25]. S. Powers: “Practical RDF”,

[26]. SUN Microsystems, Inc. “Java Programming Language”,
http://java.sun.com/

W3C Recommendation, 10 Feb 2004.


[29]. A. Seaborne, “Jena Tutorial: A Programmer’s Introduction to RDQL”,
http://jena.sourceforge.net/tutorial/RDQL/index.html


CORBA Specification V2.1, Chapter 3, August 1997.


W3C Recommendation, 24th June 2003.

UDDI Technical Committee Specification, Dated 20031014.

[36]. N. Srinivasan, M. Paolucci, and K. Sycara, “Adding OWL-S to UDDI, implementation and throughput”,
First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004) 6-9, 2004, San Diego, California, USA.


[38]. Internet Engineering Task Force, “Transport Control Protocol specification”,
http://www.ietf.org/rfc/rfc0793.txt?number=793

[39]. Napster,
http://www.napster.com/

[40]. Webopedia,
http://webopedia.internet.com/

[41]. P. Mockapetris, “Domain names - concepts and facilities”,

[42]. J. Postel, “Internet Protocol (IP)”,
IETF Request for Comment 791, USC/Information Sciences Institute, September 1981.

[43]. ICQ,
http://www.icq.com/

[44]. Kazaa,
http://www.kazaa.com/

[45]. Sun Microsystems, “Jxta 2.0 specification”,
http://spec.jxta.org/nonavi/v1.0/docbook/JXTAProtocols.html

[46]. R. Moats, “URN Syntax”,

[47]. P. Leach, and R. Salz, “UUIDs and GUIDs”,
Work in Progress.

A Peer-to-Peer Distributed Semantic Service Registry


[50]. OWL-S version 1.1, http://www.daml.org/services/owl-s/1.1/

[51]. Time-Entry OWL ontology, http://www.isi.edu/~pan/damltime/time-entry.owl


[67]. Sun Microsystems, Inc., “Java Web Services Developer Pack”,

[68]. Microsoft Corporation, “.NET Framework”,
http://msdn.microsoft.com/netframework/

http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmiTOC.html

[70]. Universal Plug and Play,
http://www.upnp.org/

[71]. Ole Martin Mevassvik, “DAML ontology of Military concepts”. 