Middleware for Semantic Service Advertising and Discovery on MANETs

by

Zef Hemel

A Dissertation submitted to the University of Dublin,

in partial fulfillment of the requirements for the degree of

M.Sc. in Computer Science

2006

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Zef Hemel

August 31, 2006

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Zef Hemel

August 31, 2006

Acknowledgments

I would like to thank my supervisor, Siobhán Clarke, for her support and guidance throughout this project. She gave direction and advice where necessary. I also would like to thank Andronikos Nedos, who was always enthusiastic while helping and developing ideas with me (if his calendar software worked and he actually showed up — just kidding). And I should not forget thanking my NDS group. Thanks guys, it has been an intensive but worthwhile ride this past year. I will never forget it.

I would also like to thank my parents for their continuous support, whatever I do. And last, but certainly not least I would like to thank my wonderful girlfriend Justyna. She supported me a lot, without her the past months would have been a lot less pleasant.

ZEF HEMEL

University of Dublin, Trinity College

August 2006

Middleware for Semantic Service Advertising and Discovery on MANETs

Zef Hemel

University of Dublin, Trinity College, 2006

Supervisor: Siobhán Clarke

MANETs (Mobile Ad-hoc Networks) offer exciting new research opportunities now that devices with wireless capabilities become more widespread. Many wireless technologies, such as 802.11, support these ad-hoc style networks. Opportunities lie in many areas, such as routing protocols, services and applications. The network topology of MANETs is constantly changing and the devices on these networks, like laptops and PDAs, have limited processing and battery power.

Research on low-level protocols that do semantic service discovery on ad-hoc networks is emerging. Pervasive and mobile computing applications require these protocols, but using them requires a lot of engineering and knowledge of network protocols and service matching.

This dissertation describes the design, implementation and evaluation of middleware that makes the task of defining, advertising and discovering semantic services on MANETs more straight-forward by offering APIs to complete these tasks. As part of the semantic service matching, context such as location and workload can be defined and matched to further improve the discovery results. Services and context are described using ontologies. Queries for services can be expressed in a newly developed query language called RaSSQL (RDF and Semantic Service Query Language). The middleware, implemented in Python, is based on ideas from the OntoMobil protocol, but can use any protocol that discovers services based on concept dissemination.

To evaluate the middleware, an application was developed that demonstrates the use of the middleware's APIs to define a set of semantic services and location context. These services are queried using a RaSSQL query defining a desired service profile and service location. A peer flood protocol is implemented as the low-level protocol. Additionally, it is explained how the OntoMobil protocol could be implemented in the middleware.

Contents

List of Figures

Chapter 1

Introduction

This chapter will explain the motivation, objective, contribution and approach of this dissertation. The structure of the rest of the dissertation will then be laid out.

1.1 Motivation

MANETs (Mobile Ad-hoc Networks) offer exciting new research opportunities now that devices with wireless capabilities become more widespread. Many wireless technologies support ad-hoc style networks, such as 802.11 or bluetooth. Many devices have these technologies built in these days, such as PDAs (Personal Digital Assistants), laptops and mobile phones. Ad-hoc networks have applications in pervasive computing [1, 2] and mobile computing [3, 4, 5].

Composing software applications from a set of modular services that can be advertised and discovered on demand is considered a key element in distributed application interoperability [6]. For fixed networks there are already many protocols to do this, such as SLP [7], UPnP [8] and Jini [9]. But also for MANETs these protocols are starting to be built, one is Konark [10].

These protocols all use *syntactic* service matching — they rely on standardised interfaces between services. That works fine in fixed, static networks, but does not work so well in heterogeneous environments like MANETs. Therefore work is emerging on protocols developed to run on MANETs that do *semantic* service discovery. These protocols rely on ontologies to match services based on meaning, rather than their syntactic interface (such as method and parameter names). Examples of

semantic service discovery protocols for MANETs are GSD [11] and OntoMobil [6]. Context-aware service matching is considered to be an important enhancement of semantic service discovery [12]. Context can aid in improving semantic service discovery results by describing the desired context attributes of the service (or its node) like location, workload and system capabilities [13, 14].

Applications using these novel protocols for semantic service discovery are not widely being built yet. I believe that the main reason for this is that it simply is too hard to do so. A lot of knowledge is needed about networking protocols, ontology matching and semantic service matching algorithms. The protocols are difficult to implement and it is hard to formulate a semantic service discovery query in an expressive way.

1.2 Objectives

The objective of this dissertation project is to develop middleware that aids developers in developing applications that do context-aware, semantic service matching on MANETs. The middleware aids the developer in developing these kinds applications by offering APIs that are straight-forward to use and take a lot of the complexity of developing such applications away. The middleware has a big focus on accommodating the needs of OntoMobil [6], a semantic service discovery protocol for MANETs developed at Trinity College. But any protocol that discovers services through concept dissemination (and some other requirements, see section 4.2.1) should work with the middleware.

1.3 Contribution

The contribution of this dissertation is middleware that fills the gap between the low-level semantic service discovery protocols that are being developed for MANETs and the applications that want to use them.

The middleware takes a lot of the complexity out of the development of applications that run on MANETs and have to discover semantic services dynamically. It does so by exposing APIs that make this task more straight-forward. A query language is developed, called RaSSQL (RDF and Semantic Service Query Language), that allows developers to discover services on an MANET by defining an ideal profile for such a service and putting constraints on the service node's ontology. The latter can

be used to do context matching.

The middleware is heavily based on the ideas from OntoMobil [6]. At the time of the project OntoMobil was not yet stable, therefore for the evaluation a simple flood protocol was implemented to demonstrate that the middleware works. It is then described how OntoMobil can be integrated and used by the middleware.

The middleware will then be evaluated by developing an application. The example (a parking place finder) will demonstrate how services can be defined, exposed and discovered. It will demonstrate how location context is used to make the service matching more accurate, by only returning parking services in the requester's vicinity.

I believe that the availability of middleware for context-aware, semantic service discovery on MANETs is an important step in encouraging developers to start developing applications for MANETs. Currently, not many applications are developed that take advantage of semantic service discovery.

1.4 Approach

The approach taken to this project is as follows:

1. Study of the state of the art. The areas that the middleware will cover are studied and described.
2. A design for the system will be made.
3. A developer API will be defined.
4. The middleware will be implemented according to the design.
5. The middleware will be evaluated by implementing a simple flood protocol and building an example application that demonstrates the middleware's features.

1.5 Structure

The structure of this dissertation is as follows: in chapter 1 the problem is explained and the contribution of this work presented. Chapter 2 provides background information on the areas that will be covered in the rest of the dissertation:

- MANETs. What are they, what are they used for, what are the challenges.
- Services. Why are they important, service advertisement and service discovery.
- Semantic service discovery. Why is semantic service discovery significant, what is ontology, what is RDF, what is SPARQL.
- Context-aware service matching. Why is context important, what is context used for, how can context aid in service discovery.
- Middleware. What is it.

After the background, chapter 3 will describe the current state of the art, in particular the following areas:

- Semantic service advertising. The latest technologies to define semantic services are discussed.
- Semantic service matching. What algorithms are there.
- Semantic service discovery on MANETs. Two approaches will be looked at (GSD and OntoMobil).
- Semantic middleware. The Edutella framework for distributed querying of RDF meta-data will be explored.
- Context-aware service matching. It will be shown how context can aid in service discovery and how context can be represented.

Chapter 4 will lay out the design of the middleware. The design choices will be discussed, followed by a global architecture. The approach to service matching will be explained. In chapter 5 the resulting developer API will be described. Chapter 6 will describe some of the implementation details. Chapter 7 will give an overview of the security aspects of the project. In chapter 8 the project is then evaluated. It is shown how the peer flood protocol works, how the OntoMobil could be integrated into the middleware and a simple application is built on top of the middleware. The dissertation ends with chapter 9 listing the project's conclusions and future work.

Chapter 2

Background

This chapter will give some background information on the technologies and concepts being used throughout this dissertation. First MANETs will be discussed, followed by services, semantic service matching, context-aware service matching and middleware.

2.1 MANETs

More and more devices become mobile and have wireless capabilities. Examples of these are laptops, but also PDAs and mobile phones. These devices can form on-the-fly networks, also called ad-hoc networks. These are networks where devices connect directly to each other forming a mesh-like network. This is different than infrastructure wireless networks where the devices connect to an access point that functions as a hub.

The idea of forming an on-the-fly ad-hoc network of mobile nodes dates back to DARPA packet radio network days [15, 16]. More recently the interest in the subject has grown due to availability of license-free, wireless communication devices that users of laptop computers can use to communicate with each other. [17]

A special kind on ad-hoc network is a MANET. MANET stands for Mobile Ad-hoc Network, an ad-hoc network composed of mobile and wireless nodes, such as PDAs, mobile phones and laptops.

MANETs have the following properties:

- Constantly changing topology. Nodes constantly come into the network and leave it, this can

be a matter of hours, minutes or even seconds.

- Low-power devices. As MANETs are wireless and running on low-power devices with little memory and few CPU cycles at their disposal the software should be optimized for this.
- Minimum configuration. So they allow for quick deployment.
- Decentralised. This follows from the fact that MANETs are constantly changing. As applications cannot rely on central servers being present at all times, centralization does not work. A peer-to-peer approach has to be taken instead.

2.1.1 Applications

Ad-hoc networks can be used in situations and locations where no network infrastructure is available or cost-effective. Areas may include [18]:

- Business. Such as meetings outside the office with clients. Or colleagues wanting to exchange files in the train or car.
- Crisis management. If a disaster has taken place and all the infrastructure is gone, MANETs can be quickly deployed, to allow for wireless communication.
- Automotive telematics. Cars on the road can exchange information. As automitive telematics is a growing industry there is a lot of opportunity for MANETs in this area.
- Third-world countries. Recently the $100 laptop, developed by MIT, became a big news item. The idea behind this project is to produce affordable laptop computers for educational purposes in third-world countries. The laptops will be equipped with 802.11 wireless cards, but as infrastructure is lacking in such countries, the laptops by default are set up to form a mesh ad-hoc network with other laptops in the area.
- Pervasive computing. Ad-hoc networks are also an essential part of pervasive computing [1]. Pervasive computing, also known as ubiquitous computing, is about integrating little computers in the environment, like clothes, fridges and items in your shopping baskets. These little computers communicate wirelessly.

2.1.2 Challenges

The main challenges in MANETs are as follows:

1. Constantly changing network topology. Nodes in the network may be moving. Nodes may also be constantly joining and leaving the network. Therefore no assumptions can be made about availability of nodes. Because of this reason it is not possible to use the client-server model, the network has to be decentralised.

2. Scalability. Because of decentralisation, scalability becomes a challenge. Routing in particular is a challenge and so are optimal network topologies for application-specific overlay networks. These challenges are very similar to the ones on peer-to-peer networks.

3. Low-power consumption. As MANETs will largely run on top of battery-driven devices such as laptops and PDAs, power consumption has to be taken into account.

4. Reachability, every node should be reachable from any other node in the network.

5. High packet loss. A lot of packets may be lost on the network, this has to be taken into account when protocols such as UDP are used.

2.2 Services

Services in computer science, and in particular the Service-Oriented Architecture (SOA), are seen as the next big step in computer science. Object-Oriented Programming allowed for reuse of pieces of software, SOA takes this up an abstraction level to allow reuse of services independent of their underlying platform. The current "hot" way to expose and consume services are web services. Web services are services consumed over the web, encoded in some XML format and using HTTP for transport. Web services allow one to easily use services that have been built and deployed already, be it by yourself, other departments in the same company or even companies the user has never heard of.

In many enterprises web service technologies are already being used. There is a standardized way to define the interface of a service and a standardized way to invoke them. The most commonly used formats for this are WSDL and SOAP, but other technologies could be used, such as XML-RPC [19] or RESTful web services.

2.2.1 Advertisement

The interface of a web service is described in an XML format called Web Service Description Language (WSDL). It defines the name and description of a service and for all operations that can be invoked their names, inputs and outputs. WSDL interface descriptions can be translated to interfaces of many programming languages, such as C# and Java.

Below a snippet of some essential parts of a WSDL interface definition is shown to give an idea of what such interface descriptions look like. It defines a stock price service. First input and outputs are defined (as messages), then these are grouped as an operation inside a `portType`. The operation is then bound to a URL which is defined to be invokable through SOAP (Simple Object Access Protocol).

```
<?xml version="1.0"?>
<definitions name="StockQuote"
       targetNamespace="http://example.com/stockquote.wsdl"
       xmlns:tns="http://example.com/stockquote.wsdl"
       xmlns:xsd1="http://example.com/stockquote.xsd"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns="http://schemas.xmlsoap.org/wsdl/">

 <message name="GetLastTradePriceInput">
   <part name="body" element="xsd1:TradePriceRequest"/>
 </message>

 <message name="GetLastTradePriceOutput">
   <part name="body" element="xsd1:TradePrice"/>
 </message>

 <portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
   </operation>
```

```
 </portType>

 <binding name="StockQuoteSoapBinding"
    type="tns:StockQuotePortType">
   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="GetLastTradePrice">
      <soap:operation
         soapAction="http://example.com/GetLastTradePrice"/>
      <input>
          <soap:body use="literal"/>
      </input>
      <output>
          <soap:body use="literal"/>
      </output>
   </operation>
 </binding>

 <service name="StockQuoteService">
   <documentation>My first service</documentation>
   <port name="StockQuotePort" binding="tns:StockQuoteBinding">
      <soap:address location="http://example.com/stockquote"/>
   </port>
 </service>
</definitions>
```

2.2.2 Discovery

Often applications do not want to depend on one single provider of a service. This can be because of reliability reasons, but also because an application simply does not know the service it will use yet. If a printer must be used, for example, the available printer may differ depending on where the application is deployed or even the context of the device the application is running on. One can

imagine, for example, that a user needs to print a page on a nearby printer. Which printer is being picked depends on the user's location, but can also depend on other factors such as how busy a printer is. The process of finding services that satisfy an application's need is called service discovery.

To allow web services to be discovered, UDDI [20] is often used. UDDI stands for Universal Description, Discovery and Integration. It consists of three parts:

- White pages, which list business names, addresses and contact information.
- Yellow pages, which list categories based on standard taxonomies.
- Green pages, which contain technical specifications and references.

Essentially UDDI is a directory of web services. Services are categorized according to a taxonomy. There is a tModel (technical model) associated with each service. The tModel contains data like the service's interface and description. It can be used to match services.

In general, the service discovery process works as follows: first a description of the required service is formulated. Then, services are discovered. These services are matching against the desired service description according to some algorithm. The matching services are then returned. One or more of them are selected and invoked.

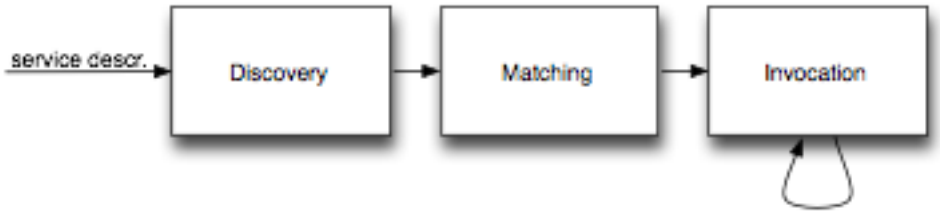

Figure 2.1: The three steps of service discovery

The main challenges in service discovery on MANETs in particular are the following:

1. Optimal network topology. The overlay network over which the different service advertisements and queries are distributed should be optimal.
2. Least possible overhead. As few messages as possible should be sent over the network to save bandwidth and with that, energy (and CPU cycles).

3. Scalability. The solution should scale. It should work with just a few nodes, but also with hundreds, as MANETs may become quite big.

4. Low response time. Response times should be as low as possible to keep applications responsive, a user does not want to wait for an application discovering a service for minutes long.

5. Reachability, all the services exposed on nodes should be discoverable and accessible.

2.3 Semantic service matching

Clients cannot depend on a particular service always being available, on MANETs this is particularly the case. Therefore services have to be discovered on the fly. Chakraborty et al. in [11] observe that MANETs are heterogeneous environments. Services can be developed by different people and different vendors. They may have different ways of describing their service. If a service directory would be used they could pick different keywords and different method and parameter names even though the services do exactly the same thing.

The problem here is the lack of semantics. In systems like UDDI, services are matched syntactically, not semantically, i.e. based on service name and method names rather than their meaning. An emerging research area currently is semantic service matching. Services are described in terms of concepts in an ontology. As the ontologies are independently developed it is not likely that they are exactly the same, however their meaning may be equal. Ontology matching algorithms can figure out whether two concepts have the same meaning. For example a service associated with the *House* concept may be requested and one particular service may be described with a *Building* concept. At first sight these concepts do not seem to match, but as their properties and relationships are analysed it may turn out they actually represent the same thing, or that *House* actually is one particular type of a *Building* (a sub-class in OOP terms).

2.3.1 RDF and ontology

The classic definition of an ontology is "a specification of a conceptualization" [21]. But Gruber also gave a more elaborate definition [21]:

An ontology is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For AI systems, what "exists" is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-based program represents knowledge. Thus, in the context of AI, we can describe the ontology of a program by defining a set of representational terms. In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these terms. Formally, an ontology is the statement of a logical theory.

Semantic web

The most famous application of ontology in computer science at the moment is the semantic web. The idea of the semantic web is to build a web-spanning ontology that adds semantic information to resources (URIs). Inventor Tim Berners-Lee describes the the semantic web in [22] as "not a separate web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

For this to be realised standards were developed. Most notably RDF/RDFS and OWL. RDF allows to define resources and relationships between them. In fact RDF is just an XML serialization of `(subject, predicate, object)` statements.

- `subject` is the resource the statement is about.
- `predicate` is the verb, the predicate.
- `object` is the value, which can be a simple type (string, integer) or an URI to another resource

An example of a triple is:

```
http://uri.org#John http://uri.org#hasDad http://uri.org#Shane
```

Once a number of statements like this are defined, it becomes possible to reason about them and infer new facts from them that were not necessarily obvious straight away. For example if there is also this statement:

```
http://uri.org#Jane http://uri.org#hasDad http://uri.org#Shane
```

One could infer that John and Jane are siblings. This is where the power of the semantic web lies. An example piece of RDF that describes the previously mentioned two facts:

```
<rdf:Description rdf:about="http://uri.org#John">
  <f:hasDad>http://uri.org#Shane</cd:artist>
</rdf:Description>
<rdf:Description rdf:about="http://uri.org#Jane">
  <f:hasDad>http://uri.org#Shane</cd:artist>
</rdf:Description>
```

RDFS extends RDF and adds the ability to define types and properties that can be instantiated. It essentially adds type information to RDF.

OWL is an extension of RDF and the successor of DAML, the DARPA Markup Language. OWL in [23] is described as follows:

> The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics. OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full.

SPARQL

SPARQL is a language to query RDF data. It is a W3C recommendation [24]. It has an SQL-like syntax. In [25] the SPARQL RDF protocol is described, a technique to perform SPARQL queries remotely over a XML web service.

An example SPARQL query is:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
PREFIX  ns:  <http://example.org/ns#>
SELECT  ?title ?price
WHERE   { ?x ns:price ?price .
          FILTER (?price < 30) .
          ?x dc:title ?title . }
```

This query retrieves the title and price of items (products) that have a price lower than 30.

Syntax A SPARQL query consists of zero or more `PREFIX` definitions, a `SELECT` and `WHERE`-clause and optionally `ORDER BY`, `LIMIT` and `OFFSET`.

The `PREFIX` definitions allow one to define namespace prefixes, this is for convenience only. Instead of writing `http://purl.org/dc/elements/1.1/title` it is then possible to write `dc:title`.

In the `SELECT`-clause a number of variables can be specified whose values the user is interested in. Variables all start with a question mark: `?`. In the `WHERE` clause the restrictions on the values of the variables defined in the `SELECT` are defined. New variables can be used in this clause.

`ORDER BY`, `LIMIT` and `OFFSET` can be used to order the result set in a define way and to limit the number of results returned. `OFFSET` can then be used to page the result set.

2.4 Context-aware service matching

Context is defined by Dey in [26] as information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to interaction between a user and an application including the user and application themselves. Context-awareness is defined as: a property of a system that uses context to provide relevant information and/or service to the user, where relevancy depends on the user's task.

Context can be categorised into three categories [12]:

1. Computing context: contextual information related to computational aspects of the context-aware system.

   (a) Application context: email received, websites visited, etc.

(b) System context: network traffic, status of resources, bandwidth, quality of service, etc.

2. User context: context information related to the service requester.

   (a) Personal context: health, mood, schedule, activity, etc.,

   (b) Social contexts: group activity, social relationship, people nearby etc.

3. Physical context: contextual information related to physical aspects of the context-aware system.

   (a) Physical context: location, time, etc.

   (b) Environmental context: weather, altitude, light, etc.

   (c) Informational context: stock quotes, sport scores, etc.

2.4.1 Context-aware applications

There are already quite some useful context-aware applications around, although they are currently usually only deployed in research environments. Some applications are as follows [27]:

1. Employee locations. Employees wear a badge that are traced by the system. On a map can be seen where a particular employee is, which allows them to be found quickly.

2. Call forwarding. The employee wears a badge that is traceable across locations by the system. The calls the user gets are forwarded to the phone nearest to his or her location.

3. Shopping assistant. The system can provide information on items the customer is browsing or aid in locating items the customer is looking for.

4. Cyberguide. A travel guide that provides information on sights that can be seen on the place where the tourist currently is.

5. Conference assistant. The system provides information on the presentations going on when the user enters a conference room.

2.4.2 Context for service matching

Context can also help to make the service discovery process more accurate.

In [13] Strang et al. describe a scenario in which context-aware service discovery proves useful. The scenario describes a tourist visiting a foreign city. He takes pictures there and wants to print some of them to show to his family. Instead of looking on the web for a place where he can make prints of the pictures he wants to find the optimal photo printing service. How optimal a given service is may depend on many contextual factors, such as the distance between the tourist and the service, the opening hours of the printing service, the price of printing and current workload of the printing service.

Not all of these contextual factors may be known, for example the tourist may not know where he is exactly, but a context provider may be able to give him this information (for example obtained through GPS). When the optimal service has been found, another context-aware service (*MapService* may be used to show the route from the tourist's current location to the printing service.

This scenario shows that in service discovery context could relate to two things:

1. The context of the service consumer, the node that discovers services, in this case the tourist.
2. The context of the service that is being discovered, the photo printing service.

Doulkeridis [14] notes the following regarding services and context:

> Mobile users usually tend to prefer using mobile services a) from nearby locations, b) that return fresh data, c) provided by trusted users, and d) returning results displayable in the requesting device.
>
> In other words, several parameters such as location, time, user identity and profile, device capabilities, that involve both the requesting user as well as the service provider, have great influence in the search for the most suitable service. This fact highlights the need for context management to support the efficient deployment of applications that strongly depend on mobility of users and data sources. Context plays the role of a filtering mechanism, allowing only transmission of relevant data and services back to the device, thus saving bandwidth and reducing processing costs.

2.5 Middleware

Middleware is software that connects two otherwise separate applications or components. It is often used in distributed systems to abstract away from a protocol's implementation details. It functions as a layer in between the protocol and the application, as shown in figure 2.2

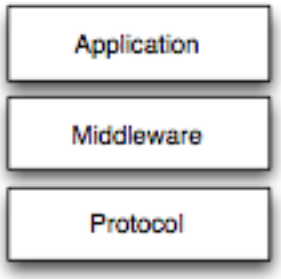

Figure 2.2: Application, middleware and protocol layers in a distributed system

Examples of middleware are the web service APIs as provided by many programming platforms (for example Java and .NET). These APIs offer a simple way to annotate a service implementation, the underlying protocol and serializations are then handled by the web service middleware, the programmer does not have to worry about that anymore.

Generally middleware is developed to simplify the task of developing a certain kind of application by offering a set of APIs that the developer can use.

Chapter 3

State of the art

This chapter will give an overview of the state of the art of five relevant research areas: semantic service advertising, semantic service matching, semantic service discovery on MANETs, semantic middleware and context-aware service matching.

3.1 Semantic service advertising

In order to advertise a service, whether in real-life or in computers, an advertisement has to be written. For semantically described services these are in practice semantic service descriptions, often written in some XML format. This section will briefly explore two technologies to describe semantic services: OWL-S and WSDL-S. After that it will comment on some specifics when it comes to semantic service advertisements on MANETs.

3.1.1 OWL-S

OWL-S (Web Ontology Language for Services) [28] defines a semantic service description separate from the implementation of the service itself. The semantics are described in OWL-S and the actual callable interface is described in WSDL [29]. An OWL-S description of a semantic web service consists of three parts:

1. A Semantic Profile, in which the following is defined:

(a) Parameters, which are service parameters which have a name and a value which link into the ontology.

(b) Inputs, which each refer to some concept within an ontology.

(c) Outputs, which each refer to some concept within an ontology.

(d) Preconditions, which define some preconditions that have to hold for the service to be able to work properly.

(e) Effects, which define the effects of execution the service (also known as post-conditions).

2. A Process Description, which defines how the service operates. A process is an entity that transforms input values into output values. There are three types of processes:

   (a) Atomic, these are directly invokable.

   (b) Simple, these are not directly invokable. They are more like interfaces. They have to be implemented or extended.

   (c) Composite, these are composites of multiple atomic processes and can contain control structures.

3. A Grounding, which defines how the semantic profile matches on the actual service interface (often defined in WSDL).

A semantic profile in OWL-S looks as follows:

```
<profileHierarchy:AirlineTicketing rdf:ID="ReservationProfile">
  <service:presentedBy rdf:resource="&ba_service;#Reservation"/>
  <profile:has_process rdf:resource="&ba_process;#Process"/>
  <profile:serviceName>ReservationAgent</profile:serviceName>
  <profile:textDescription>...</profile:textDescription>
  <!-- ... -->
  <profile:serviceCategory>
    <addParam:NAICS rdf:ID="NAICS-category">
      <profile:value>Airline reservation services</profile:value>
      <profile:code>561599</profile:code>
    </addParam:NAICS>
  </profile:serviceCategory>
  <!-- ... -->
  <profile:hasInput rdf:resource="&ba_process;#OutboundDate_In"/>
  <profile:hasInput rdf:resource="&ba_process;#ReservationID_In"/>
  <profile:hasOutput rdf:resource="&ba_process;#AcctName_Out"/>
  <profile:hasOutput rdf:resource="&ba_process;#ReservationID_Out"/>
  <profile:hasEffect rdf:resource="&ba_process;#HaveSeat"/>
</profileHierarchy:AirlineTicketing>
```

3.1.2 WSDL-S

WSDL-S [30] is an extension of WSDL (as described in section 2.2.1) and adds semantic information to it. It allows to link concepts from ontologies into the WSDL interface descriptions. This can be done in any of the following:

- Data structure definitions (in XML Schema), for example using the `wssem:modelReference` attribute, like in this example:

  ```
  <xs:element name="processPurchOrderResp" type="xs:string"
      wssem:modelReference="POOntology#OrderConfirmation"/>
  ```

  This can also be used to annotate input and output parameters.

- Interface definitions, like with the `wssem:precondition`, `wssem:effect` elements:

```
<interface name="PurchaseOrder">
   <operation name="processPurchaseOrder" pattern=wsdl:in-out>
   <input messageLabel="processPurchaseOrderRequest"
      element="tns:processPurchaseOrderRequest"/>
   <output messageLabel ="processPurchaseOrderResponse"
      element="processPurchaseOrderResponse"/>
   <wssem:precondition name="ExistingAcctPrecond"
      wssem:modelReference="POOntology#AccountExists"/>
   <wssem:effect name="ItemReservedEffect"
      wssem:modelReference="POOntology#ItemReserved"/>
   </operation>
</interface>
```

WSDL-S is part of the bigger METEOR-S [31] project which aims at creating a framework for configuring and executing dynamic Web processes.

3.1.3 MANETs

Services in MANETs are generally much more light-weight than services in enterprise applications. Technologies like OWL-S and WSDL-S allow to define very complex service descriptions and to perform very complex matching. Semantic service interfaces in MANETs are generally defined in a much simpler and light-weight way as will be shown in section 3.3.

Semantic interfaces in MANETs generally consist of a set of concepts [6]. These concepts are associated with inputs and outputs. There usually is also a concept to define the service category or group as Chakraborty calls it in [11].

3.2 Semantic service matching algorithms

Paolucci et al. in [32] describe an algorithm that can match DAML-S service profiles. It identifies the problem of defining what a *sufficient match* is and that there should be means for the requester to set the degree of similarity. The degrees of matching are defined as exact, plug in, subsumes and fail.

Tang also describes the process of matching DAML-S services in his master's thesis [33]. He describes a full procedure of matching service categorization, matching of inputs and outputs and determining if a service satisfies the requirements. He also gives the implementation details of doing this. The algorithm is very similar to [32].

Both algorithms are based on the idea of for each input, output and category concept in the requested profile, finding a concept in the service profile that is being matched against, that is as similar as possible. Depending on if it is a input, output or category concept it may be acceptable for concepts to be semantically equal or to subsume each other.

3.3 Semantic service discovery on MANETs

The simplest approach to service discovery on a MANET is to simply broadcast a query to all nodes on a MANET, but obviously this approach generates a lot of traffic on the network and does not scale well, this is the pull approach to service discovery.

The other approach is the push approach. Services advertise themselves on the network. Periodically they broadcast advertisements which may be cached by interested nodes. Service queries are then performed on the caches of the nodes. However, as network become bigger, larger caches are needed and memory is scarce on a device that is typical to a MANET. This system does not scale well either. Therefore other approaches have been developed to more efficiently discover services on MANETs. The two protocols that are discussed in this section use the semantic information in the service profiles to more efficiently discover services.

3.3.1 GSD

The GSD protocol described in [11] utilizes the semantic capabilities of DAML (the predecessor of OWL) to describe services and resources present on nodes in a MANET. Service requests are also expressed in DAML and are matched using a service-matcher module.

Services are classified into a hierarchy of groups starting from the root group called *Service*. A node advertises its service to its neighbours within n hops. Advertisements contain a number of groups that the sending node has in its vicinity, this information can be used by other nodes to more

intelligently route service discovery requests to areas in the network that contain services in the group that is being requested.

GSD extends the DReggie [34] service description ontology. It contains capabilities, inputs, outputs, platform constraints, device capabilities etc. A preliminary service group hierarchy has been defined using DAML's `Class` and `subClassOf` axioms.

All nodes that contain services send advertisements to all the nodes in their vicinity at a set interval. This interval depends on how rapidly the network changes, the faster it changes the shorter the interval. An advertisement can contain a hops-to-live counter, if it is set the nodes that receive the advertisement will lower the counter and forward it to their neighbours.

Service discovery is initiated from the application layer of one of the nodes in the network. The first thing that will happen is that it will check for any matching services in its local service cache. If no matching service is found in its cache, the request is selectively sent to nodes in the source node's vicinity that have seen services in the requested group before.

3.3.2 OntoMobil

GSD assumes a common domain model [11] for its group hierarchy. The protocol called OntoMobil described by Nedos et al. in [6], however, is based on the idea that the service ontologies are independently developed and therefore do not have to be agreed upon with other service nodes. As MANETs are decentralised, it only seems logical to also decentralise the development of their ontologies. Therefore a node will have its own local ontology with the concepts that it needs to describe its local services.

A shared understanding of ontologies is still required to do meaningful interpretation. Because ontologies are now independently developed they have to be matched. Matching ontologies is a resource intensive task as is shown in [6]. Therefore the protocol takes the approach of progressive matching. The process to discover services is as follows:

1. a service request is formulated as a concept-based query,
2. the query is sent to a random subset of nodes in the network,
3. the matching concepts provide a list of candidate nodes that host compatible ontologies,

4. the query can now be redirected to the candidate nodes where semantic match making can take place between the required and advertised services.

Concepts that are used for service matching are gossiped in a random fashion. Every node has three buffers associated with it:

1. *Node View*, which contains addresses (or identifiers) of a subset of the nodes in the network.

2. *Ontology View*, which contains the node's local ontology.

3. *Concept View*, which contains a set of concepts from other nodes. This set is randomised because it only contains a set of the concepts of other nodes and is constantly changing. It is also evolving because the gossip protocol constantly inserts and removes concepts from this view while the matching algorithm in each node establishes associations between received and stored concepts.

The representation of concepts in the *Concept View* is an extension of the one used in the *Ontology View*, it is referred to as the *network representation*. Essentially it adds a reference to the origin node of the concept.

A more detailed description of how the protocol operates is given in section 8.2 where it is described how this protocol could work with the middleware that this dissertation describes.

3.4 Semantic middleware

It is also interesting to look at middleware that is already out there that does some sort of semantic matching. Edutella [35] is a semantic RDF meta-data infrastructure built on top of Sun's JXTA [36]. The JXTA website [36] describes JXTA as follows:

> JXTA technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner. JXTA peers create a virtual network where any peer can interact with other peers and resources directly even when some of the peers and resources are behind firewalls and NATs or are on different network transports.

Written in Java, JXTA provides an extensive set of APIs to easily build peer-to-peer systems. The JXTA architecture is shown in figure 3.1.

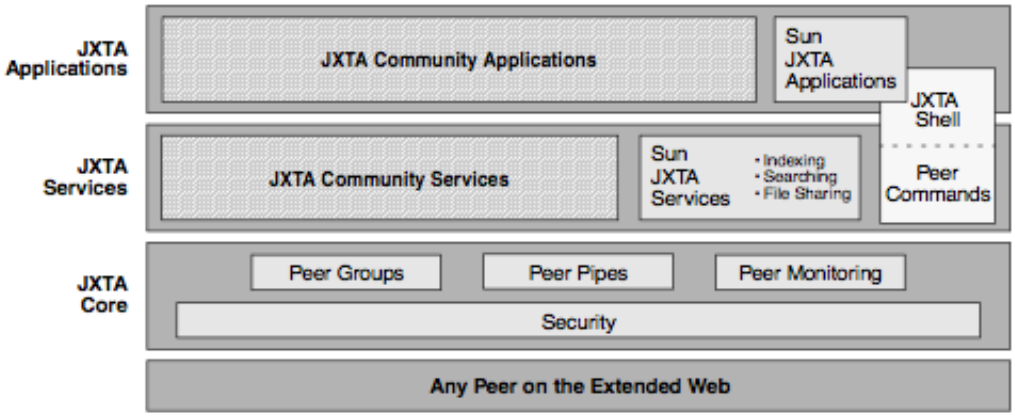

Figure 3.1: The JXTA architecture [37]

3.4.1 Edutella

The most famous peer-to-peer applications deployed today, like Gnutella, Kazaa and eMule allow users to query the peer-to-peer network for certain types of files, such as music files, videos, applications and documents. The applications extract meta-data like artist, song title and duration from the files themselves. The kind of meta-data that can be queried is limited, however, and cannot be easily extended. A complex query like "give me all course materials written in Oxford between 1995 and now about bio-chemistry" certainly cannot be supported by these current systems. Therefore Nejdl et al. in [35] describe and implement Edutella, a meta-data infrastructure utilising RDF built on top of the JXTA [36] peer-to-peer framework. A framework that essentially allows peers to query the network for arbitrary RDF resources.

Edutella provides the following five services:

1. *Query Service*: a standardized way to query and retrieve RDF meta-data.

2. *Replication Service*: provides data persistence/availability and workload balancing while maintaining data integrity and consistency.

3. *Mapping Service*: translates between different meta-data vocabularies to enable interoperability between different peers.

4. *Mediation Service*: defines views that join data from different meta-data sources and reconcile conflicting and overlapping information.

5. *Annotation Service*: to annotate materials stored anywhere within the Edutella Network.

Especially interesting are the mapping and mediation service. Because peer-to-peer networks (like MANETs) are heterogeneous environments one cannot assume that the vocabularies and schemas used to describe resources match. The mapping and mediation translate and merge information from different sources to overcome this problem. The query service is also interesting.

Query Internally Edutella represents queries and their results in a datalog based model. Nejdl et al. in [35] describe it as follows:

> [Datalog is] a non-procedural query language based on Horn clauses without function symbols. A Horn clause is a disjunction of literals where there is at most one positive (non-negated) literal. A Datalog program can be expressed as a set of rules/implications (where each rule consists of one positive literal in the consequent of the rule (the head), and one or more negative literals in the an- tecedent of the rule (the body)), a set of facts (single positive literals) and the actual query literals (a rule without head, i.e., one or more negative literals).

The query "Return all resources that are a book having the title 'Artificial Intelligence' or that are an AI book" can be expressed as follows in Datalog:

```
aibook(X) :- title(X, 'Artificial Intelligence'),
             type(X, Book).
aibook(X) :- type(X, AI-Book).
?- aibook(X).
```

The language being used to exchange queries is actually a set of five languages with increasing expressiveness. The simplest one is called RDF-QEL-1, the most advanced one RDF-QEL-5. Each

language describes the kinds of queries a peer can handle (conjunctive queries, relational algebra, transitive closure, etc.)

The query previously expressed in datalog can be expressed as follows in RDF-QEL-1:

```
<edu:QEL1Query rdf:ID="AI_Query_1">
   <edu:hasVariable rdf:resource="#X"/>
</edu:QEL1Query>
<edu:Variable rdf:ID="X" rdfs:label="X">
   <rdf:type rdf:resource="http://www.lit.edu/types#AIBook"/>
</edu:Variable>
<edu:QEL1Query rdf:ID="AI_Query_2">
   <edu:hasVariable rdf:resource="#Y"/>
</edu:QEL1Query>
<edu:Variable rdf:ID="Y" rdfs:label="X">
   <rdf:type rdf:resource="http://www.lit.edu/types#Book"/>
   <dc:title>Artificial Intelligence</dc:title>
</edu:Variable>
```

Mapping When a peer joins a network, it will join a network hub registering itself with a set of schemas it uses and the maximum level of RDF-QEL-i language it can support. This information can then be used to translate queries and results to the (schema) type the peer understands.

3.5 Context-aware service matching

This section will describe the way context is used in service discovery and two different ways to represent context: attributes and ontologies.

3.5.1 Use of context

Broens notes in [12] that context in service matching can be used for service request completion, namely

- Request completion. Contextual information could be added to the service request to enable

matching based on the service requester. This could be used when a service requester requests a service nearby, to be able to determine closeness the requester's location is sent with the request.

- Input completion. When certain services require inputs that are not provided by the service requester, these may be obtained from the service requester's context.

3.5.2 Context representation

Context can be represented in different ways. This section will describe two of them.

Attributes The simplest one is using *(name, value)* pairs. Service discovery technologies like [7] use this simple attribute-based system. Example [38]:

```
(printer-name=Hugo),
(printer-natural-language-configured=en-us),
(printer-location=In my home office),
(printer-color-supported=false)
```

This approach is fairly limited and purely syntactic, the meaning of these attributes might be clear to the user (for example from documentation), but are simply strings to a computer. What would be better (especially in heterogeneous environments like MANETs) is a semantic description of the context.

Ontology In [39] Strang et al. describe the reasons to use an ontology for context as follows:

> It is highly desirable, that each participating party in a service interaction shares the same interpretation of the data exchanged and the meaning behind it (so called shared understanding). This is done in our approach by the use of ontologies. Ontologies seem to be well suited to store the knowledge concerning context.

The language proposed in [39] is called CoOL: Context Ontology Language. It consists of a core, which is defined in DAML, OWL and also in F-Logic, and a integration part which allows it to be used in several service frameworks. A *Aspect-Scale-Context* model is defined. An *aspect* consists of

a number of related *scales*. The spatial distance aspect, for example, may consists of a meter scale and a kilometer scale. Context has associated quality attributes.

In [40] Wang et al. propose an OWL-encoded context ontology called CONON for modeling context in pervasive computing environments and for supporting logic-based context reasoning. Wang et al. realise it is not possible to create a ontology for all possible contextual information one ever would want to describe. Instead they observed that location, user, activity and computational entity were fundamental context concepts. CONON specifies an upper-ontology as shown in figure 3.2 that defines those basic contextual concepts and some subclasses of them. Domain-specific context can then be defined by sub-classing concepts in this ontology.

Additionally, reasoning can be done about context. First-order predicates in the form of `(subject, predicate, object)` are used to describe context, for example `(Zef, locatedIn, Bedroom)`. If another fact is known, for example `(Bedroom, locatedIn, House)`, a transitivity rule could be formulated which allows the fact that `(Zef, locatedIn, House)` to be inferred automatically.

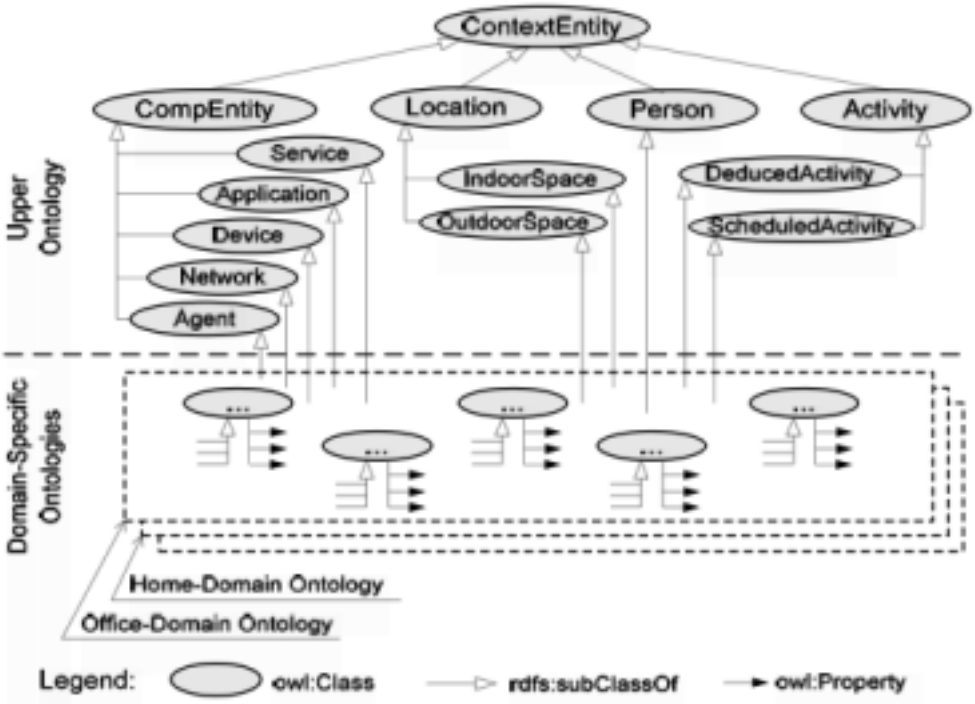

Figure 3.2: The context ontology described in [40]

3.5.3 Context reasoning

CONON [40] allows one to reason about context using rules. Forstadius et al. in [41] propose a rule-based system for context reasoning based on RDF. Context information can be defined in an RDF ontology, reasoning about this context then takes place using reasoning rules. In this system rules like "if there is a Projector in the same room with the user, recommend it." can be formulated as RDF statements.

Rules are used to infer new facts from current ones defined in an RDF ontology. They are in the form *if A and B and C and ... then Z*. For example *if X locatedIn Y and Y locatedIn Z then X locatedIn Z*. [41] uses these rules to do recommendations for services. The rules can be stored in an RDF ontology. A rule is represented as an instance of the `Rule` concept. A rule consists of a `condition` and a `action`. Both condition and action can be a set of RDF triples which contain variable resources. These resources represent variables like the X, Y and Z in the example.

3.5.4 Context on MANETs

One property of MANETs is that they are heterogeneous environments. Applications are independently developed and one would assume, so are their context representations. However, it appears in the literature such as [42, 43] that current context-aware related technologies depend on a predefined and agreed upon context ontology. This is a shame, but I am sure that people are working on ways to do context matching, like they do with service matching.

Chapter 4

Design

This chapter describes the design of the middleware. It describes design choices made and the architecture.

4.1 Choices

This section lists and elaborates on a number of key design decisions made during the design process.

4.1.1 Decentralised and protocol

Because the middleware has to operate on a MANET, the middleware should not use a client-server model, instead a peer-to-peer model shall be used. It is decided that the focus of this project is not on protocols but rather on middleware and developer APIs. The protocol that the middleware should work with (OntoMobil [6]) has not been finalized yet. To be able to support it in the future the protocol will be abstracted and all protocol related operations will be put in a separate class. To demonstrate the middleware a simple peer flood protocol will be implemented. It will be described how OntoMobil can be integrated into the middleware later.

4.1.2 Python

Python [44] is a free and open source dynamic programming language created by Guido van Rossum. It is simple to learn and easy to read, even for those who do not know the language. Python is often

described as pseudo-code that actually works. Python is often used for rapid prototyping applications.

Python was chosen to implement the middleware because the author is familiar with it and because OntoMobil is implemented in Python too. Also, the support for Python in terms of RDF libraries is deemed sufficient.

4.1.3 RDF

RDF will be used to define and represent ontologies. The main reason is that the OntoMobil protocol that the middleware is supposed to work with is built using RDF. Using more advanced ontology, such as OWL, is considered future work.

4.1.4 Service and context representation and querying

Using technologies like OWL-S and WSDL-S on MANETs is found to be overly complex for the application of MANETs as was noted in section 3.1.3. OntoMobil therefore only supports a simple semantic interface consisting of a set of concepts.

Semantic service definition languages like OWL-S and WSDL-S also support these means to associate concepts with a service, both for categorisation and for inputs and outputs. Additionally they also include means to define pre-conditions and post-conditions. OntoMobil, however, does not support the latter two as of yet. It is therefore decided that semantic services are defined with a semantic profile with the following concepts associated with it:

category A concept to describe the service's category.

inputs A concept associated with each of the service's inputs.

outputs A concept associated with each of the service's outputs.

The standards that were looked at to define semantic service interfaces all used custom XML-based standards. As one of the focuses of the middleware is to be easy to use, writing XML documents should not have to be necessary. APIs will be provided that allow to define semantic services and semantic profiles, similar, but simpler than to the ones in OWL-S. There is a RDF serialization available for this, but it will be automatically generated by the middleware and does not have to be typed by the developer.

For each service a semantic profile is defined. A semantic profile is an object that contains attributes for the service's category concept, input concepts and output concepts. In early prototypes this profile object was also used directly when querying for services. The developer would define an ideal profile as a semantic profile, which would be handed to the middleware for discovery.

This approach worked, but broke when context-awareness was added to the middleware. This was because context was hard to be described in a semantic profile.

Context

Initial design The initial design (which was eventually never implemented) was to let developers define context on a network node. A network node would have an associated context object on which attributes could be defined. Additionally, developers would define reasoning rules to reason about context, similar to [41]. These reasoning rules would be distributed over the network and so would the context. There would be two kinds of context properties:

1. Static context. This is context that does not change often and which therefore can be cached on other nodes. Examples of this could be the screen resolution of a monitor.

2. Dynamic context. This is context that cannot be cached and is changing frequently. This can be the location of the node for example, or the temperature surrounding the node.

Whenever the value of a property is needed that is a dynamic property, a call would be made to the context object on the appropriate network node. A context would contain a registry of dynamic context properties and is able to determine their values by calling developer-specified functions. For example the location of a network node may be dynamic (the user may be walking around), therefore that property should defined to be a dynamic property and a function should be written that determines the current location and returns it. Such a function would take two parameters:

1. `subject` which is the resource to determine the property of.

2. `property` which is the property to determine.

The function should return a list of triples containing the requested information. As follows:

```
def determineLocation(subject, property):
   location = getLocation()
   return [ (subject, property, location) ]

context = context.Context()
context.registerDynamicContextProperty(NS['location'],
   determineLocation)
```

Rules would be used to do reasoning about context, examples of rules could be:

```
(?requesterContext ns:location ?l), (?serviceContext ns:location ?l)
   -> (?service ns:locationMatch ?requestedProfile)
(?serviceContext ns:brand ?brand) ->
   (?service ns:brandMatch ?requestedProfile)
```

This rule states that when the requester and the service are at the same location this means a location match between service and the requested profile (which is the ideal profile as specified by the requester), which could be a requirement of the service requester. The second rule states that if a service's context contains a property `ns:brand` with the value of some variable `?brand`, that means the service is a *brand match* for the requested profile.

Rules would globally hold, they would not be specific to one particular service discovery request. When services are discovered, one or more contextual constraints can be passed with the request, as follows:

```
contextConstraints = [ NS['locationMatch'] ]
services = node.discoverService(idealProfile, contextConstraints)
```

Which would search the network for services matching the `idealProfile` and where the contextual constraint `locationMatch` (as defined above) holds. It would also be possible to bind variables for the request, this would be done by passing in a hashmap with the `discoverService` call as follows:

```
contextConstraints = [ NS['locationMatch'], NS['brandMatch'] ]
vars = { 'brand': 'Dell' }
services = node.discoverService(idealProfile,
    contextConstraints, vars)
```

In this case it would find all services matching the `idealProfile`, that are in the same location as the requester and where the brand is 'Dell'.

Problems with this approach There are three important problems with this approach. First, it is quite complicated to understand. It involves understanding rules and adds quite a bit of complexity. The second problem is that the protocol the middleware should be able to use does not support rules or distributed context. The third problem is that implementing this system would be very complicated and a lot of work.

Final design It was decided to take a much simpler, but equally powerful approach. Rules were dropped, as was the idea to represent the ideal profile as a profile object. This idea was hard to let go, as it seemed nice from a design perspective, it provided symmetry.

The context was decided to be represented in the local ontology, as in the previous design. The local ontology will contain RDF representations of the network node and exposed services and their associated profiles, the developer can link context to those. Because RDF(S) is completely extensible it is, for example, possible to link a location to a node, or the number of current users to a service.

This solution allows the developer to define any kind of context, or any other ontology information for that matter, such as:

- *Node context*. Context such as the node's location, CPU usage or temperature.
- *Service context*. The number of requests it can handle per second, its up-time and so on.
- Any other context that the developer deems useful.

Assumption This idea is based on a very strong assumption: the representations of context match between nodes. In practice that means that a context ontology should be agreed on for all nodes.

For example, imagine querying for all services where the network node has a *location* property with the value “Basement”. Services on other nodes in the network may not have a *location* property at all, or the location may be represented differently (such as a GPS location). This assumption was also found in the literature, as described in section 3.5.4.

Possible solutions for this problem could be a *mapping service* similar to Edutella as described in section 3.4, or automated ontology matching. Finding a solution to this restriction is considered future work.

In order to query for services in a semantic and context-aware way, a query language had to be developed. A language that could be used to define an ideal profile, but also query the local ontology for context. This language was decided to be an extension of the SPARQL [24] language, which is a W3C standard to query RDF data sources. A more elaborate explanation on the reasons behind developing this language can be found in section 4.2.5.

The advantage to this approach, compared to the initial design, is that it is simple and that a query is self-contained. No rules are necessary and no context has to be distributed over the network. The middleware can simply extract the set of concepts from the query, use that to discover services and the rest of the query could later be used to query the local ontology of nodes that contain matching services. It was a lot easier to implement and could be supported by OntoMobil.

4.1.5 Matching engine

There are no semantic matching engines available in Python. Therefore one has to be implemented. Ideas were taken from [33]. It uses RDF instead of DAML and is simple, but it works.

4.2 Architecture

The high-level architecture of the developed system is shown in figure 4.1. The UML diagram of the whole system can be found in appendix 1. Chapter 5 will describe the developer API in more detail. Some implementation details are given in chapter 6.

The application is built by a developer utilising the middleware. The middleware is built to aid the developer to make developing the application more straight-forward. The protocol does the low-level

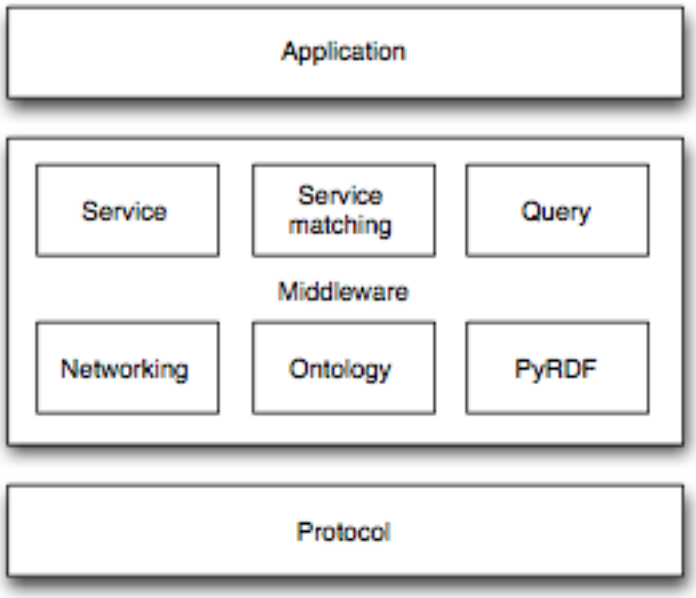

Figure 4.1: The system architecture

networking and discovery of services using facilities (such as service matching and query parsing) provided by the middleware.

4.2.1 Protocol

`Protocol` is an interface that implements a networking protocol. This can be any protocol, ranging from UDP broadcast to semantic matching routing to postal mail. An implementation that implements a simple totally connected peer to peer flooding protocol is part of the middleware and will be described in section 8.1.

The `Protocol` interface (figure 4.2) has the following methods:

discoverNodes discovers an initial set nodes in the network if the protocol requires it. This is called by the middleware when the node is started.

run this is the protocol's main loop, it is started on a separate thread. It can listen to incoming requests, such as service queries, the different pieces of the middleware can then be used to handle the request.

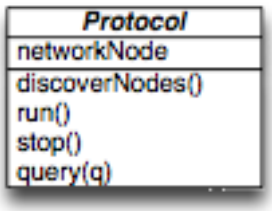

Figure 4.2: The protocol interface

stop this method must make the `run` method stop running.

query this method takes a RaSSQL query and returns its results, usually it will pass the query on to one or more other nodes.

A protocol that has to be supported by the middleware has the following requirements:

1. It should be able to describe services in terms of a set of concepts. One associated with the category and one for each of the inputs and outputs.
2. Ontologies have to be described in RDF.
3. For discovery the protocol should extract the concepts from the RaSSQL query using the `getProfile` method in the `query` package and use those to discover services.
4. The resulting services then have to be sent the RaSSQL query, which is locally translated into a SPARQL query that has to be executed locally on the service node's ontology.

4.2.2 Networking

A node in the network (usually one per machine) is represented by a `NetworkNode` object. A `NetworkNode` has an associated protocol that does all the underlying networking. It sometimes calls functions of the protocol, sometimes the protocol make callbacks to the `NetworkNode`. The `NetworkNode` is used to expose services and query the network. The UML diagram of the networking package is shown in 4.3.

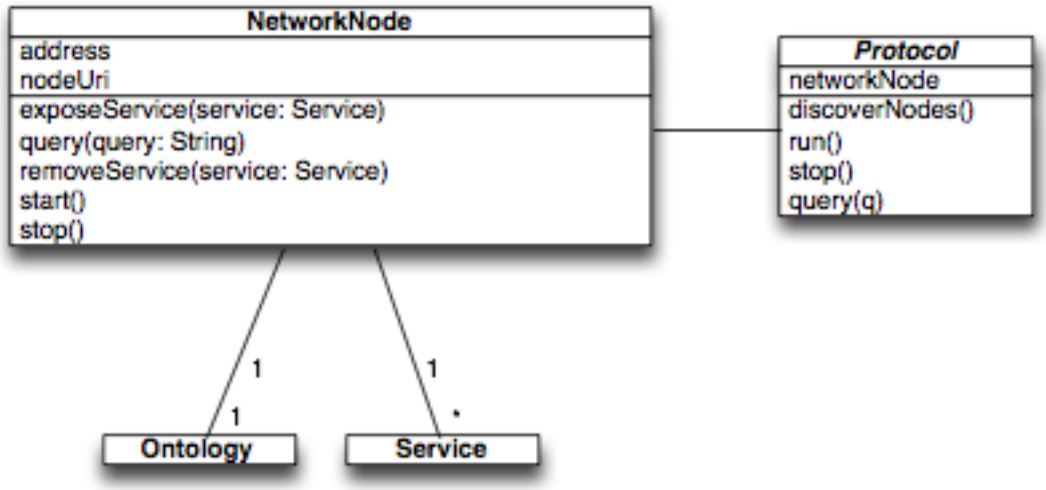

Figure 4.3: The network package

4.2.3 Ontology

Each node has its own local ontology containing all the concepts and its relationships that it is using or has used in the past, but it will also contain any other relevant information, for example context information or any other RDF information that has to be queried. So the ontology on each node contains:

- Concepts used in services this node advertises.
- Instances of concepts in the base ontology, such as `NetworkNode`, `Service` and `Profile`.
- Context information (that can be part of RaSSQL queries, more on RaSSQL later).

The UML diagram of the ontology package is shown in figure 4.4.

4.2.4 Services

Services are defined in terms of `Service` objects. A service consists of a `serviceUri` and a profile (a `Profile` object). A `Profile` defines the category of a service, the input concepts and outputs concepts, all in terms of URIs. The URIs being referred to are expected to be described in the local ontology. The UML diagram of the service package is shown in figure 4.5.

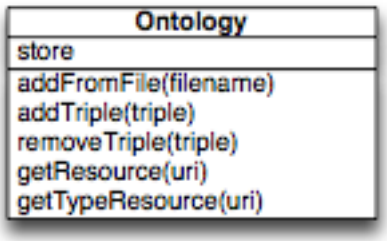

Figure 4.4: The ontology package

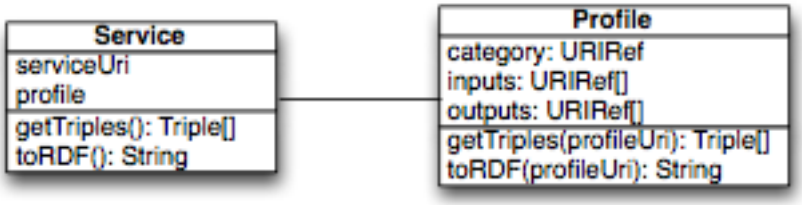

Figure 4.5: The service package

4.2.5 Query

In order to allow to query the network for services and also query the service provider's local ontology, a new query language had to be invented. This query language will be called RDF and Semantic Service Query Language (RaSSQL). RaSSQL is an extension of SPARQL. The UML diagram of the query package is shown in figure 4.6.

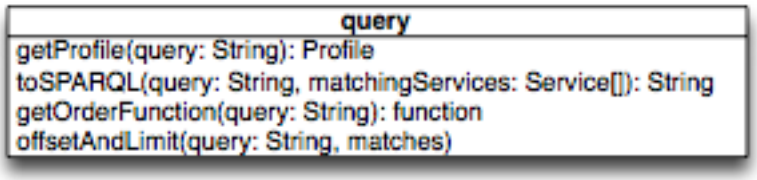

Figure 4.6: The query package

Why SPARQL is not enough

One could imagine that SPARQL is enough to describe a requested service, but it is not. The problem is that matches in SPARQL have to be exact, there is no concept of close matches or ranking of matches. SPARQL is fine for querying for exact profile matches, but close matches it cannot do. Therefore it needs to be extended to allow to specify an ideal profile.

Extending SPARQL

SPARQL is extended to allow for a definition of an ideal service profile and additionally to match any other RDF data present in the service's ontology. For this `SERVICE`, `CATEGORY`, `INPUT` and `OUTPUT` clauses are added to SPARQL.

SERVICE Defines a name for the variable that will contain the URI of the service that is being matched against. This variable can be used in the `SELECT` and `WHERE` clauses for further matching.

CATEGORY Defines the category concept a service should be linked to.

INPUT Defines an input concept.

OUTPUT Defines an output concept.

A semantic service that

- Falls into the category of the concept
  `http://www.zefhemel.com/ontology/travel#PlaneTravel`
- Takes an instance of the concept
  `http://www.zefhemel.com/ontology/travel#AirportOrigin` as an input
- Takes an instance of the concept
  `http://www.zefhemel.com/ontology/travel#AirportDestination` as an input
- Has an output of the concept
  `http://www.zefhemel.com/ontology/travel#Flight`

can be found using the following RaSSQL query:

```
PREFIX travel: <http://www.zefhemel.com/ontology/travel#>

SELECT ?service
SERVICE ?service
CATEGORY travel:PlaneTravel
INPUT travel:AirportOrigin
INPUT travel:AirportDestination
OUTPUT travel:Flight
```

Constraints can be put on the service node's local ontology. For example if a location property has been defined for nodes it is possible to query for services only in a certain location. It is possible to extend the previous query as follows:

```
PREFIX travel: <http://www.zefhemel.com/ontology/travel#>

SELECT ?service ?node ?score
SERVICE ?service
CATEGORY travel:PlaneTravel
INPUT travel:AirportOrigin
INPUT travel:AirportDestination
OUTPUT travel:Flight
WHERE { ?service rassql:node ?node .
        ?service rassql:score ?score .
        ?node travel:location 'Basement' }
```

The query will now only return services hosted on nodes that are located in the basement. In this way context can be used to make the search for services more accurate. A full specification of the RaSSQL language is given in section 5.6.

4.2.6 PyRDF

To allow developers to easily define RDF information in the local ontology, a special Python API is developed that allows developers to treat RDF concepts and instances as regular Python objects with

properties that values can directly be assigned to. What this API looks like is described in detail in section 5.4.2.

4.2.7 Service matcher

The matcher package contains some functions to match two profiles and associate a score according to their similarity. These can be used by the protocol to perform the actual service matching. The service matching process is described in section 6.1. The UML diagram of the matcher package is shown in 4.7.

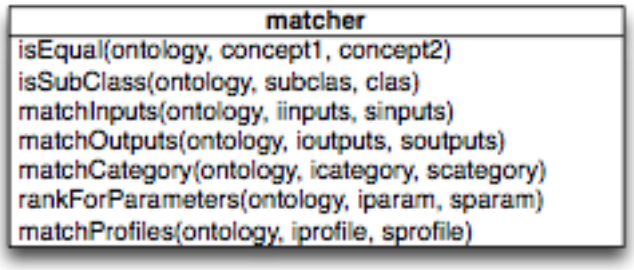

Figure 4.7: The matcher package

Chapter 5

Developer API

This chapter will specify the API for the middleware that can be used by the developer. First the base ontology will be discussed, followed by the network APIs, ontology APIs, context definition APIs, semantic service APIs and the specification of the RaSSQL language.

5.1 Base ontology

Initially, a base ontology will be loaded in each of the node's ontologies. The base ontology contains concepts and relationships that the middleware will use and can be extended by the developer. The concepts in this ontology can also be used within RaSSQL queries. An example piece of RDF code that instantiates the concept will be given for each of the concepts.

5.1.1 NetworkNode

The `NetworkNode` class represents a network node in the local ontology (there is at most one of those in each ontology), it can have one or more `offersService` properties linking to instances of the `Service` concept. An instance of a `NetworkNode` is automatically added to the local ontology when the node is created.

```
<rassql:NetworkNode rdf:ID="http://uri.org#Node1">
  <rassql:offersService rdf:resource="http://uri.org#Service1"/>
  <rassql:offersService rdf:resource="http://uri.org#Service2"/>
</rassql:NetworkNode>
```

5.1.2 Service

The `Service` class represents a service in the local ontology. It has a `node` property, representing the `NetworkNode` the service is running on. It also has an associated `profile` property (class `Profile`), which is a resource describing the service's profile. During querying a matching service it also has a `score` associated with it with a value between 1 and 4. Instances of the `Service` class are automatically created in the local ontology as services are exposed.

```
<rassql:Service rdf:ID="http://uri.org#Service1">
  <rassql:score>4</rassql:score>
  <rassql:node rdf:resource="http://uri.org#Node1"/>
  <rassql:profile rdf:resource="http://uri.org#Profile1"/>
</rassql:Service>
```

5.1.3 Profile

The `Profile` class represents a semantic profile for a service in the local ontology. It has a `category` property, which will link to its category concept, there are zero or more `input` properties that link to its input concepts and there are zero or more `output` properties that link to output concepts. Instances of the `Profile` concept are created when its associated service is exposed on the network node.

```
<rassql:Profile rdf:ID="Profile1">
  <rassql:category>http://uri.org#SomeCategory</rassql:category>
  <rassql:input>http://uri.org#SomeInput1</rassql:input>
  <rassql:input>http://uri.org#SomeInput2</rassql:input>
  <rassql:output>http://uri.org#SomeOutput</rassql:output>
</rassql:Profile>
```

5.1.4 Category

The `Category` is the root class the developer's category hierarchy should inherit from. It does not have to have any properties, but it can. The developer ought to subclass the concept. The sub-classed concept can then be referred to from a semantic profile.

```
<rdfs:Class rdf:ID="http://uri.org#SomeCategory">
  <rdfs:subClassOf rdf:resource=
    "http://www.zefhemel.com/rassql/rassql.rdf#Category"/>
</rdfs:Class>
```

5.1.5 Input

The `Input` is the root class the developer's input hierarchy should inherit from. It does not have to have any properties, but it can. The developer ought to sub-class the concept. The sub-classed concept can then be referred to from a semantic profile.

```
<rdfs:Class rdf:ID="http://uri.org#SomeInput">
  <rdfs:subClassOf rdf:resource=
    "http://www.zefhemel.com/rassql/rassql.rdf#Input"/>
</rdfs:Class>
```

5.1.6 Output

The `Output` is the root class the developer's output hierarchy should inherit from. It does not have to have any properties, but it can. The developer ought to subclass the concept. The sub-classed concept can then be referred to from a semantic profile.

```
<rdfs:Class rdf:ID="http://uri.org#SomeOutput">
  <rdfs:subClassOf rdf:resource=
    "http://www.zefhemel.com/rassql/rassql.rdf#Output"/>
</rdfs:Class>
```

5.2 Python's RDF library

The middleware uses Python's RDFLib [45] library for RDF representation. This library contains, among many other things, objects to represent RDF values such as literals (`Literal` class), URIs (`URIRef` class) and namespaces (`Namespace` class). Throughout the examples in this chapter these classes will be used. An important thing to know to understand the examples is that `URIRef` objects can easily be built from namespaces as follows:

```
NS = Namespace("http://uri.org#")
uri = NS['Something']
```

The `uri` variable will now contain a `URIRef` object pointing to the URI `http://uri.org#Something`. This notation essentially appends the string within quotes to the end of the namespace.

5.3 Network APIs

In order to be able to expose and discover services on a network, the node has to be connected to one. All network connectivity is dealt with by the Network APIs. It manages connections to other nodes and sends, receives and answers requests to and from other nodes. The Network API consists of two classes (`Protocol` and `NetworkNode`), but only one is used directly by the developer.

5.3.1 Network node

The `NetworkNode` object is the base of all networking. Each node has one instance of it, it runs a server that connects to other peers and sends requests to them and receives requests from them. It does the following:

- It *manages* the node's ontology.
- It *runs* the protocol that manages communication with other peers.
- It *exposes* services offered on the network.
- It *discovers* services on the network.

Each network node has a URI, default namespace (used to build URIs for internal use) and a protocol associated with it. After the `NetworkNode` has been constructed it can be started using the `start` method, stopped with the `stop` method, services can be exposed through the `exposeService` and removed using the `removeService` method. Services on the network can be queried using the `query` method.

An example:

```
NS_TRAVEL = Namespace('http://www.zefhemel.com/ontology/travel#')
node = rassql.net.NetworkNode(NS_TRAVEL['MyNode'], NS_TRAVEL,
   rassql.floodprotocol.PeerFloodProtocol)
node.start() # Start node
node.exposeService(ryanAirService) # Expose service
# Query services
results = node.query(query)
node.removeService(ryanAirService) # Remove service
node.stop() # Stop node
```

5.4 Ontology APIs

The middleware is based on independent ontologies. Each node maintains its own ontology, which initially only contains the middleware's base ontology but as services are exposed and RDF files loaded this ontology grows. Application developers also use the ontology to store any information that has to be queried, like context information. Once concepts and instances are loaded inside a node's ontology they can be used for describing services and context and they can be discovered by other nodes in the network.

5.4.1 Ontology

The main class of the ontology API is the `Ontology` class. Concepts and instances can be loaded into it in different ways.

- From a file. It can load a `.rdf` file into its store.

- Individual statements. It is possible to load individual RDF (`subject, predicate, object`) triples into the ontology.
- Using the PyRDF APIs (described in section 5.4.2)

Each of the ways to populate the ontology are demonstrated in the following example:

```
NS_TRAVEL = Namespace('http://www.zefhemel.com/ontology/travel#')
node.ontology.addFromFile('travel.rdf')
node.ontology.addTriple(
   (NS_TRAVEL['someThing'], NS_TRAVEL['size'], Literal(22)))
someThing = node.ontology.getResource(NS_TRAVEL['someThing'])
someThing.size = 23
someThing.users = ['Sean', 'Sinead']
```

5.4.2 PyRDF

PyRDF is a Python API that was developed as part of the middleware to allow developers to easily define RDF information, most importantly context. It allows the developer to treat concepts and instances of them as if they were Python objects with properties. PyRDF can be used both inside the RaSSQL middleware but also separately, the focus now is just on the use within the middleware. There are two classes a developer should be aware of.

1. `RdfResource`, which represents a resource, this can be largely treated like a simple Python object, properties can be set and so on.
2. `RdfType`, which is a direct subclass of `RdfResource`, it represents a concept in a RDF Graph. It is just the same as a RdfResource that registers itself as being an `rdfs:Class` as its `rdf:type`.

`RdfResource`s have properties, just like objects. Properties can have other resources, literals (strings, integers etc.) or lists (of resources or literals) as values. PyRDF tries to automatically guess what kind of type a property is. If you start using it as a list, it will act as a list, if you put or literals or `RdfResource`s in it, it will act as expected.

An example:

```
locationResource = node.ontology.getResource(NS['nodeLocation'],
    rdf_type = NS['Location'])
locationResource.longitude = 53.3472
locationResource.latitude = 6.2592
nodeResource = node.ontology.getResource(node.nodeUri)
nodeResource.location = locationResource
nodeResource.loggedInUsers = []
for username in getLoggedInUsers():
   nodeResource.loggedInUsers.append(username)
```

By default the property name is combined with the default namespace of the node, so for example if the default namespace is `http://www.zefhemel.com/ont#` and the property name is age, then the URI of the property will be `http://www.zefhemel.com/ont#age`. If a prefix is used followed by an underscore in the property name, like `j_description`, the default namespace will be overridden by the namespace associated with the `j` prefix. So, if the `j` prefix is associated with `http://www.zefhemel.com/ont/job#` the URI will be `http://www.zefhemel.com/ont/job#description`.

An example:

```
node.ontology.bind('j',
   Namespace('http://www.zefhemel.com/ont/job#'))
nodeResource = node.ontology.getResource(node.nodeUri)
nodeResource.j_title = 'Printer service'
```

5.5 Semantic Service APIs

The Semantic Service APIs are used to define semantic services.

5.5.1 Service

A service consists of two parts:

1. A URI, this must uniquely define the service on this particular node, i.e. there may be no other services running on this node that use the same URI.

2. A profile, this object describes the service in a semantic manner by referring to concepts in the local ontology.

Example:

```
ryanAirService = rassql.service.Service(NS_TRAVEL['RyanAir'],
   ryanAirProfile)
node.exposeService(ryanAirService)
```

This defines a service called `RyanAir` that has a profile `ryanAirProfile` (defined in the next section). The service is then exposed on the node.

5.5.2 Profile

A profile of a service gives a semantic description of the service. A profile consists of the following:

1. A categorization, a concept associated with the service that determines the category of service it belongs to. This refers to a concept in the local ontology describing a hierarchical service categorization.

2. A concept for each of the inputs.

3. A concept for each of the outputs.

Example:

```
ryanAirProfile = rassql.service.Profile()
ryanAirProfile.category = NS_TRAVEL['PlaneTravel']
ryanAirProfile.inputs.append(NS_TRAVEL['AirportOrigin'])
ryanAirProfile.inputs.append(NS_TRAVEL['AirportDestination'])
ryanAirProfile.outputs.append(NS_TRAVEL['Flight'])
```

This defines a profile for a service that belongs to the `PlaneTravel` category and outputs something in terms of a `Flight`. It takes two inputs: something in terms of a `AirportOrigin` and something in terms of a `AirportDestination`.

5.6 RaSSQL

The queries described in this chapter refer to RaSSQL queries. RaSSQL stands for RDF and Semantic Service Query Language. It is an extension of SPARQL [24] that allows to query a network for services in a semantic way and also allows to put further constraints on the results by filtering the results based on the service node's local ontology.

5.6.1 Syntax

The syntax of a RaSSQL query is as follows:

```
[PREFIX prefix: <namespaceuri>]
[PREFIX prefix: <namespaceuri>]
...

SELECT ?var1 ?var2 ?var3 ...
SERVICE ?serviceVar
CATEGORY <serviceuri>
INPUT <inputuri>
INPUT <inputuri>
...
OUTPUT <outputuri>
OUTPUT <outputuri>
...
[WHERE { <SPARQL constraints> }]
[ORDER BY ASC(?var1) DESC(?var2)]
[OFFSET <number>]
[LIMIT <number>]
```

PREFIX

`PREFIX`es allow one to abbreviate namespaces. The syntax is the same as in SPARQL. If a prefix is registered as follows:

```
PREFIX pre: <http://www.someuri.org/ont#>
```

One can easily refer to things in this namespace using the `pre:something` syntax, which would be expanded to `http://www.someuri.org/ont#something`. `PREFIX` statements are not required, but generally used a lot. By default one prefix is already available and that is the `rassql` prefix referring to the RaSSQL namespace.

SELECT

`SELECT` allows one to specify the variables in the query that one is interested in (and want to obtain the values from). Variables in RaSSQL all start with a question mark `?` followed by a series of lowercase and uppercase letters and numbers, like variables in many programming languages. The syntax for the `SELECT` clause and variables are the same as in SPARQL.

After the query has been executed the values of the variables can be obtained from the results as follows:

```
query = '... SELECT ?var1 ?var2 ?var3 ...'
results = node.query(query)
for result in results:
   print result['var1'], result['var2'], result['var3']
```

SERVICE

The `SERVICE` clause defines which variable in the query will be used to refer to the service's URI. This variable can be assumed to contain the URI of the service being matched against and can be used in `SELECT`, `WHERE` and `ORDER BY` clauses.

```
SELECT ?service
SERVICE ?service
...
```

CATEGORY, INPUT and OUTPUT

The CATEGORY, INPUT and OUTPUT clauses are used to specify the concepts of the category, inputs and outputs of an ideal service match. The category, inputs and outputs are defined in terms of URIs, either full URIs or in the prefixed form, like pre:something. The CATEGORY, INPUT and OUTPUT are used to define the concepts that are used to match against the different services on the network. A score is associated with each service (which can be obtained through the rassql:score property of the service).

```
PREFIX travel: <http://www.zefhemel.com/ontology/travel#>

SELECT ?service
SERVICE ?service
CATEGORY travel:Travel
INPUT <http://someuri.org#LocationOrigin>
INPUT travel:LocationDestination
OUTPUT travel:TravelPlan
```

WHERE

The WHERE clause can be used to put further constraints on the results returned, but also to obtain more information on the results. The syntax of the WHERE clause is the same as in SPARQL. It can contain triples that must match (and can contain variables) and FILTERs. For exact syntax refer to the SPARQL specification [24]. A WHERE clause is not required, but generally used.

```
...
SELECT ?service ?node ?score
SERVICE ?service
...
WHERE { ?service rassql:score ?score .
        ?service rassql:node ?node .
        ?node n:ram ?ram .
        FILTER(?ram > 256) }
```

ORDER BY

The `ORDER BY` clause can be used to sort the results based on one or more variable, either in ascending or descending order. The syntax of the `ORDER BY` clause is the same as in SPARQL. The default sort order is ascending, but this can be changed by surrounding the variable name with either `DESC(...)` or `ASC(...)`. The `ORDER BY` clause is optional. The variables being used in the `ORDER BY` clause must also appear in the `SELECT`.

```
...
SELECT ?service ?score ?ram
SERVICE ?service
...
WHERE { ?service rassql:score ?score .
        ?service rassql:node ?node .
        ?node n:ram ?ram .
        FILTER(?ram > 256) }
ORDER BY DESC(?score) DESC(?ram)
```

This will query the network for services, order them based on score (highest first) and amount of RAM on board (most RAM first).

OFFSET and LIMIT

The `OFFSET` and `LIMIT` clauses can be used to limit the number of results returned and to page through them. To limit the number of results returned to 10, for example, one can use the `LIMIT 10` clause. To only retrieve results 10-30, one can do the following:

```
...
SELECT ?service
SERVICE ?service
...
OFFSET 10
LIMIT 20
```

5.6.2 Usage

RaSSQL queries can be executed through a `NetworkNode`, using the `query` method. The result of this is a list of hashtables with each variable name as a key and its associated value as its value. An example listing of its use can be found below.

```
query = '''
PREFIX travel: <http://www.zefhemel.com/ontology/travel#>

SELECT ?service ?node ?score ?category ?location ?maxSpeed
SERVICE ?service
CATEGORY travel:Travel
INPUT travel:LocationOrigin
INPUT travel:LocationDestination
OUTPUT travel:TravelPlan
WHERE { ?service rassql:score ?score .
        ?service rassql:node ?node .
        ?service rassql:profile ?profile .
        ?profile rassql:category ?category .
        ?node travel:maxSpeed ?maxSpeed .
        ?node travel:location ?location }
ORDER BY DESC(?score) ?maxSpeed
LIMIT 10
'''

results = node.query(query)

for result in results:
   print '----------------------'
   for (k, v) in result.items():
      print k, '=', v
```

Chapter 6

Implementation

This chapter will detail some aspects of the implementation of the middleware.

6.1 Service matching

How service matching takes place exactly depends on the protocol, but service matching will generally happen in two stages. In both the `PeerFloodProtocol` and OntoMobil protocol (implemented and described, respectively in chapter 8) it works as follows.

6.1.1 Phase 1: Discover services through concepts

The utilities in the `query` package are used to extract the ideal profile of the service from the RaSSQL query. This ideal profile is then matched against other services on the network. Concepts similarity is being scored in a simple way. First concepts have to be matched, how this happens depends on the protocol. If the concepts do not match directly it is checked if they can match through subsumption. The way this is done depends on whether a concept is a service categorization, input or output concept.

The matching algorithm being used is based on [33]. Using the matching algorithm a score can be determined that describes how well two service profiles match. The possible scores are listed in table 6.1.

Globally the matching algorithm is very simple. Scores are assigned to how the categories, inputs and outputs match and the minimum of those scores is chosen to be the overall score (`iprofile` is

Table 6.1: Possible service matching scores

Score	Description
0	Fail
1	Partial fail
2	Invert subsume
3	Subsume
4	Match

the ideal profile as defined by the requester, `sprofile` is the service's profile that is being matched against):

```
def matchProfiles(ontology, iprofile, sprofile):
   categoryscore = matchCategory(ontology, iprofile.category,
        sprofile.category)
   inputscore = matchInputs(ontology, iprofile.inputs,
        sprofile.inputs)
   outputscore = matchOutputs(ontology, iprofile.outputs,
        sprofile.outputs)
   score = min(categoryscore, inputscore, outputscore)
```

Categorization

If *B* is a subclass of *A* and one requests a service in category *A*, then it is acceptable to also return services in category *B*. For example, assume there is a category `BuyComputer` which has a subcategory `BuyLaptop`.

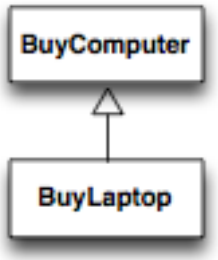

Figure 6.1: `BuyLaptop` is a subcategory of `BuyComputer`

If somebody would be looking for a service to buy a computer, returning a service in the BuyLaptop category would be just fine.

The category's score is determined using the following algorithm (icategory is the ideal category's URI, scategory is the URI of the category of the service matched against):

```
def matchCategory(ontology, icategory, scategory):
   '''Based on Stefan Tang\'s algorithm.'''
   if isEqual(ontology, scategory, icategory):
      return M_MATCH
   elif isSubClass(ontology, scategory, icategory):
      return M_SUBSUME
   else:
      return M_FAIL
```

Inputs

For service inputs it could works both ways. If a certain concept is requested, it sometimes is alright to return services that take something more general as its input. For example if one requests a service that takes a Laptop as an input, a service that takes Computers as input could be acceptable (if Laptop is defined to be a subclass of Computer). However, the other way around is usually more preferable. If a Computer is requested as input, a Laptop is fine as well. For the details refer to [33].

The algorithm to match the profile's inputs is listed below (iinputs is a list of the ideal profile's input concepts, sinputs lists the input concepts of the service being matched against).

```
def matchInputs(ontology, iinputs, sinputs):
   '''Based on Stefan Tang\'s algorithm.'''
   minOverallRank = M_MATCH
   if len(iinputs) != len(sinputs):
      return M_FAIL
   for sinput in sinputs:
      bestMatch = None
      maxRank = 0
      for iinput in iinputs:
         rank = rankForParameters(ontology, iinput, sinput)
         if rank > maxRank:
            maxRank = rank
            bestMatch = iinput
      if bestMatch == None:
         return M_FAIL
      if maxRank < minOverallRank:
         minOverallRank = maxRank
   return minOverallRank
```

`rankForParameters` is defined as follows:

```
def rankForParameters(ontology, iparam, sparam):
   if isEqual(ontology, iparam, sparam):
      return M_MATCH
   elif isSubClass(ontology, sparam, iparam):
      return M_SUBSUME
   elif isSubClass(ontology, iparam, sparam):
      return M_INVERT_SUBSUME
   else:
      return M_FAIL
```

Outputs

A service output is similar to the service categorization. If a `Computer` is requested it's perfectly fine to return a `Laptop` instead, as a `Laptop` is a kind of `Computer` (a subclass of `Computer`).

The algorithm for matching the output is listed below (`ioutputs` is a list of the ideal profile's output concepts, `soutputs` lists the output concepts of the service being matched against):

```
def matchOutputs(ontology, ioutputs, soutputs):
   '''Based on Stefan Tang\'s algorithm.'''
   minOverallRank = M_MATCH
   if len(ioutputs) != len(soutputs):
      return M_FAIL
   for soutput in soutputs:
      bestMatch = None
      maxRank = 0
      for ioutput in ioutputs:
         rank = rankForParameters(ontology, soutput, ioutput)
         if rank > maxRank:
            maxRank = rank
            bestMatch = ioutput
      if bestMatch == None:
         return M_PARTIAL_FAIL
      if maxRank < minOverallRank:
         minOverallRank = maxRank
   return minOverallRank
```

6.1.2 Phase 2: Filter services based on local ontology

Once nodes have been identified that contain a service that match the ideal profile, the RaSSQL query is sent to them. Locally the query is transformed into a SPARQL query that is being executed on its local ontology. The result of this query, which are the services that both matched the ideal profile and also fit the constraints put in the RaSSQL query, are returned to the requester.

6.2 RaSSQL engine

The transformation from RaSSQL to SPARQL is done by the `toSPARQL` function in the `query` package. `toSPARQL` takes a RaSSQL query and a set of matching services and turns it into a SPARQL query that can be executed on the local ontology.

Essentially what it does is strip the `SERVICE`, `CATEGORY`, `INPUT` and `OUTPUT` clauses, add an additional `FILTER` in the `WHERE` clause where it limits the matches to one of the matching services. It also removes the `ORDER BY`, `LIMIT` and `OFFSET` clauses. These clauses are removed because they do not make much sense and have to be reimplemented on the service discoverer's side. Remember, the RaSSQL query is sent to many nodes and the results are combined on the requesting node. The result sorting and limiting has to be implemented on that node.

`getOrderFunction` returns a function based on a RaSSQL query that, when used in the `sort` function of the matches, will sort the array based on the criteria as defined in the `ORDER BY` clause of the query.

`offsetAndLimit` performs the `OFFSET` and `LIMIT` clauses taken from a query on a set of results.

The SPARQL queries are performed using the Redland librdf library.

Chapter 7

Security

This chapter takes a look at the security aspects of this project. It covers the three important concepts in security (CIA):

- Confidentiality
- Integrity
- Authentication

Each of these will be looked at to see how applicable they are to the project. After that some general observations about security in MANETs will be made.

7.1 Confidentiality

Confidentiality deals with keeping information secret, people intercepting messages should not be able to read them. As my project functions within an ad-hoc network there are some confidentially features already in place. Ad-hoc networks on 802.11 can be setup with WEP encryption for example, which, although broken, still offers some confidentiality. However, this is only protection from people intercepting traffic from outside the network. As ad hoc networks route message in a peer-to-peer fashion to other nodes in the network, it is very simple for nodes in the network itself to intercept and read messages. If no additional layer of encryption is used these messages are sent in clear text.

My project has two main aspects:

- Service advertisement
- Service discovery

In the case of service advertisement encryption really does not make much sense. If a node is part of the network we assume it is allowed to see what services are being offered, intercepting messages is no problem, it is almost encouraged — it's advertising.

Service discovery is another issue, is service discovery a private matter? Do you want other nodes to find out what you are interested in? One may argue that service discovery is a private business, if somebody is looking for, say, a service offering pornographic materials; others should not have to know about it. However, the nature of an ad hoc network is a quite open one. Depending on the protocol a node basically starts asking around "hey, I need this and this service, do you know of someone that offers it?" nodes receiving such a discovery request can pass it on, likely sending its origin along with it, therefore service discovery is not private business and there is no reason to keep it confidential. No means of security is necessary here.

For service invocation, however, it is another story. Confidential information may be sent to the service, such as credit card numbers. Also we generally really do not want nodes in the middle to be able to modify our messages. Therefore it may be a good thing to consider using end-to-end encryption of data being sent between the service consumer and service provider on invocation. There are reasons, however, not to do this. One reason becomes obvious if we consider what kind of devices are connected to ad-hoc networks, we then see some potential problems. Devices used in MANETs often will be low-power devices, they do not have much CPU cycles to spare and they are meant to run for a long time on their battery. Therefore using strong encryption may not be an option because of all of its additional overhead. However offering end-to-end security it is definitely a thing to consider when it comes to service invocation.

7.2 Integrity

Integrity deals with if a person or node really is who he/she says he/she is. This however is very likely to be a much too big problem for this dissertation to solve. For example, consider the scenario where

a user wants to make a payment to somebody. He attempts to discover a service on the network to do this. Say that some node replies. Now, how do we know this node can be trusted? We are likely to give it our bank account information or credit card number, what if it's just somebody who tries to collect credit card information from people for non-legitimate purposes, how do we know a service can be trusted?

In "traditional" distributed systems one could set up a trusted third-party that issues certificates or tickets or something of this kind to trustworthy services, however in MANETs this model does not work. The network topology changes all the time, nodes leave and enter and there are no centralised servers we can trust. Setting up an infrastructure like this in MANETs is simply not feasible. A solution to this problem definitely has to be found, but I would say that service advertising and discovery on ad hoc networks is too young an area to deal with this problem. But in the future it has to be dealt with, it also applies to peer-to-peer networks in general. It is likely that it can fill a whole dissertation on its own, however, it is not something that can be solved by some scribbling in the margin.

7.3 Authentication

Authentication deals with determining a user's identity. This as well is something that is likely to be a problem in ad hoc networks. Most, if not all, authentication schemes use a user database of some kind to check a user's identity. As the network topology in an MANET changes constantly and there are no centralised servers, authentication is a problem. I see no real solution to this problem as of yet; I suppose MANETs are just not suitable for services that require user authentication.

7.4 Some general observations and comments

Because of the lack of centralised servers, MANETs pose a whole range of new security and in particular cryptographic problems. How do nodes agree on and distribute keys and how do you make sure the node you are talking to actually is the node you want to talk to? Especially the latter problem has traditionally been solved by Public Key Infrastructures (PKIs), however these have been centralised and don't seem suitable for a peer-to-peer infrastructure like MANETs.

However the nature of services offered on MANETs is quite different than "regular" distributed

systems. Services are often simpler, lighter and not business-critical. They are often used for games, finding printers and such, not a lot of damage can be done by third-parties.

7.5 Conclusion

Service discovery and advertising in MANETs is a still new and research-y area. Because it is in its early stages security is not really an issue yet, as there are few industry applications. However there are some places where security can be taken into account, such as service invocation. Also there is the area of security vulnerabilities. As an implementation platform for this project Python will be used. This avoids some common vulnerabilities, most notably buffer overruns.

Chapter 8

Evaluation

The middleware is evaluated in two ways. First, it is shown how a simple flood protocol can be implemented as the middleware's protocol and then it is described how OntoMobil protocol could be implemented as well. Second, it is shown how an application can be implemented on top of the middleware that demonstrates its features.

8.1 Flood protocol

To demonstrate that the middleware actually works, a simple flood protocol was written as an implementation of the middleware's `Protocol` interface. It runs a UDP server that listens to incoming messages. It also maintains a list of all the other nodes on the network (`Node` objects). It receives messages from nodes and responds to them.

8.1.1 Discovering nodes

While bootstrapping, the protocol will build a fully connected network as shown in figure 8.2.

As a `NetworkNode` is started the protocol's `discoverNodes()` function will find all the other `NetworkNode`s on the network. This is done essentially by doing a IP and port scan. As this protocol should function in a simulation environment with multiple nodes running on one system (each listening on their own port), a broadcast cannot be used. `discoveryNodes()` determines the current machine's IP, and then scans all 255 IPs in its sub-net, on each of those IPs it will send a `nodeDiscover`

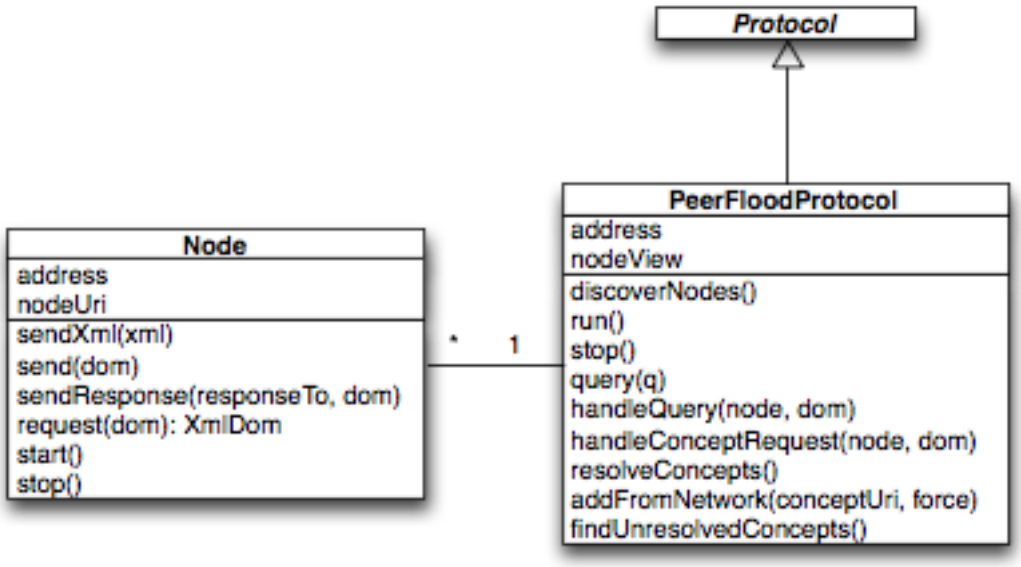

Figure 8.1: The flood protocol package

message to port 1338 through 1358 (ports are automatically allocated on start-up, starting from port 1338), whenever a `nodeAlive` message is returned to this `nodeDiscover` message, the source IP and port is added to the list of known nodes (`nodeView`).

8.1.2 Communication

Messages are being passed around as XML. The root element determines the message type. The following message types exist:

serviceQuery is an incoming (RaSSQL) service query that has to be handled and whose result has to be sent back.

conceptRequest is a request for a concept that the originating node does not know about yet. The local ontology is queried to find it and any triples related to it are returned.

nodeDiscover is the message that is broadcasted to all nodes on the network to find other `NetworkNodes` on the network.

nodeAlive is the message that an alive `NetworkNode` sends back.

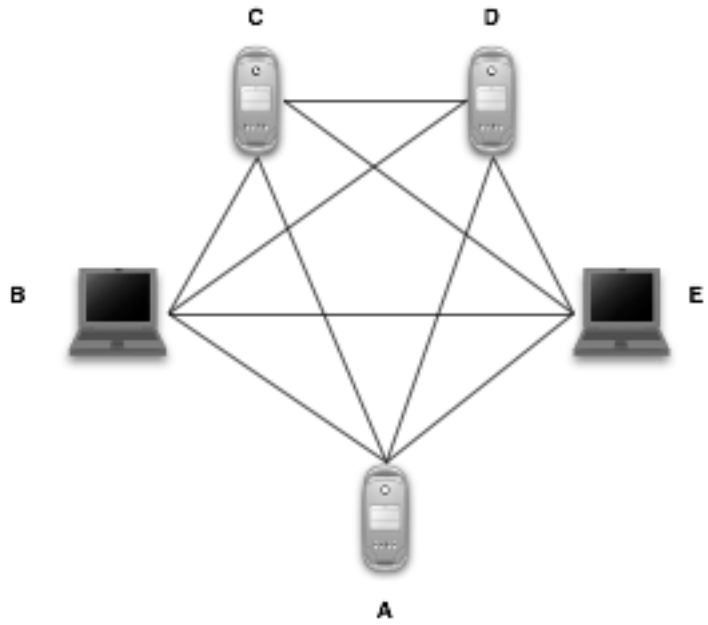

Figure 8.2: A fully connected network

8.1.3 Querying

As this is a simple flood protocol, when the `query(q)` method gets invoked it is simply passed on to each of the known `Nodes` in the `nodeView`. Their results are combined and returned to the user. A possible sequence of messages being sent after a `discoverService` request from node *A* is depicted in figure 8.3.

When a `serviceQuery` message is received from another node the `handleQuery(node, dom)` method is invoked. This is where the semantic matching takes place. It will act in two stages:

1. Extract the ideal profile from the RaSSQL query. And iterate over each locally exposed service and determine its matching score (using functions from the `matcher` package).

2. Using the matching services, translate the RaSSQL query into a SPARQL query (using the `query` package) that can be executed on the local ontology. Execute the query and send the results back as an XML document

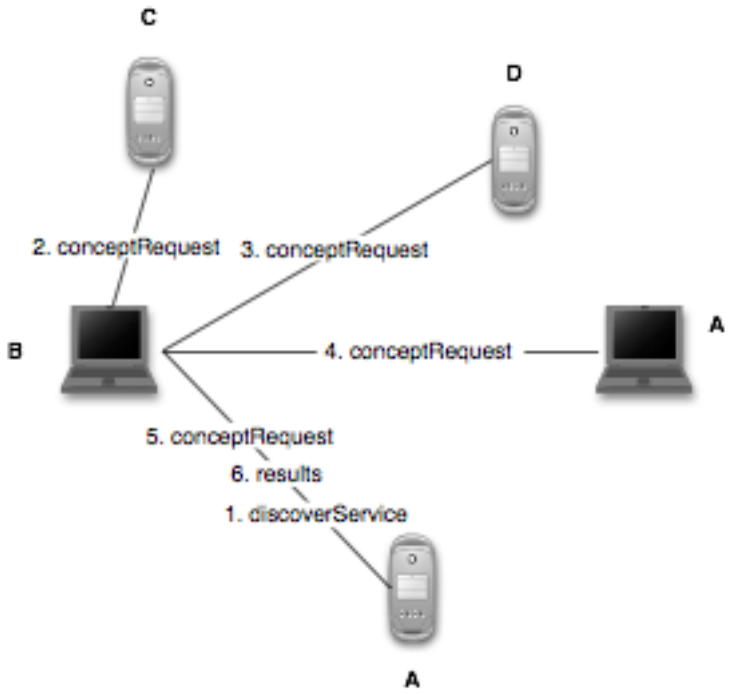

Figure 8.3: A possible sequence of messages sent during service discovery

8.1.4 Service matching

Service matching happens using the `matcher` package that comes with the middleware. The ontology's `isEqual` function that compares two concepts is implemented in a simplistic way, it simply compares the URIs. But it could be imagined that it would use a ontology matching techniques to determine if they are semantically equal, instead of syntactically (i.e. equal by URI). The OntoMobil protocol does this for example.

8.1.5 Concept distribution

Not all nodes necessarily have all the concepts in the network contained in their local ontology. Therefore, when a node receives a `serviceQuery` message that contains concepts it does not know, it will query the other nodes for them (using a `conceptRequest`), the nodes then return an RDF description of all the information they have on the concept and its relationships. The results are stored

in the local ontology. Therefore the local ontology will grow over time as more `serviceDiscover` requests come in.

8.2 OntoMobil, a scalable protocol

The flood protocol that was described in the previous section does not scale well. However, scalable protocols that do semantic matching on MANETs do exist, such as [6] as was described in section 3.3. This section will describe how this protocol could be implemented as a protocol for the middleware.

The protocol requires three buffers to be present on a node:

1. *Node View*, which contains addresses (or identifiers) of a subset of the nodes in the network.
2. *Ontology View*, which contains the node's local ontology.
3. *Concept View*, which contains a set of concepts from other nodes.

The *Ontology View* is already present in the middleware in the form of `node.ontology`. The *Node View* could be represented as a list of addresses, or as in the flood protocol described in section 8.1, as a set of `Node` objects. The *Concept View* can be implemented as another ontology (represented as a PyRDF `RdfStore`).

8.2.1 Discovering nodes

Before the protocol is operational it first has to be bootstrapped. This would be done in the `discoverNodes()` method of the protocol. The node will flood the network with node discovery messages. Once a node receives such a message it will at random decide (give some threshold) whether to reply to it or not. When the node receives enough replies to fill the *Node View* buffer it will enter normal execution. The *Node View* will now contain a random subset of the network nodes.

8.2.2 Querying

When a query is issued by the user, the protocol's `query` method is invoked given a RaSSQL query. The protocol will now operate in two phases.

The first phase is service discovery. What will happen first is that the ideal profile is extracted from the query using `matcher.getProfile`. This profile contains a set of concepts that can be used by the protocol to discover the associated services. The query containing the set of concepts is sent to a small subset of the nodes in the node's *Node View*. Not just the concepts' URIs are sent, but their whole definition including properties. The reason for this is better semantic matching. Concepts in this protocol just are names, not full URIs. Similarity between concepts is determined by compared both the concept's name and its properties.

When a node receives a service discovery query, it will not only look for exact matches in its *Concept View* and *Ontology View*, but also sub-classes and super-classes of the concept, as this information is used in the matching algorithm. The *network representation* of the concepts in the `Concept View` traces the concepts back to their source nodes. The RaSSQL query is then sent to the nodes containing matching services.

The second phase takes place at the nodes that contain matching services. On each node the RaSSQL query will be translated into a SPARQL query (using `query.toSPARQL`) which is executed on the node's local ontology. Its result will be returned to the requesting node. At the requesting node all results will be combined and returned to the user.

8.2.3 Concept distribution

Gossip messages are continuously passed around while nodes are running. A *gossip* message contains a set of concepts. When such a message is received, some matching will be done with concepts already in the *Concept View* and the node may or may not decide to store the concepts in its *Concept View*. As concepts are stored, an attribute is added that identifies the concept's origin node (for example `host:port` information) and properties.

Gossip messages are sent by all nodes in the network participating in the protocol. Concepts are randomly selected from the union of the *Concept View* and *Ontology View* and then sent to a subset of its *Node View*.

8.3 Example application: parking

In order to demonstrate the middleware, a sample application was built. It simulates a parking place finder. Given a location it will find all the parking places near that location. In the system parking places are represented as services. Locations are represented as instances of the `Location` concept. The location is a property of the node. The locations that are defined are:

1. aungierStreet
2. camdenRow
3. dameStreet
4. dawsonStreet
5. georgesStreet
6. graftonStreet

The programme consists of two parts: a script that runs a number of nodes and services and the client part that connects to the network and queries it for services. The application's ontology is shown in figure 8.4.

8.3.1 Services

The first part of the application is the script that starts a number of nodes and exposes some services on them. Each node represents a parking location. Table 8.1 shows the service profiles associated with the services defined.

Service	**Category**	**Input**	**Output 1**	**Output 2**
`parcAndSons`	`ParkingGarage`	`VehicleSize`	`EuroPrice`	`NumberOfSpots`
`blackYard`	`ParkingSpot`	`VehicleSize`	`Price`	`NumberOfSpots`
`aungierParking`	`ParkingSpot`	`VehicleSize`	`EuroPrice`	`NumberOfSpots`
`camdenArea`	`Parking`	`Size`	`EuroPrice`	`TotalArea`

Table 8.1: The services on the network

As said, each node also has a number of nearby locations associated with it. `parcAndSons` is located near `georgesStreet` and `dameStreet`. `blackYard` is located near `dameStreet`, `graftonStreet`

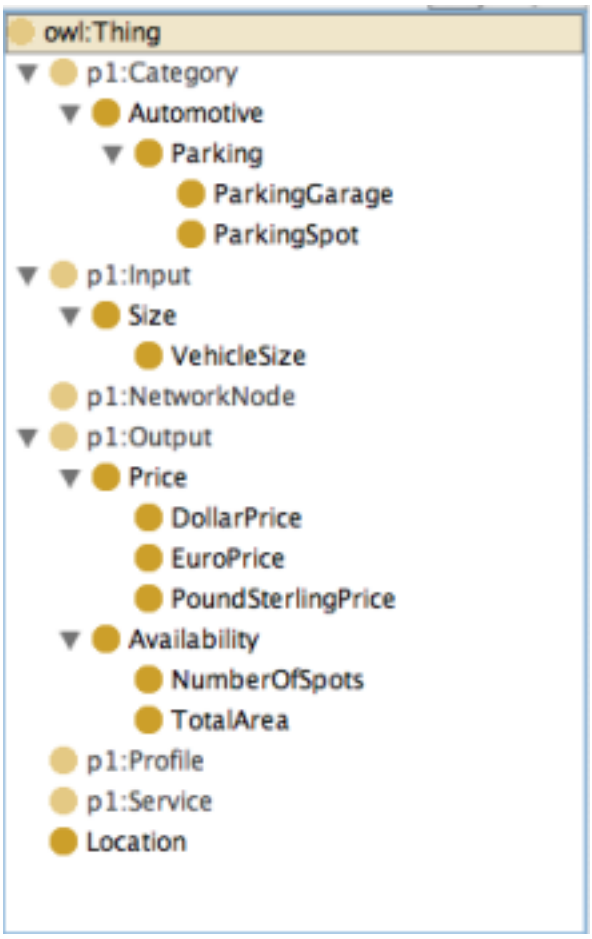

Figure 8.4: The parking concepts

and `georgestStreet`. `aungierParking` is located near `aungierStreet`. `camdenArea` is located near `camdenRow` and `aungierStreet`.

The whole listing of this programme will not be given here, instead just the definition of the `parcAndSons` node, service and its associated context will be listed. The other services are defined in a very similar way.

```
# Create and start a node
parcAndSonsNode = NetworkNode(NS_PARKING['parcAndSonsNode'],
    NS_PARKING, PeerFloodProtocol)
parcAndSonsNode.start()
parcAndSonsNode.ontology.addFromFile('sample/parking.rdf')

# Define a service
parcAndSonsProfile = Profile()
parcAndSonsProfile.category = NS_PARKING['ParkingGarage']
parcAndSonsProfile.inputs.append(NS_PARKING['VehicleSize'])
parcAndSonsProfile.outputs.append(NS_PARKING['EuroPrice'])
parcAndSonsProfile.outputs.append(NS_PARKING['NumberOfSpots'])
parcAndSons = Service(NS_PARKING['parcAndSons'], parcAndSonsProfile)

# Define context
r = parcAndSonsNode.ontology.getResource(NS_PARKING['parcAndSonsNode'])
r.near = []
r.near.append(NS_PARKING['georgesStreet'])
r.near.append(NS_PARKING['dameStreet'])

# Expose the service
parcAndSonsNode.exposeService(parcAndSons)
```

8.3.2 Client

The client is a simple GUI application that connects to the network and will then attempt to discover services near to it. The GUI is shown in figure 8.5.

The query in the GUI is a RaSSQL query and can be edited if desired. The results box shows the matching services, what node they run on and their score.

Again, because the code for the client is too long to list here, only the essential (querying part) will be given, edited a bit to get rid of the GUI plumbing code and printing the results to the standard

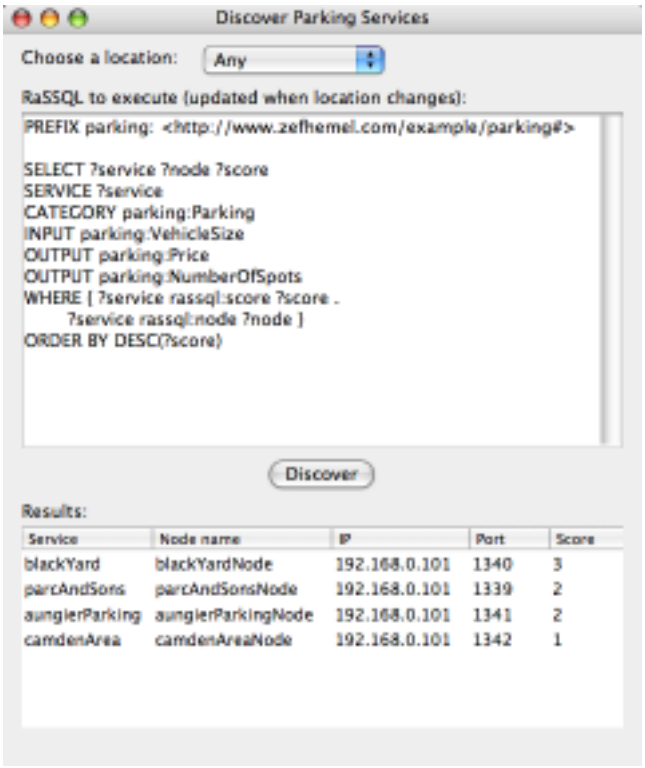

Figure 8.5: The client GUI

output instead:

```
# Create and start a node
node = NetworkNode(NS_PARKING['clientNode'], NS_PARKING, PeerFloodProtocol)
node.start()

# Formulate query based on locationUri variable
query = '''
PREFIX parking: <http://www.zefhemel.com/example/parking#>

SELECT ?service ?node ?score
SERVICE ?service
CATEGORY parking:Parking
INPUT parking:VehicleSize
OUTPUT parking:Price
```

```
OUTPUT parking:NumberOfSpots
WHERE { ?service rassql:score ?score .
        ?service rassql:node ?node .
        ?node parking:near <%s> }
ORDER BY DESC(?score)''' % locationUri

results = node.query(query)
for result in results:
   print result['service'], result['node'], result['score']
```

8.3.3 Testing

If the client is started before the service nodes it will discover no services. Once the service nodes are started, the client will be able to discover them, without restarting, as expected. If a location is specified only the services near that location appear with their associated matching score.

When the service nodes are shut down, the client will find that out after a while too. This shows that this programme would work in a MANET where nodes move in and out of the network.

Chapter 9

Conclusions

I think the results of this project in general are satisfying. In this chapter I will describe some of the things that I would have liked to do but did not have time for, the things I would have done differently and the things that went well.

Service invocation If service invocation would be part of the middleware it could be used straight away. Right now it just allows one to advertise and discover services, which is nice to prove a point, but is not enough to actually be used in applications. Semantic service invocation, however, is far from trivial. If more time would have been available, this would be the first thing I would work on. Ideas on how to do it could be taken from OWL-S's grounding. It is interesting to see how little literature can be found on this topic.

Context matching Another issue that has to be solved is context matching. The OntoMobil protocol allows one to discover services based on a set of concepts. These concepts can be part of independently developed ontologies, which is good, because MANETs are heterogeneous environments. The concepts are matched semantically, by looking at their properties and so on. However, when it comes to context matching, no semantic matching takes place. It is assumed that the context ontology is the same on each node. This ought to change. Solving this problem was too big an issue for this project, however, so is to be considered future work.

One can imagine developing the ultimate context ontology containing every single contextual

element ever needed, but in practice that will not be feasible. A solution could be found in something similar to Edutella's mapping service, briefly described in section 3.4.

Another issue with context matching is that the current approach does not scale well. The service concepts are used to discover nodes with services associated with those concepts, after that the context query is sent to every single one of these nodes. That works fine if there are only a couple of nodes with matching services, but as networks grow a more scalable way of context matching has to be invented.

Python Using Python for the implementation had both advantages and disadvantages. Python allows one to develop very user friendly APIs, especially PyRDF is a good example of that. Something similar could not be developed in Java. Python in general is a pleasure to programme in. However, on platforms like Java, the APIs for manipulating ontologies are richer. The Jena library, for example, has support not just for RDF, but also for OWL and has a built-in reasoner. The RDFLib used in Python is quite simple and only does the bare minimum.

RDF vs. OWL As OWL is a more expressive way to describe ontologies, it may have been a better choice to use OWL ontologies instead of RDF. It probably would have been more work to implement as well, especially in Python. The main reason to go for RDF and not OWL was that OntoMobil uses RDF. In OntoMobil RDF was chosen for its simplicity.

Implementing OntoMobil It would have been a good idea to actually implement the OntoMobil protocol for the middleware, but there was just not enought time to do this. Although explaining how it would be integrated gives you some feel for how it would work and that it *should* work, it does not prove it. Implementing it would have been a stronger statement.

Security Security was not a big issue in this project, but when service invocation will be implemented it may be something to be considered. It is more something for the future, however.

Simplicity One of my personal goals was to design APIs that would be simple to use. I know my supervisor does not like the term *simple*, but I am going to use it anyway. I have the strong belief that

programming is too hard, and whatever I can do to make it easier, I will do. I think frameworks and middleware really help developers. The middleware designed during this project indeed does make advertising and discovering semantic services a lot simpler (or: more straight-forward). I am satisfied with the APIs that were designed, they are reasonably easy to understand, concise and not too big.

I am particularly happy with PyRDF, which is does not only prove useful in the middleware, but could be used in any Python application that has to manipulate and query RDF data sources.

Impressions I had a good time the past months working on this project. The hardest thing was to understand the problem. It took a long time to actually figure out what needed to be done and all the things involved. It was also hard to decide on the focus and scope of the project. As I often said to people, there is a Ph.D. of work in every direction you look when it comes to research in context-awareness, semantic service matching, MANETs and probably most computer science areas. There is so much work left to do.

Future It is likely that OntoMobil will be implemented as a protocol for the middleware, by the author of OntoMobil. It will serve as an evaluation of the applicability of their protocol. I am glad that the middleware also proves to be of use to the OntoMobil people.

Bibliography

[1] Jun-Zhao Sun. Mobile ad hoc networking: an essential technology for pervasive computing. In *Proceedings of International Conference on Info-tech and Info-net (ICII'01), Beijing*, volume 3, pages 316–321, October 2001.

[2] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K.S. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3):33–40, July–September 2002.

[3] G. Kortuem, J. Schneider, D. Preuitt, T. G. C. Thompson, S. Fickas, and Z. Segall. When peer-to-peer comes face-to-face: Collaborative peer-to-peer computing in mobile ad-hoc networks. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, pages 75–94, 2001.

[4] L. Holmquist, J. Falk, and J. Wigstrm. Supporting group collaboration with inter-personal awareness devices. In *Journal of Personal Technologies, Special Issue on Hand-Held CSCW, Springer Verlag*, 1999.

[5] Leonard N. Foner. Yenta: A multi-agent, referral-based matchmaking system. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents (Agents-97)*, pages 301–307, New York, NY, 1997. ACM Press.

[6] Andronikos Nedos, Kulpreet Singh, Raymond Cunningham, and Siobhán Clarke. A Gossip Protocol to Support Service Discovery with Heterogeneous Ontologies in manets. Technical Report TCD-CS-2006-34, Distributed Systems Group, Computer Science Department, Trinity College Dublin, 2006.

[7] Rfc 2165: Service location protocol, 1997. http://www.faqs.org/rfcs/rfc2165.html.

[8] UPnP Forum. Upnp.

[9] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., 1999.

[10] S. Helal, N. Desai, V. Verma, and C. Lee. Konark - a service discovery and delivery protocol for ad-hoc networks. In *Proceedings of the Third IEEE Conference on Wireless Communication Networks (WCNC), New Orleans*, 2003.

[11] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha. Gsd: A novel groupbased service discovery protocol for manets. In *th IEEE Conference on Mobile and Wireless Communications Networks (MWCN 2002)*, 2002.

[12] Tom Broens. Context-aware, ontology based, semantic service discovery. Master's thesis, University of Twente, July 2004.

[13] Thomas Strang, Claudia Linnhoff-Popien, and Korbinian Frank. Applications of a Context Ontology Language. In Dinko Begusic and Nikola Rozic, editors, *Proceedings of International Conference on Software, Telecommunications and Computer Networks (SoftCom2003)*, pages 14–18, Split/Croatia, Venice/Italy, Ancona/Italy, Dubrovnik/Croatia, October 2003. Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split, Croatia.

[14] Michalis Vazirgiannis Christos Doulkeridis, Nikos Loutas. A system architecture for context-aware service discovery.

[15] J. PJubin and J.D. Tornow. The darpa packet radio network protocols. In *Proceedings of the IEEE vol. 75*, 1987.

[16] D.L. Nielson B.M. Leiner and F.A. Tobagi. Issues in packet radio network design. In *Proceedings of the IEEE Special issue on Packet Radio Networks*, 1987.

[17] C. Perkins. Ad-hoc on-demand distance vector routing, 1997.

[18] H. Bakht. Some applications of mobile ad-hoc networks, 2004. http://www.computingunplugged.com/issues/issue200409/00001371001.html.

[19] Xml remote procedure calls (xml-rpc), 1999. http://www.xmlrpc.com/spec.

[20] Universal description, discovery and integration version 3.0, 2004. http://uddi.org/pubs/uddi_v3.htm.

[21] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[22] J. Hendler T.B. Lee and O. Lassila. The semantic web. *Scientific American*, 2001.

[23] Owl web ontology language overview, 2004. http://www.w3.org/TR/owl-features/.

[24] Sparql query language for rdf, 2006. http://www.w3.org/TR/rdf-sparql-query/.

[25] Sparql protocol for rdf, 2006. http://www.w3.org/TR/rdf-sparql-protocol/.

[26] Abowd G. Dey. Towards a better understanding of context and context-awareness. In *1st International Symposium on Handheld and Ubiquitous Computing (HUC '99)*, 1999.

[27] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[28] Grit Denker David Martin, Mark Burstein and Jerry Hobbs. Owl-s 1.0, 2003. http://www.daml.org/services/owl-s/1.0/.

[29] Greg Meredith Erik Christensen, Francisco Curbera and Sanjiva Weerawarana. Web service description language (wsdl), 2001. http://www.w3.org/TR/wsdl.

[30] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web service semantics - wsdl-s. Technical report, UGA-IBM, April 2005. http://lsdis.cs.uga.edu/projects/METEOR-S/WSDL-S.

[31] Kunal Verma, Karthik Gomadam, Amit P. Sheth, John A. Miller, and Zixin Wu. The meteor-s approach for configuring and executing dynamic web processes. Technical report, University of Georgia, June 2005. http://lsdis.cs.uga.edu/projects/meteor-s/techRep6-24-05.pdf.

[32] T. Payne M. Paolucci, T. Kawmura and K. Sycara. Semantic matching of web services capabilities. In *First International Semantic Web Conference*, 2002. To appear.

[33] Stefan Tang. Matching of web service specifications using daml-s descriptions, 2004.

[34] D. Chakraborty, F. Perich, S. Avancha, and A. Joshi. Dreggie: Semantic service discovery for m-commerce applications. In *Workshop on Reliable and Secure Applications in Mobile Environment, 20th Symposiom on Reliable Distributed Systems*, 2001.

[35] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjoern Naeve, Mikael Nilsson, Matthias Palmer, and Tore Risch. Edutella: A p2p networking infrastructure based on RDF. In *Proceedings of the 11th World Wide Web Conference*, Hawaii, USA, 2002.

[36] Sun Microsystems. Jxta, 2006. http://www.jxta.org.

[37] L. Gong. Project jxta: A technology overview, 2002. http://www.jxta.org/project/www/docs/jxtaview_01nov02.pdf.

[38] Wikipedia: Service location protocol, 2006. http://en.wikipedia.org/wiki/Service_Location_Protocol.

[39] Thomas Strang, Claudia Linnhoff-Popien, and Korbinian Frank. CoOL: A Context Ontology Language to enable Contextual Interoperability. In Jean-Bernard Stefani, Isabelle Dameure, and Daniel Hagimont, editors, *LNCS 2893: Proceedings of 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS2003)*, volume 2893 of *Lecture Notes in Computer Science (LNCS)*, pages 236–247, Paris/France, November 2003. Springer Verlag.

[40] X. Wang. Ontology-based context modeling and reasoning using owl. In *Modeling and Reasoning Workshop at PerCom 2004*, 2004.

[41] Tapio Seppänen Jari Forstadius, Ora Lassila. Rdf-based model for context-aware reasoning in rich service environment. In *Proceedings of the 3rd Int'l Conf. on Pervasive Computing and Communications Workshops*, 2005.

[42] E. Valavanis, C. Ververidis, M. Vazirgiannis, G. C. Polyzos, and K. Norvag. Mobishare: Sharing context-dependent data and services from mobile sources. In *Proceedings of IEEE/WIC Int. Conf. on Web Intelligence (WI'03), Halifax, Canada*, October 2003.

[43] M. Khedr and A. Karmouch. Enhancing service discovery with context information. In *ITS'02, Brazil*, 2002.

[44] Guido van Rossum. Python, 2006. `http://www.python.org`.

[45] Daniel Krech. Rdflib, 2006. `http://rdflib.net`.

Appendix 1: Middleware UML Diagram