Federated Service Discovery

Interconnecting different Web Service Discovery Mechanisms.

Andreas Thuen

Institutt for informatikk

UNIVERSITETET I OSLO,
HØGSKOLEN I BERGEN

3 May 2015

Abstract

A Web Service consists of two parts, a provider and a consumer. The provider seeks to reply to incoming requests from the consumer, offering an answer to the requests sent. For the consumer to be able to discover and browse different Web Service providers, a Web Service Discovery Mechanism is often used. This mechanism lists all available Web Services that is registered with the Web Service Discovery Mechanism. This is a well functioning prevalent mode of operation, but the reach of these Web Service Discovery Mechanisms are often limited to the local network in which it resides. If the consumer wishes to discover services that resides outside the consumers local network, an application that can connect different Web Service Discovery Mechanisms is necessary. This application is what this thesis strives to design, create and test.

The thesis reviews different methods of achieving federation between different Web Service Discovery Mechanisms across a Wide Area Network (WAN). Through the use of gateways set up between the different instances of Web Service Discovery Mechanisms. The data can be transferred between these gateways, regardless of network type and size, as well as between different networks. This enables the user to use different Web Service Discovery Mechanisms, in different networks, and exchange information between these Web Service Discovery Mechanisms. The application is designed so that it is easy for other developers to add or remove features however they see fit, making it a solid foundation for further research on the field. It implements easily pluggable, predictable interfaces, that enables anyone who wants to test their own Web Service Discovery Mechanism or to use a different mechanism to transport the data between the different instances of the application.

The application has been tested in multiple different environments, and proven to work as intended in all of these. The biggest scenario involved 45 different computers distributed across Europe, which is representative of a WAN, and thus is a very good test considering the types of environments the application is intended to run in.

Preface

This thesis is written as a part of my master's degree in Computer Science at the University of Oslo, Institute of Informatics. The degree is a joint effort between the University of Oslo, UniK University Graduation Center and Bergen University College.

I have always been interested in how and why things work as they do. Ranging from a simple light bulb, to a computer. To be able to understand and review the inner workings of things makes them far more comprehensible, and it also helps understanding why they work. Designing, implementing and testing this application has been a very rewarding and educating process and the learning outcome of this has been a lot wider that I had anticipated.

I would like to thank Frank T. Johnsen for continuous feedback and support through the process of writing this thesis, Professor Carsten Griwodz for very good feedback on testing and writing, as well as facilitating the testing of the application in the Nornet Core. I am also grateful for Trude Hafsøe Bloebaum and Professor Knut Øvsthus contributions as supervisors for the thesis, as well as the team at Nornet helping med to be able to test the application in the Nornet Core distributed test bed.

A special thank you also goes to my partner Camilla, for her patience and support with me during the work on this project.

Andreas Thuen

Bergen, May 2015

Contents

1. Introduction 6
   1.1 Central Terminology 8
   1.2 Problem Statement 8
   1.3 Premises 9
   1.4 Scope and Limitations 10
   1.5 Research Methodology 11
   1.6 Contribution 12
   1.7 Outline 12
2. Background and Requirements Analysis 14
   2.1 Military Context 14
   2.2 Technological Background 15
      2.2.2 SOA realized using Web Services. 17
      2.2.3 Service discovery in-depth 20
      2.2.4 Service discovery standards. 22
      2.2.5 Communication methods 26
      2.2.6 WAN Mechanisms 27
      2.2.7 Response times 31
   2.3 Related Work 31
   2.4 Requirements summary 35
3. Design and Implementation 36
   3.1 Design 37
      3.1.1 Design decisions 38
      3.1.2 General design 40
   3.2 Implementation 57
4. Testing and Evaluation 61
   4.1 Unit Tests 63
   4.2 Functional tests 65
   4.3 Performance tests 67
   4.4 Test environments 68
      4.4.1 Test tools 68
      4.4.2 Small scale testing 69
      4.4.3 Medium scale testing 71
      4.4.4 Large Scale Testing 74
5. Conclusion and future work 79
   5.1 Conclusion 79

5.2 Future work. 80
Figures: 80
Tables: 82
References 82
Appendices 88
Appendix A – List of abbreviations 88
Appendix B – Technology Basis 89
Technology basis 89
Java-WS-Discovery 89
UDDI 90
Mist 92
ActiveMQ 93

1. Introduction

For many years, military research and technology was far beyond what the civilian population was using, they used a vast amount of money and the all the research was cutting edge. But when the cold war ended in 1989, the amount of resources set aside for military research and development has been steadily decreasing ever since. In the same period of time, the pace of the civilian research and development has rapidly increased, and the limited resources of the military has not been sufficient to keep up with the civilian development. Due to this change, the military has been forced to think differently, and try to adapt the technology initially developed for civilian use, if this succeeds the military will be able to keep the costs low while using civilian equipment, also referred to as Commercial off-the-shelf(COTS) products.

For many of the same reasons as the use of civilian equipment was introduced to military use, the focus on alliances and cooperation between nations has become is increasingly popular. This makes each nation stronger through unity, reducing the amount of forces needed, which saves money. These alliances also enable the different parties to share technology, which reduces the amount of resources needed for research and development.

The North Atlantic Treaty Organization(NATO) is focusing on working as a federation, there are several definitions and interpretations of a federation, but in this thesis the following definition is applied:

"A federated body formed by a number of nations, states, societies, unions, etc., each retaining control of its own internal affairs [1]*."*

In this setting the idea is that all NATO member countries will take part in the federation.

There are two different ways to do this, either a common guideline for software and hardware is released, and every party in the collaboration will have to use this equipment. The other way is to let everyone use their own custom solutions, and then try to make all of these proprietary solutions work together as a whole. NATO has chosen the second one of these approaches, creating a need for common standards.

In order to achieve this, NATO has decided to apply the Service Oriented Architecture (SOA) way of thinking, where all nations will contribute to the federation with their services, and allowing other nations to use these services when needed. This can for example be done by using a shared listing where all services provided are listed, for authorized users to browse and invoke these services if they want to.

SOA is a paradigm that gives a number of principles for how to build service oriented distributed systems. Two of the most basic SOA design principles are loose coupling and the use of standardized contracts. The first principle says that each component of a SOA system should work as intended, regardless of what other components that are connected. This design enables each component to be switched with a different component implementing the same interface, thus enabling loose coupling. This is achieved through the use of the second principle mentioned, namely the use of standardized contracts. The most common way to realize SOA is through the use of Web Services, this is also the

technology that NATO has chosen. The World Wide Web Consortium(W3C) has released a set standard, describing in detail how a Web Service should work.

> "A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-process able format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [2]."

A third principle of SOA is that all services should be discoverable, i.e. that it should be possible to dynamically find out which services exist, and how to connect to them. This process is known as service discovery, which W3C defines as follows:

> "Discovery is the act of locating a machine-processable description of a Web Service-related resource that may have been previously unknown and that meets certain functional criteria. It involves matching a set of functional and other criteria with a set of resource descriptions. The goal is to find an appropriate Web Service-related resource."

There exists several well-developed Web Services discovery and publishing services today, but there is no easy way to combine these technologies so that they can reach over a wider range, for example: between two different networks. This can be utilized in a number of situations, for instance when you want to share a Web Service with someone outside the local area network (LAN) you are connected to. A connection of multiple LANS is often referred to as a Wide Area Network(WAN) a WAN can for example be the Internet, the following defines the term WAN.

> -***"****A wide area network (WAN) is a network that exists over a large-scale geographical area. A WAN connects different smaller networks, including local area networks (LAN) and metro area networks (MAN). This ensures that computers and users in one location can communicate with computers and users in other locations. WAN implementation can be done either with the help of the public transmission system or a private network* [3].***"***

The goal of this thesis is to enable easier interconnection of Web Services in operations. The focus of the work is on designing and implementing a mechanism that enables the interconnection of multiple different Web Service Discovery Mechanisms, so that different systems, from different domains, can connect to each other and publish their services across a WAN. By enabling easy interconnection of these mechanisms, which offer the same basic service, but do so in different ways, one can always choose the mechanism that best meets the criteria for the situation, and still having the same endpoint interface in the commanding headquarters.

1.1 Central Terminology

Through reading this thesis, you will be introduced to several concepts and terms, some which are more important than others. The most significant and widely used terms have been summarized quickly here to enable the reader to gain a quick insight in their meaning.

SOA:

SOA – Service oriented architecture is an approach used to model business processes after. One of the main purposes and intensions of SOA is to facilitate and encourage to the use of standardized principles for building distributed systems. It also promotes the use of loose coupling and other good practices that makes software easier to create, maintain and use. A more thorough explanation and definition of SOA can be found in section 2.2.2.

Due to its ease of use and flexibility, you can now find Web Services in almost every computer system that is connected to the Internet or an internal intranet for a corporation. Some heavyweights are; Banking and Finance, Hospitals, Public management, Schools and many more. Web Services can be found in almost every system that involves retrieving, altering and adding information to a system or a database. A Web Service system consists of several parts and different mechanisms, further described in section 2.2.2 SOA realized using Web Services.

1.2 Problem Statement

The thesis is focused around SOA, and the realization of SOA through Web Services. The aim is to provide a mechanism that is able to distribute service information across the entire federation, without compromising each autonomous partner's ability to independently determine how to provide service information within their own domain.

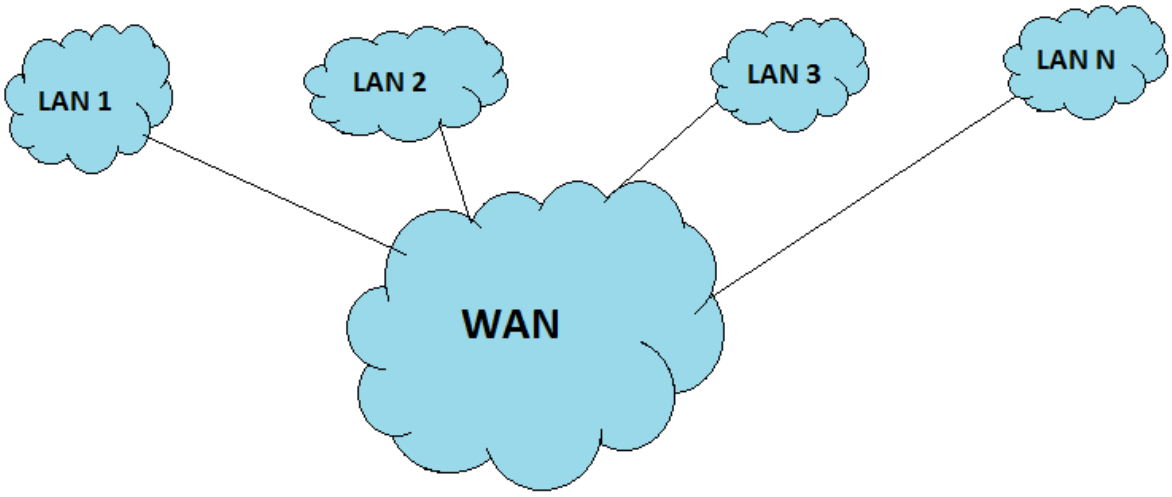

Figure 1 Multiple independent LANs connected via a WAN

In addition, the mechanism provided must be able to provide this information even when the partners in the federation are not directly connected to each other, but communicate via a WAN

such as the Internet. Figure 1 shows such a network set-up, where each of the four LANs represents the network of an autonomous partner, communicating via a WAN.

The goal of this thesis is to enable the exchange of service information in a federation consisting of N independent networks, using M different service discovery mechanisms, across a WAN.

The federation mechanism should be able to support a range of different service discovery mechanisms without requiring any modifications or extensions to any of the supported service discovery mechanisms. Furthermore, it should be possible to add support for further mechanisms as needed.

1.3 Premises

In order to ensure a common understanding of the task and what criteria are important when evaluating the application being developed, a set of premises has been identified:

An overreaching premise for the task is to develop the application in accordance with SOA guidelines and principles [4]. The SOA guidelines and principles is a list containing several different features and principles that is considered good SOA practice. This list includes features such as: Loose coupling, service reusability, service statelessness, and many more. By following these, the amount of work required to adapt the system to new or existing applications are kept at a minimum.

The system should be able to support standardized Web Service Discovery Mechanisms and service descriptions out of the box, also the overhead of implementing support for a new mechanism should be kept at a minimum. This supports the SOA way of thinking, by making it easy to add or exchange modules in the system at minimum amount of effort.

The system should be implementable as a part of a federation, where each node has no control over the other nodes systems. This means that it must be adaptable to many different technologies, as well as offering a standardized endpoint between the different instances of the system. This premise ensures that different parties in NATO are able to use and utilize the system, regardless of what system they are using.

The system has to be able to work regardless of what type of network it is deployed in. This means that it can not depend on any mechanisms that may be supported by some networks. The system has no predefined area of work, and must therefore support any type of network. Making the system network independent greatly increases the number of different settings it can be utilized in.

1.4 Scope and Limitations

In order to achieve an as good as possible end result for the federation mechanism, it is important to know where to focus and put extra effort. The scopes and limitations can be seen as a set of rules, governing what the application being developed should do, and what issues it should not address.

The main intent of the mechanism is to provide an interface through which any type of service discovery protocol can be connected. The main focus of the design and development of the federation mechanism is on the application layer. Issues residing in other layers will not be in the scope of this thesis.

The aim is to create a mechanism that allows the user to connect any different type of Service Discovery protocol with a minimal amount of coding and effort required. However, the system will not be implementing multiple different Service Discovery mechanism, only facilitate and prepare a generic endpoint that can be used to achieve this. This feature will be demonstrated by implementing one or two Service Discovery mechanisms, functioning as a guideline for other developers wanting to expand the reach of the application.

The federation mechanism must be a distributed system, so it is essential that it can be run in multiple locations simultaneously. The different instances should not need any prior knowledge to the other instances, other than where to connect to them. The mechanism providing the data transfer is also going to be a third party mechanism that can be changed to any such application by those who want to. It is therefore also important that the system sends and receives data that are on a standardized format.

Even though the data transferred between the different instances of the application may be sensitive data, which should be kept secret to others that may be inside the network, or eavesdropping the communication channel, the application takes no measures to ensure the integrity or security of the data. The data is delivered as a best effort service, and any additional security or quality of service (QOS) must be provided from elsewhere.

The user should be able to control the amount of information that is sent to other instances of the system. This should be enabled through a simple mechanism that will be incorporated as a part of the system. The system will not be able to automatically determine what information to share and not to share at this point.

1.5 Research Methodology

Steve Denning's engineering design approach [5] focuses on four main steps in which the process is divided:

1) Perform requirements analysis
2) Derive a specification based on the requirements
3) Design and implement the system
4) Test the system

In the engineering approach the hypothesis is that the system fulfills the specification and thereby meets the requirements. [6]

The thesis is modeled after this approach and is divided into the following chapters and related process steps.

The first step of the process is to analyze the requirements that the project needs to meet in order to meet the specified demands. Chapter 2 is a review of all the involved components and their function as a part of the whole system. Key components are identified and given sufficient attention for a working prototype to emerge at an early phase.

The second step is covered in the first part of the third chapter, which presents an implementation plan and discusses how the involved components are included as a part of the federation mechanism. The specification emphasizes the main components, and introduces these first, before it progresses to the less vital components.

In step number three the design made in the previous step is implemented - this is done in the second part of chapter three. The first goal of this step is to implement a working application that serves as a foundation for the rest of the development. Further development is done by utilizing a light version of the SCRUM/AGILE methodology, where each sprint will be 1-2 weeks. Each sprint has an objective that is to be completed within the time of the sprint. Between each sprint, an evaluation is conducted to check the level of achievement for the current goals. Goals for the next sprint are also decided on. This enables easier progress tracking and makes is easier to check for inconsistency or errors in an early phase, making them easier to avoid.

The final step is covered both in chapter 3 and 4. Unit tests of different components are performed alongside the development. This is done in order to ensure that each component works as desired. It also encourages loose coupling and reusable code, since each component should be testable by itself, without any dependencies to other components. The finished product is also tested for the most common use cases. Finally, there are tests where the application is evaluated to confirm that it meets the requirements made in the second step.

1.6 Contribution

Through the work with this thesis several questions and issues has been raised. Every one of these have been reviewed and evaluated against the premises and requirements for the system, in order to find the best solution possible.

By reviewing previous conducted research on the field, several useful issues and topics were introduced. We could then adapt and focus the work to be able to find answers to these issues.

Multiple different Web Service discovery standards have been reviewed and compared to each other. This was done to establish a common ground from which a Web Service could be described and saved in applications. This result is intended to work with any given Web Service Discovery standard.

A new framework for storing and synchronizing these services between multiple different repositories has been made and tested, this enables users to always have the latest information easily accessible.

1.7 Outline

This thesis consist of five chapters, each chapter addresses different parts of the process from the very beginning to a finished product. This first chapter introduced the reader to the task, along with the main terminology and concepts. The remainder of the thesis is organized as follows:

The second chapter is in many ways a more thorough review of the technology that has been utilized to solve the problem. Different aspects that affect the way the application should be solved are also explained in this section. The chapter also reviews and explains the requirements for the federation mechanism, followed by a short summary about how each requirement is going to be fulfilled. Another important part of this chapter is also the reviewing of related work, where similar work is evaluated against the criteria of this task. This process enables us to learn from previous experiences and can avoid excessive work.

The second chapter is in many ways a more thorough review of the technology that has been utilized to solve the problem. Different design aspects that affect the way the application will work are also explained in this section. The chapter also reviews and explains the requirements for the application,

followed by a short summary about how each requirement is going to be fulfilled. Another important part of this chapter is also the reviewing of related work, where similar work is evaluated against the criteria of this task. This process enables us to learn from previous experiences and avoid excessive work.

The third chapter addresses the design and implementation of the system. Section 3.1 Design starts by looking at the design of the system from a birds eye view. Each component and its interaction with the other components are quickly described before each components and its functions as a part of the system is described in greater details. Section 3.2 Implementation reviews the actual implementation of the design described in section 3.1 Design, any deviates from the original design is described and justified in this section.

Chapter four is a review of the finished application. The chapter first describes the different test types that are conducted on the system, and why they are conducted. Furthermore the chapter swiftly describes the different environments the application is tested in, and the differences between these. The results from the different tests are then reviewed and compared. The results are then reviewed as a whole, and compared to research conducted on the area.

The last chapter reviews the thesis and the application as a whole. It is measured against the criteria and premises listed in chapter one and two. The last section then looks at possible future improvements and future work that could improve or extend the reach of the application.

In the very end there are tables of figures, tables and references is included, as well as any appendices.

This project plan, represented through a Gant diagram displays the progress of the work and what parts have been emphasized through the different phases of the project.

There are also two appendixes, a containing a list of abbreviations and their meanings. As well as appendix b, where third party technologies that has a great influence on the task is reviewed and documented.

2. Background and Requirements Analysis

The intended outcome of this thesis is a system that enables interconnecting multiple Web Service Discovery Mechanisms residing in different LANS. Through this connection these mechanisms should be able to share their services with each other, and make all shared services available to consumers in all the LANS that are connected to the system. The system is intended to work across LAN boundaries. This includes the case where the LANs are connected via a WAN, e.g. over the Internet.

There are many different considerations to take when designing and developing this federation mechanism. Through this chapter you will find the requirements and background criteria that the thesis is based on.

2.1 Military Context

In order to support efforts where multiple independent partners, such a military units from different nations, come together to solve a common mission, NATO provides guidance on how to build a federation information infrastructure. The NATO Network Enabled Capability (NNEC) was through the NNEC feasibility study [8], the first such effort to identify SOA and Web Services as the key enabling technology.

More recent efforts, such as the Federated Mission Networking (FMN) [9] and the NATO work group IST 118 SOA Recommendations for Disadvantaged Grids in the Tactical Domain [10], aims to provide further guidance on the usage of SOA and Web Services in a military context. Disadvantaged grids are networks operating in non-optimal environments, where stable unstable links and low bandwidth may be issues.

For military purposes, a vast amount of different data is sent, both in size and in type. The demands for stability, dependency, scalability and adaptability are very high, and failure for a system to work as expected can have drastic consequences [11]. There are also some major challenges in conjunction with network-based communications in disadvantaged grids [12], such as a battlefield or in an operation [13]. The nature of these types of operations and herein the networks used, are highly dynamic, which requires the networks used to be optimized for the task. An operation can consist of several different systems, depending on which layer of the operation you are in, in order to make all of the systems interoperable, federated service discovery is a solution that can save the users a lot of time and extra work. With federated service discovery enabled, all of the systems can exchange information between each other, across system borders. A system can be a unit, a location or a country, depending on how you build and structure your network. The bottom line is that there will always be multiple systems, all containing different services that can benefit other units, and that is why we want these services to be available regardless of where you are and what network you are a part of.

An example illustrating this is if you think of an operation that consists of three types of units. The stationary units, which are administrating the operation and in charge of all the other units. For these units neither power consumption nor size of the communications equipment is any problem, they can also usually have the benefit of high-speed cabled Internet, as well as usually good time for planning and deployment The next type of units are mobile, motorized units. This can be many

different types of vehicles. They do have some limitations in power consumption and equipment size, but are not very restricted by this. However, they move around a lot and hence the signal strength and data rate may vary throughout the mission. The last group is units on foot, they are highly mobile, and the conditions can vary from very good to offline. The units on foot are also restricted by the amount of weight they can carry, therefore the equipment may not be too big or too heavy. The foot units may also have to reach quickly and be set up and organized in a matter of hours. This scenario emphasizes the need for different equipment and service running on said equipment for each type of units. Some Service Discovery mechanisms are also faster to set up and more agile than others, making them easier to use in an ad-hoc fashion than others.

2.2 Technological Background

The goal of this thesis is to create a federation mechanism for Web Service Discovery Mechanisms. The federation mechanism has two main purposes; to work as an integrator between Web Service Discovery protocols and to transmit the information gained from the connected Web Service Discovery protocol across a WAN to other, similar nodes. Figure 2**Error! Reference source not found.** illustrates how the federation mechanism is placed between the service discovery mechanisms and the WAN, and functions as a bridge between the different Web Service Discovery protocols. The main focus of the application will be to implement and work on the WAN mechanism that will ensure the communication of the system.

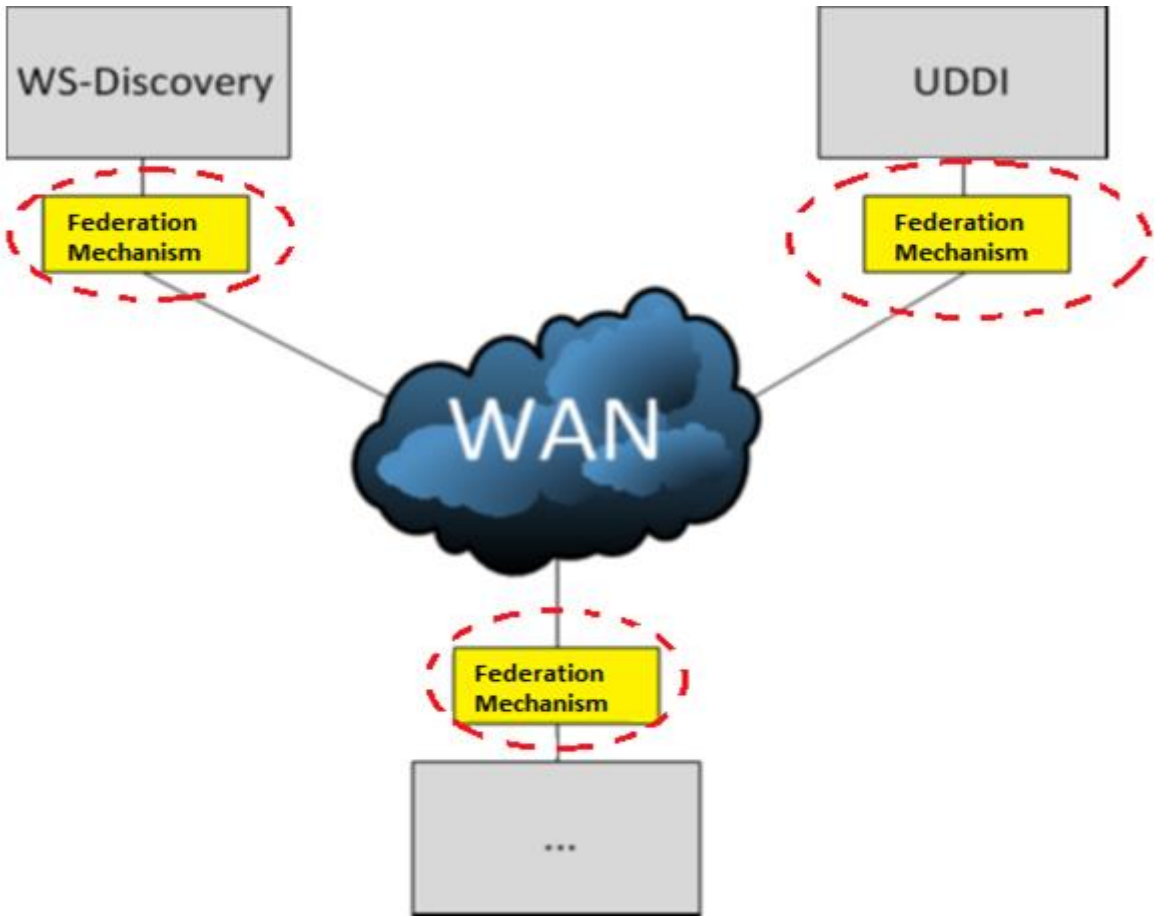

Figure 2. The focus of this task is to develop the federation mechanism that you can see inside the red circles, this is a utility enabling interconnection between different Web Service Discovery Mechanisms.

Realizing such a federation mechanism will require using both existing standards and libraries, as well as developing software that links these existing technologies together. In this chapter we give an overview of the existing standards and technologies that are relevant for the development of the federation mechanism. We will start with a more in-depth introduction to Web Service, with a particular focus on service discovery mechanisms.

2.2.2 SOA realized using Web Services.

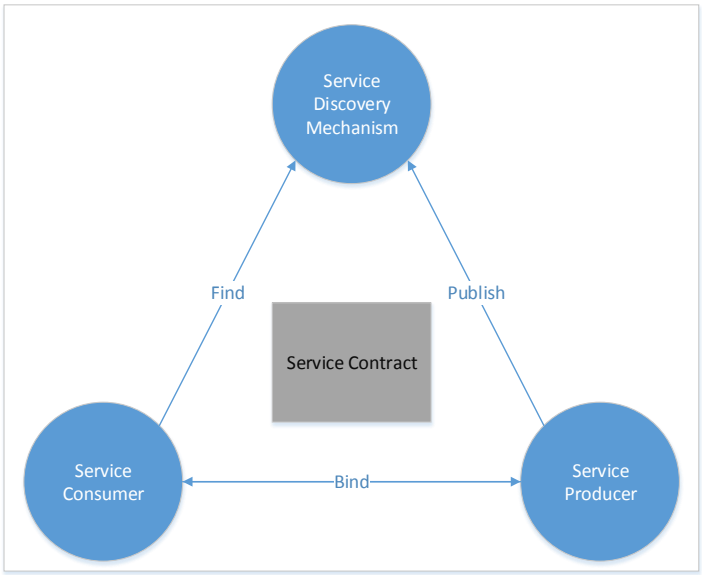

Figure 1. The SOA Triangle, with the service contract in the center [11].

When building a distributed system based on SOA principles, there are three main roles that must be realized by different entities, as illustrated in the so-called SOA triangle in Figure 4. The two main roles are the service producer and the service consumer.

The service producer is an entity that hosts one or more services that it offers to other users. The service producer also provides a service contract, which is a document that describes the interface of the Web Service provided. It says something about what input the service requires and in which format the response will be. This information is vital for developers to be able to create a consumer that can utilize the Web service.

A service consumer is entity that utilizes the service offered by a service producer. By sending a request to a service producer and then present the response of this request to the user or another application.

The last role in the SOA triangle is the Web Service Discovery Mechanism. This mechanism can either discover the services as they are deployed to the network, or allow the service to register with it when they deploy. A service consumer can then search for a service that matches some criteria, or just retrieve the entire list of services. This search can be done by searching for a service by port type, which is the attribute that contains information regarding the operations and messages the service uses. If you do not know exactly what service you are looking for, you can search for services by properties as input and return values. A machine can conduct the search process if you have an advanced matching algorithm, or if you know what service you are looking for, and you are more or less checking for its location. In many other cases, the best matching is done through human interaction, where the needed response is analyzed and decided by the developers and users of the service. This is on the other hand considerably more time consuming than a machine doing the task. The last is to have all available services to answer the request, regardless of their use or function. The user will then receive the whole list, the list will contain information about the services location,

input values, output values and workings. This is quite common to do on a regular basis in dynamic networks, in order to keep an updated list of the available services in the network.

The messages between the three parties in a SOA can be on many different formats, depending on which technology is used when building a distributed system. In Web Service-based systems this is done using SOAP [14]. Below we introduce XML, the data format used in SOAP, the SOAP protocol itself, and the service description language used in Web Services, known as WSDL.

2.2.1.1 eXtensible Markup Language (XML) [15]:

Extensible markup language, describes a serialization of data objects called XML documents, in order to be able to work with the objects as programming objects, you need to use a parser that can "translate" them to the language of your choice. XML is a widely used format for communication between computers, it was initially developed as a good base format to meet the requirements of large scale electronic publishing, but has evolved to be the most popular standard for data interchange. XML was invented as an easy way to markup and define variables within a document, enabling tags on each variable, describing the variable. When it was introduced, it was the first universal data focused display format, giving developers, customers and clients a common format, to which they could all adapt their technologies. At the time being, heavy, expensive middleware software was needed in order to enable interconnection of systems across different programming languages, system platforms and operating systems. With XML, all that had to be done, was to send the data and let the receiver do whatever they wanted to. Today you can find that many web page subscription or application programming interfaces will return a XML file, this can for example be found at www.bring.no and www.yr.no.

AS of December 2013, the number of XML compatible application programing interfaces(APIs) offered by the www.programmableweb.com API directory, is above 55% [16], which makes XML the most popular choice for online data interchange. Even though XML is losing shares to other simpler formats, it is still a very adaptable format that ranges from very simple to advanced representation of data. XML is also used in several major Internet languages, such as; XHTML, XML Schema, SVG, WSDL and RSS, some of which will be further explained through this thesis.

The main idea behind the XML language, that you can include any variable enclosed in a markup that describes the value it is containing. XML documents are text based which makes them easier for humans to read, one downside of this feature may be that it makes the documents verbose. Due to this, the amount of overhead in the document is very large compared to the amount of data, especially for documents with small amounts of data.
As a solution to this problem, several different initiatives that aimed to reduce the size and overhead, some at the cost of human readability, was initiated to speed up and streamline the data for optimal computer-to-computer communication.
One of these initiatives is the Efficient XML Interchange, EXI [17] language, which is a binary XML, that significantly reduces the amount of overhead, and computational cost involved in parsing and transferring the document. By using a more compact representation of XML, such as EXI, the size of the XML documents can be reduced by as much as 95% [18].

2.2.1.2 SOAP messages [19]:

The communication between different parties in a Web Services consists of SOAP messages. SOAP is a well-known message structure, known for its adaptability and interoperability. SOAP messages are also supported by many other standards, such as WS-Security. WS-Security is included in the Web Service Specifications [2] made by OASIS, it enables encryption and digital signatures of the SOAP messages. The standardized structure of the messages makes them easy to transfer between several parties, even though the parties do not have any prior knowledge of each other. A SOAP message consists of four main parts; the envelope is the wrapper that contains the other three elements. The main part of the message is the header and the body, where the header contains request and response information, and the body contains the data being transferred inside the message, as well as the optional Fault element, this element contains errors that may have occurred when processing the message. The envelope and the body element are also required, but the header is optional. On the other hand, SOAP standards are known to carry a lot of overhead, and the language is quite verbose, which makes the messages a lot bigger than they have to be [18].

2.2.1.3 Web Service Description Language [20]:

WSDL is an XML format for describing network services (Web Services) as a set of endpoints that communicates using messages, these messages can be either document- oriented or procedure-oriented. A document-oriented message is formed as an XML document and can be read as an XML document. A procedure-oriented message is a message that is parsed into an object containing different variables and their values. A common use of procedure-oriented messages is when you marshall [21] a java object into a SOAP message, before you send it, and unmarshall it at the receiver side. The operations and messages are described abstractly, and then bound to a concrete transport protocol and message format to define an endpoint. WSDL is extensible to allow description of endpoints and their messages regardless of what message format or network protocol.

The main function of the WSDL document is to describe the service and communication a Web Service provides. It contains information about how the service should be called, what parameters it expects, and what return values or types it gives the client. The WSDL document is machine-readable; this makes the implementation of clients and services easier.

The WSDL holds several elements, Table 1 contains a list of variables fetched from the W3C document describing the WSDL elements [20].

Variable name	Description
Service	Contains a set of services that has been exposed to the web through the service.

Port/Endpoint	Defines the address of the service, usually in the form of a URL or similar.
Binding	Specifies the PortType/Interface and defines the SOAP style used by the service. There are four different SOAP styles, they have minor differences with regard to setup and terminology.
PortType/Interface	Defines the service and the operations that the service offers.
Operation	Contains information about how the SOAP messages are encoded.
Message	Usually a message that corresponds to the operation field.
Types	Describes the data, this is done using XML.

Table 1 The different fields of the WSDL document.

The WSDL consists of two parts, an abstract part and a concrete part. The abstract part contains information about the interface, the output and input values and so on, this information will be the same wherever the service is deployed. The abstract part consists of four elements: Types, Message, Operation and PortType. The abstract part is utilized when a developer is implementing a client, the abstract part can also be used as a foundation for creating a service. This is because the abstract part contains information about what input and return values the application accepts.

The concrete part contains information that is specific to the implementation and where it is located, this part is used for the service invocation and is needed as a final step before the client can call the service. The part is made up from the Binding, Services and Port elements listed above.

2.2.3 Service discovery in-depth

Web Service Discovery protocols are the basis for this thesis . In order to be able to connect these protocols over long distances, a thorough understanding of their mission, and their way of operation is necessary when creating the federation mechanism that is the goal of this thesis.

There exists numerous different Web Service Discovery protocols, they all strive to connect Web Service providers with Web Service Consumers. There are multiple different approaches to categorize these. The following section will start out by explaining a few different modes of operation, before the most relevant alternatives are highlighted and discussed. The different types of categories are: Design time versus Runtime Discovery, Centralized versus distributed infrastructure, as well as mode of operation and Service richness will be discussed in the next sections.

The main idea behind using Web Service Discovery Mechanisms is the ability to make services available for all interested parties within a network, and enable invocation of the provided services in the easiest and most efficient way possible [22]. There are two types of service discovery, design time discovery, and runtime discovery.

Design time discovery is a process where you know the service you are going to invoke at the time you are developing the client, this enables you to tailor the client to fit the specifications of the service, thus making it less flexible for use with other services. However, the connection between the provider and consumer is still considered a loose coupling. If a Web Service Discovery protocol is going to support the creation of design time service clients, it must also offer the entire WSDL to any potential consumers.

Runtime discovery enables loose coupling and late binding, this coupling can be defined as even looser than the one from design time discovery. This feature enables the consumer to be designed for any service matching the criteria of the service consumer. The consumer is not aware of the provider until it discovers it in the network, if done properly, the only information the consumer needs is the endpoint address to the provider, in order to invoke the service. When designing a service for runtime discovery, you need the abstract part, and when invoking the service you need the concrete part of the WSDL.

A centralized Web Service Discovery protocol is designed to work as a database or registry for different Web Service Providers. Whenever a service is deployed and ready for invocation, the service is also registered with the Service Discovery mechanism. This allows anyone looking for services to browse the registry to see if they can find a service that suits their needs. A centralized registry is the most basic type of Web Service Discovery Mechanism to set up, as each node in the network only have to communicate with one other node. It is also easy to administer, as all the information available in the network is centralized and easy to monitor. The centralized solution also have some drawbacks that needs to be considered, the most major one being robustness. As all information is stored in a single location, you get a single point of failure. This included with issues involving scalability of data replication [23], are important drawbacks to take notice of when considering a centralized solution.

The counterpart of a centralized Web Service Discovery Mechanism is a distributed one. Instead of keeping a centralized registry that contains information about all of the different services, a network of nodes are responsible for keeping the information, whilst the rest of the nodes just pull and push information from and to these centralized nodes. There are varying levels of distribution, from two centralized nodes, to every node being a part of the centralized network, this can be configured as one pleases. The distributed topologies can be divided into three different subcategories [24]: clusters, distributed hash tables and multiregistries. Clusters are basically a centralized mechanism that is duplicated on multiple nodes. Distributed hash tables are based on storing hash tables with

information on intermediate nodes, which in total makes up the entire registry. Mulitregistries are several individual registries with different content.

The last important feature of the different Web Service Discovery protocols is their service richness. This is how much information each Web Service Discovery protocol stores about each service. Some protocol stores a minimum of information, such as name, description and address, while other can contain thorough information about the provider, the service, the intended use and so on. This is an interesting topic because a part of this thesis goal is to create a simple, unified way to represent a Web Service.

2.2.4 Service discovery standards.

There are several standards involved in service discovery, the aim is to take a closer look at each of the standards and review their workings and evaluate their relevancy for this project. For Web services, there are three standards – all by OASIS : UDDI, WS-Discovery and ebXML. There also exist a lot of non-Web service specific protocols [25]. Of these, mDNS-SD is important because it has been suggested for military use by the TIDE community. Hence, the most pertinent protocols at this point are:

- **UDDI –** Universal Description Discovery and integration – a standard for Web Services registry
- **WS – Discovery –** a standard for mainly local Web Services discovery.
- **ebXML –** Electronic Business using eXtensible Markup Language – another registry standard.
- **mDNS-SD** (also known as Zeroconf/Bonjour)

2.2.3.1 UDDI [26]

Universal Description Discovery and Integration(UDDI) is a platform independent, XML based registry by which businesses worldwide can list themselves on the Internet and a mechanism to register and locate Web Services. UDDI was proposed in August 2000, the vision behind UDDI was to make all Web Service producers able to publish their services online, and anyone who needed a service could find a service or several services that matched their criteria, and then invoke the service they had found. UDDI is included in the OASIS Web Service interoperability (WS-I) basic profile, as a major contributor to the Web Service infrastructure.

In later years, UDDI has lost some of its popularity to other discovery mechanisms. The work on UDDI was completed and closed late 2007, and there is no longer anyone responsible for maintaining the UDDI registry However, NATO still recommends UDDI as a standard Service Discovery Mechanism, due to support for design time discovery.

The UDDI systems in use today are most commonly used inside companies, where they are used to dynamically bind client systems to service implementations.

One of UDDIs biggest drawbacks for use in tactical distributed network solutions is the fact that UDDI is designed as a centralized repository solution, while a tactical distributed network is often very unstable and unreliable. This can affect the scalability and the impact of availability problems would probably be quite noticeable [11].

2.2.3.2 WS – Discovery [24]

In the OASIS Web Services Dynamic Discovery V 1.1 [27] standard we can read:

"The primary scenario for discovery is a client searching for one or more target services. The protocol defines two modes of operation, an ad hoc mode and a managed mode. In an ad hoc mode, to find a target service by the type of the target service, a scope in which the target service resides, or both, a client sends a probe message to a multicast group; target services that match the probe send a response directly to the client. To locate a target service by name, a client sends a resolution request message to the same multicast group, and again, the target service that matches sends a response directly to the client. »

WS-Discovery is a discovery protocol to locate services, it has two main modes of operation, Ad hoc mode and managed mode. The main use and intent for this protocol is Web Service clients searching for one or more target services. In ad-hoc mode the client can search for a service by searching for the type of target service it needs or by searching a scope of target services. In addition to these two modes, there exists an option called discovery proxy, this can be implemented in both modes, but will have a slightly different function depending on which mode it implements. WS-Discovery is a hybrid protocol.

Discovery proxy is an optional feature of ad-hoc mode, but a mandatory feature of managed mode. The reason for implementing discovery proxy is to increase the scalability of the system, and to increase the reach of the services beyond the local or ad-hoc network. The discovery proxy works as a hub between the service providers and consumers. When discovery proxy is enabled in ad-hoc mode, it intercepts the Hello multicast messages going from the client and into the network. The discovery proxy then responds to these messages with a unicast message, making the client aware that a discovery proxy exists in the network, and that all future requests and messages may be sent unicast to the discovery proxy, rather than multicast to the entire network. This behavior is called multicast suppression, but the clients are not obliged to honor this solicitation. They can choose to ignore this keep going as they used to. This way the amount of overhead network traffic is greatly reduced.

A discovery proxy is a standard feature of the WS-discovery when in managed mode, the main function is the same, but there are some slight changes to the process. The most important being that the client and the service knows upon the time of connecting to the network, that there is a managed discovery proxy present. Based on this knowledge the client and service will send their Hello and Probe messages to this discovery proxy unicast, rather than flooding the network with this information. The result is less network overhead and a reduced number of messages to discover and establish a connection between a service and a client. A network may also consist of several discovery proxies, where each proxy is responsible for its one hop or local network neighbors. The discovery proxies is also responsible for interchanging the information inter domain, to other discovery proxies on other domains. One domain may also include more than one discovery proxy for redundancy and increased reliability.

Due to the dynamic and unreliable nature of MANETS and ad-hoc networks, the clients may be set up to dynamically switch their mode of operation from ad-hoc mode to managed mode if this is feasible.

The main intent of implementing Managed mode in favor of ad-hoc mode is to increase the scalability of the solution and facilitate it for use in larger networks with a higher number of clients and consumers. It also enables you to share Web Services across multiple networks, which greatly increases the reach of your Web Service. This is achieved using a discovery proxy, which is an endpoint for both clients and services. When the discovery proxy detects such a message, it will "respond" to this message by broadcasting its own presence and the clients and services that receives this broadcast, will start sending unicast messages to the discovery proxy, hence greatly reducing the network load. The clients will now keep themselves updated through querying the discovery proxy regularly, to check for any updates or unresponsive proxies. If the discovery proxy does not respond to the client messages, it will switch back to ad hoc mode.

Nevertheless, in spite of its many upsides, there are some drawbacks when favoring managed mode over ad-hoc mode, the two main problems is liveness and availability problems. These problems are related. Liveness is where the service endpoint is up and running, but can not be found because the service registry containing its address is offline. Availability is when the registry lists the service endpoint as online, but for whatever reason the endpoint can not be reached.

Network based defense (NBD) is also an important aspect of the challenges one may face when implementing your system for use with larger networks and easier accessibility. The more open and

2.2.3.3 ebXML [28]

Electronic Business using eXtensible Markup Language, commonly known as e-business XML or ebXML. ebXML is most known for its business message transfer implementation, however ebXML have also developed a registry that can be used to store information about available Web Services in a network, and provide Web Service consumers in the network with information about these services.

The general workings of the ebXML registry is mostly the same as its relatives, UDDI and WS-Discovery Discovery Proxy. They all enable different entities to discover each other, exchange messages and participate in mutually beneficial collaborations. ebXML is set up mainly for content-based listings, which enables the clients to search for services based by their contents, or if they know the service they are going to use, they can search by name. . In NATO ebXML has been chosen as the metadata registry [29] and not for service discovery. Hence, we do not pursue ebXML further for service discovery in this thesis.

2.2.3.4 mDNS-SD [30]

Multicast Domain Name System – Service Discovery is a protocol used in smaller networks where no name server is present. It is a zeroconf service, hence it should be able to find and connect to other nodes running mDNS-SD in the network, without any additional information than the information given at the time of discovery. It was developed with ease of use, easy setup and adaptability in dynamic environments as three important key goals for the system. In short, the system works like this: When an mDNS node connects to a network it sends DNS packets over UDP to a multicast address, all mDNS capable hosts in the network is listening at this address and responding to the DNS messages. The process of service discovery using mDNS is done in two steps, the first step gives you the name of the host, whilst the next step gives you the IP-address of the host, this is because the IP can change, but the name will remain the same. The mDNS protocol is designed to keep overhead

and network chatter at a minimum, this is done by few messages and extensive caching. The services will then be listed in the local domain name system, exposing them for anyone that might want to use them. The TIDE community has suggested using mDNS-SD in military networks. That suggestion has so far gained little momentum, and TIDE has recently started focusing on WS-Discovery as an alternative.

2.2.3.5 Service discovery mechanisms summarized

Mechanism	Discovery type	Topology	Mode of operation
UDDI	Design-time and run-time	Centralized (federation possible)	Proactive
ebXML	Design-time and run-time	Centralized (federation possible)	Proactive
WS-Discovery	Run-time	Distributed	Hybrid
mDNS-SD	Run-time	Distributed	Reactive

Table 2 – Comparison chart for different Web Service Discovery Mechanisms.

Table 2 summarizes the main characteristics for the different types of Web Service Discovery Mechanisms. Whether it is a reactive or a proactive mechanism is not a very important feature. Regardless if it is a reactive, proactive or hybrid mechanism, the environment it is going to operate in for this thesis will be able to handle it. If it at some stage, e.g. the environment would change, this decision may have to be redone. All the listed service discovery mechanisms have openly available java implementations that can be used as a part of the project.

The topology of the different mechanisms are a much more important issue. A fully centralized mechanism will leave the system very vulnerable as it will introduce a single point of failure. Another question is also who is going to be responsible for drifting this centralized registry, as it may require a lot of resources to do so. In order to maintain a decent level of scalability and redundancy, a Centralized solution where federation is possible, or a distributed solution should be chosen for this project.

The discovery type is also an important aspect of the different mechanisms. If the connected mechanism is going to support design time discovery or not, is very dependent on the situation. One advantage of offering design time discovery support is that is increases the reach of the system, making it more versatile and agile. On the other hand, a design time discovery mechanism is simpler and easier to implement, which is also a good feature. For ultimate flexibility both types of discovery protocol need to be supported by the federation mechanism. As is evident from the discussion of protocols above, the two most important ones in NATO at the moment are UDDI (i.e., for FMN) and WS-Discovery (suggested by IST-118 and TIDE). Hence, both these protocols will be used in this thesis. Next, we survey WAN information dissemination mechanisms (hereafter referred to as “WAN mechanisms”).

2.2.5 Communication methods

The other third party component this application will rely on, aside from the Web Service Discovery Mechanism, is a WAN mechanism. The WAN mechanism will be responsible for transferring the data between the different instances of the application. The WAN mechanism will be attached to the application through an interface.

There are several fundamentally different approaches to data transfer, Broadcast, multicast, unicast and peer to peer are some of the most significant solutions. Each of these technologies represent at different way to transfer data from one or many nodes to one or many other nodes, they were all designed with different purposes and applications in mind, making them good at some features, and inadequate at others.

The main idea of each of these solutions predates all technology we have today, although their naming and standardization may have come with televisions, computers or telephones.

Broadcasting is an old concept where a message, signal or other content is sent from one broadcaster to multiple receivers, referred to as one to all in non-technical language. The signal is sent on a specific frequency, channel or IP and anyone who wants to listen to or receive this signal will only have to listen to the signal. This can be compared to a person giving a speech, where whoever interested in listening just have to get in range to hear the person speak. In typical broadcasting mediums, such as television or radio networks, broadcasting is cheap, reliable and scalable. However, in the Internet it requires more resources, mitigating the big advantage broadcast has as an internet medium compared to as a television or radio network. The effort required for a sender to reach ten nodes is the same as the one needed to reach a million, making this a very powerful way of transmitting information. Broadcasting is also the concept behind television broadcasting. However, there is no control over who or how many nodes that is receiving the signals, and the communication can be difficult to secure.

Multicasting is a similar concept to broadcasting, but it is done in a slightly different way. There are two ways of utilizing multicast in a computer network, IP multicast and application multicast [31]. IP multicast is where each node and router in the network will listen to a multicast address for updates. However, this presupposes that the network supports multicast, small or medium networks mostly do this, but larger networks e.g. the Internet rarely supports multicast. If you want to utilize multicast in an network that does not support IP multicast, application layer multicast can be used. Application layer multicast creates a hub on top of the network, which then allows user to subscribe to multicasts going through this hub. Multicasting is also based upon a one to many foundation, or even many to many. However, the receivers that wants to listen to the multicaster, must register with the multicast in order to receive the transmissions. This reduces scalability, as there are only a finite number of available channels, as well as administering access to the different groups can be a tiresome task if there are a large number of users. On the other hand, it enables the transmitter to know how many nodes that are receiving the messages, and it is easier to enforce message encryption or integrity checks.

Unicast is the opposite of broadcasting, it is the act of sending the data to only one specific receiver, often identified by an unique address. This is not a good communication mode for message to many recipients, but it does have better support for Quality of Service (QOS), where the sender can for example be offered an acknowledgement of the message being delivered to the receiver. Data integrity and security is also easier to handle than when using multicast. Unicast is mainly suitable for messages between few nodes, as it does not scale very well.

Peer-to-Peer is a nonhierarchical system, where all nodes connected to the system have the same authority as everyone else. There is no predefined sender or receiver, which makes the system very redundant, as all nodes is of equal significance. When a node requests a resource, the resource can be sent through any available path, from any node that possesses the resource, the resource can also be fetched piece by piece from different nodes. This ensures a very high redundancy, since if a node goes down, the transmission can continue through or from another node.

The requirements for the WAN mechanism depends on the environment the application is going to be used in. If the application is going to communicate to other instances through a MANET or disadvantaged grid, the WAN mechanism should be optimized for these conditions. If the network between the nodes is a high speed Internet connection, speed and reliability are more important aspects. Another important aspect to take into account, is the number of instances and the number of services that will be hosted on each instance. A system may perform very well with 2-3 instances, each holding 5-10 services, the amount of traffic will be fairly low and easy for the WAN mechanism to handle, but if you increase these numbers by ten or a hundred, it may not work at all.

For this application, it is most likely that the communication between the different instances of the application will be communicating a Internet like Wan, which may be a closed network created for the mission, but with the same features as the Internet. This Internet may not always be a wired network, it may also be a wireless network. Using these prerequisites as a foundation a few key requirements for the WAN mechanism is:

- An existing, available implementation must exist.
- This implementation must start in the application layer and end up in the application layer, what the data does between these two points is not of great importance.
- The implementation must scale well, as the numbers of users and number of traffic can grow very fast
- The implementation must be easy to implement and to get started with.
- The implementation must to some extent support some kind of QOS mechanism, that allows for error checking and reliable transmission.

2.2.6 WAN Mechanisms

There exists several options when it comes to suitable WAN mechanisms that could match one or more of these criteria. In order to find the best option, a comparison of the different options and the features they offer will be conducted. A score will be given based on the number demands that are met.

The different alternatives at the time being is; Pastry, Tapestry, Gnutella, KadScribe, AMQP and Mist [32] [33] [34] [35] [36] [37]. These have been selected as possible candidates based on input from other researchers at the field.

2.2.6.1 Pastry [38]:

Pastry is an overlaying implementation of a Distributed Hash Table(DHT) [39]. A DHT is a key value pair table where the key is a hash of the value, a hash is a transformation of data into a fixed length variable, the output of a hash function is the same every time the for the same value. When a system is distributed it means that it is not centralized, but spread across multiple nodes It is based on a peer-to-peer network model where each node is equally important as the other. The implementation

features self-repairing hash tables, that automatically updates as there are changes in the network. Since it is a peer-to-peer application, it is very redundant and fail safe. The protocol can also use externally supplied routing metrics, to calculate the best route. It has been tested in a simulated environment, counting more than 100 000 nodes, with good results regarding resilience to failure and scalability. There exists an open, available, well-tested implementation of Pastry, called FreePastry [40].

2.2.6.2 Tapestry [41]:

Tapestry is also a peer-to-peer overlay network like Pastry, but it also implements multicasting and increased scalability through location aware routing where the nodes knows its neighbors and the optimal path to them. It also utilizes self-repairing hash tables like Pastry. It is also a decentralized structure, where there is no central hub, but all nodes are equally important. The nodes are location independent, and it has shown to be resilient under high load. It is self-administering and utilizes randomness in the way it distributes load and choses paths for its data. There exists an implementation of a network storage application called OpenStore where Tapestry is utilized, but this is not very well suited for the needs of the application this thesis is describing. This is because OpenStore is implemented as a federated storage mechanism, and this application needs a simple network messaging client.

2.2.6.3 Gnutella [42]:

Gnutella is a distributed search protocol, implemented using peer-to-peer methodology. It is similar to the two previous alternatives, except for that the main intent of Gnutella is to be a search protocol. It can be used to transfer data, but the main focus is towards optimizing network searching through a peer-to-peer network. All data transfer done through Gnutella is done by invoking a small server/client architecture for each node, where the connection is established through the p2p packages, but the transfer itself is done through a GET request on port 8080. Research also points out that the amount of data held by each node can impact the performance of the network, especially in larger networks. At the time being there are no available implementations of Gnutella that suits the needs of this application, as the only openly available implementation Gtk-Gnutella [43] is written in C, which makes it very hard to incorporate as a part of this application.

2.2.6.4 KadScribe [44]:

The purpose of KadScribe is to enable a subscription-based message dissemination mechanism for a large number of participants. The mechanism is intended as a building block for other protocols and applications. It is based on a merger between Kademlia and Scribe, where Kademlia is responsible for the routing whilst Scribe handles the rest of the work. Scribe is built on top of Pastry, but in order to improve routing performance in disadvantaged grids the routing mechanism from Kademlia has been chosen. Possible applications include SOA messaging, weather information or an instant messaging presence service. The focus is on best-effort, low data rate services. The special challenges of disadvantaged networks such as volatile user behavior, low transmission capacity and faulty network connections are respected. Due to the fact that KadScribe is a Frauenhofer FKIE project, the implementation has not been published to the public, and no other available implementation can be found.

2.2.6.5 AMQP [45]:

Advanced Message Queuing Protocol(AMQP) is an open standard middleware. It is standardized by Oasis and is a supported and implemented by a number of different major contributors in the software and banking industry. AMQP is in short a direct messaging client, and can be compared to an instant messaging service or email. It differs by offering the user numerous options with regard to QOS, delivery time, security and so on. The backbone of the AMQP protocol is the broker, which works as a hub, connecting the different nodes, and allowing configuration options for the messages. A well tested, open source implementation of AMQP exists in ActiveMQ, which is released by Apache under the Apache 2.0 license, which offers very wide boundaries for anyone interested in using the software.

2.2.6.6 Mist [37]:

-*"Mist aims to provide efficient distribution of content in MANETs without relying on valid routes being provided by a routing protocol.* [37]*"*

Mist is a reliable and robust middleware, well suited for publish/subscribe data transfer between nodes in a dynamic network. Mist is a proprietary and experimental protocol, which is still in beta version. Mist is designed for message delivery in fully connected mesh networks, tests and emulations show that it performs well in these types of networks [37]. This makes the protocol well suited for use in mobile ad hoc networks (MANETS), where the topology is highly dynamic and it is likely to get a high error rate.

2.2.6.7 WAN mechanism summary

	Pastry	*Tapestry*	*Gnutella*	*KadScribe*	*AMQP*	*Mist*
Available implementation	1	0	0	0	1	1
Application layer	1	1	1	1	1	1
Scalable	1	1	1	0	1	0
Easy startup/ implementation	0	1	1	1	1	1
QOS Compliant	1	1	1	0	1	1
SUM	4	4	4	2	5	4

Table 3 WAN Mechanism Evaluation

As you can see from Table 3, the different alternatives has been listed and evaluated against the prerequisites defined for a sufficient WAN mechanism. Even though the scores for many of the alternatives are equal, the different criteria is weighted differently. Any alternative that does not have an available, open implementation will be discarded, as implementing a wan mechanism is outside the scope of this task. Having said that, that leaves us with three options, Mist, AMQP and Pastry.

Mist is available as an open source, freely available Java implementation, making it very easy to implement as a part of the project and get it up and running, fast. Mist also support both UPD and

TCP network transfer, this enables the user to select either speed or reliability, which is a good feature.

However, there is a drawback using Mist. Mist is originally created for use in MANETS and other disadvantaged grids, making it good on supporting QOS and reliability, but not so good at scalability and high speed transfer. This is an important point, and should be kept in mind when deploying the application. As a preliminary solution however, Mist will suffice and do the job it is supposed to do, without any hassle. If the system is to be deployed in larger environments with a greater amount of users, changing the protocol to something more suited could be advantageous.

AMQP, through the implementation of ActiveMQ is a very good alternative as it is easily accessible, well tested and thoroughly documented. The default setup that AMQP ships with delivers good performance. One very big advantage of ActiveMQ over Mist is that ActiveMQ features direct connection between the nodes, where Mist requires multicast support, rendering Mist useless in networks that does not support multicast, IE. the Internet.

Pastry is also a good alternative to use as a WAN mechanism for the application. It achieved good scores on the WAN mechanism comparison test, and meets most the criteria set.

The application will be designed with a standard interface that allows for different WAN mechanisms to be connected to the system. This approach enables the user to select and adapt a WAN mechanism that is well suited for their needs and demands.

Looking at the available options when it comes to WAN mechanism, the initial WAN mechanism for the application will be Mist, due to it fulfilling the set demands, as well as being the easiest one of the alternatives to implement. However, in order to test and prove the modularity of the system in practice, ActiveMQ will be implemented when the application reaches large scale testing phase. This is due to the fact that the large scale tests will be conducted using the Internet as communications channel, and the Internet does not support multicast, which is required for Mist to work. This will also test the requirement for modularity, as a module will have to be replaced.

2.2.7 Response times

The field of web and application performance response times is a well researched and documented one. It has been an applicable measure of user friendliness for as long as there has been computer applications, and it still is. Jakob Nielsens book, Usability Engineering [46], has a very good section about this topic, called Response Times: The 3 important limits [47]. In short he dictates three time limits for application response times, and the user's reaction to these. The three limits are 0.1, 1.0 and 10 seconds. At 0.1 seconds, the user will not experience any latency and the action is perceived as immediate, no other action is required to keep the users attention. At 1.0 seconds, the user will notice the delay, but due to the short wait, the user's chain of thought will stay uninterrupted and the user will remain focused. The feeling of working directly with the data will not be maintained. The last benchmark is the 10 second one. An action can take up to 10 seconds before the user will lose interest in the dialogue of the application, and would like to do something else while waiting. If a process is expected to take more than 10 seconds, a measure to inform the user of the progress should be implemented.

2.3 Related Work

In order to achieve good results from this project it is crucial to have thorough knowledge to similar work. Researching previous related work can save a lot of time by learning from it, knowing where to focus the research and effort. Therefore, the related work section is an important section where I will review related work, and try to draw parallels from this work to this thesis. I will also look at how the related work will affect future work and related decisions.

There have been conducted other case studies on similar problems, but there are many slight alterations that makes each study unique. However, many of the other studies covers some areas of this system, creating a very good basis for comparisons, and a reference point from which to start researching.

The WS-Discovery case study [32] focuses on some of the challenges in this task, the main goal of the experiments conducted is to connect two local dynamic networks both containing local Web Service providers to each other through a WAN. They set up two local networks, each containing dummy sensors that provide the information they collect as a Web Service provider, sending the data out into the network. The two networks are then connected using P2P technology. The specific technology used was Pastry [49] coupled with SCRIBE [50]; this made the Web Services discoverable in both the local domain and the remote domain.
The difference between the two is best described by an illustration:

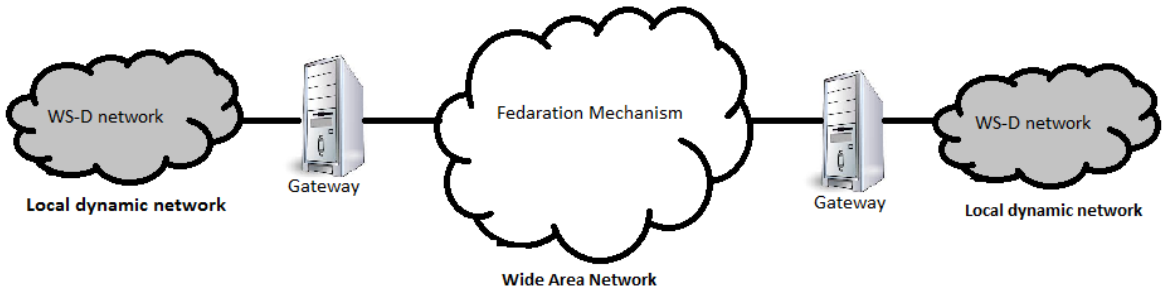

Figure 5. Federation of two independent WS-Discovery (WS-D) domains [32].

Figure 5 is from the WS-Discovery case study, and illustrates how the two local dynamic networks are connected through their gateways and over the Wide Area Network. The study addresses several issues that are very relevant for both the study and the thesis.

In chapter five of the WS-Discovery case study, conclusion and future work is discussed. The conclusion states that they are overall satisfied with the progress and outcome of the testing. Some shortcomings that should be rectified for the system to work as intended in a live deployment environment are identified. The list they created is as follows:

- There is no redundancy, only one instance of the WAN discovery mechanism can run in a subnet at any time.

- Currently, all services are shared across the networks. A selection mechanism needs to be present, for example configurable filtering using the services' scope.
- Freepastry was chosen for the P2P network because it was freely available. Other, better alternatives to extend the reach of WS-Discovery can possibly be found.
- The demonstrator only supports one service discovery mechanism. Ideally, it should support other relevant protocols as well in a seamless manner.

The first point is somewhat solved through the use of repositories. Having one repository per instance ensures that each instance is responsible for the information it holds. If a repository for any reason is to go down, the other repositories will still be able to maintain contact and their list of local and external services will not be affected. Nevertheless, liveness may be an issue with this setup, if a repository fails, the other repositories will have to find out, in order to remove the services offered by the failed repository/application instance. But the repository is only as redundant as the WAN mechanism connecting the different repositories, if the WAN mechanism is not redundant, neither is the system. A more thorough assessment of this issue and other related problems can be found in chapter 3.

The next point is regarding sharing of service, in a military context you may not always want to share all of the services that you are hosting, therefore an option to select what information you share with the other instances of the application is set as a requirement for the application. Through the use of a GUI, the user can browse a list of all available local and external services, and the select which of the local services that should be shared and made available for other nodes, and which should be kept private. A complete description of the design and features of the GUI can be found in chapter 3.

The third issue on the list is related to the communications mechanism, the WS-Discovery case study is mentioning issues regarding their use of Freepastry, and how it reduces the reach of the inter-gateway communications. Due to this point Mist and ActiveMQ has been chosen as communications mechanisms for the application this thesis addresses.

The last point from the conclusion of the WS-Discovery case study is related to the Web Service Discovery Mechanism used. In the WS-Discovery case study, the application only supported one distinct Web Service Discovery Mechanism, this may be sufficient in a test environment, but in a real life deployment, implementing support for multiple Web Service Discovery Mechanisms is crucial. This will not be a problem in the system this thesis describes, as the service discovery interface enables easy swapping of the connected Web Service Discovery Mechanism, with very little effort.

Through the use of this report and its conclusions, it is easier to identify the areas that needs the most attention, and what issues to be most cautious of. This will save a lot of time, and improve the finished product.

The focus of the thesis is therefore directed towards facilitating for further development on the area, where the goal is to make a federation mechanism. The goal for this mechanism is to enable interconnection of different Web Service discovery standards and facilitate for these to communicate to each other. The task also focuses on the importance of modularity and expandability, making it easier for other developers to continue the work on the project in any direction they see fit. The main idea behind this approach is to create a foundation through this project, which can be used as starting point for future projects. To solicit future use of the project, it is also important that there

are clearly defined interfaces that make it easy and simple for people to expand the reach of the project and its use. There will be no definitive correct answer to this task, the scope of the given task is very wide and there are almost infinite amounts of possible solutions.

A very relevant study is the "**Pervasive Web Services Discovery and Invocation in Military Networks**" [51] case study and its report. It is a thorough analysis and implementation of different Web Service optimization techniques as well as a review of the most common Web Service discovery standards and how they perform in a military setting. The military setting focuses on the need for tight security and the performance in disadvantaged grids. The task is in two parts, one focusing on Web Service optimization, which is an important issue, but not so relevant for this report. The other part focuses on Web Service standards related to service discovery, which is very relevant for this task. Most relevant Web Service Discovery Mechanisms can be used in a military context, but are not suited for use in disadvantaged grids.

One of the issues that they address is the problem concerning service interface richness, where different Web Service Discovery Mechanisms utilizes different variables to describe the same services. This introduces the need for a filter and a storage function somewhere between the services as when the service is going from a mechanism with a richer dataset to a mechanism with a smaller dataset, the filter will have to remove the excess information. The intermediary storage is addressing the same problem, but in an opposite direction, when the service is coming from a simpler dataset the missing information will have to be added from somewhere. However, the latter process requires human interaction to be able to work as intended. AS their focus is to look at more automated system, the rest of the discussion is discarded with regard to the scope of the article. However, it is an interesting issue for this thesis, as it is something I will be forced to take a stand on the issue. A more thorough review of this issue can be found in the design and implementation section.

They are also evaluating centralized registry models for use in MANETS, they conclude that registries are not very well fit for use in unreliable, disadvantaged grids. They also conclude that approaches and systems that has been known to perform well in stable, reliable networks can turn out to become counterproductive in MANETS, even though this is not at the very center of this tasks scope, it is an important thing to take notice of and keep in mind when designing and developing the system.
The most relevant and interesting section in the thesis is chapter 6. The chapter is about pervasive service discovery and how to best solve it. They look at a few different approaches, and evaluate these based on their specifications and features. The main criteria for the evaluation is to enable communication and interoperability between different types of service discovery mechanisms, located in separate domains or networks.

The three main approaches for achieving pervasive service discovery across multiple domains are: Adaptive service discovery, Layered Service discovery and Service discovery gateways.

Adaptive service discovery is where you utilize one Web Service Discovery Mechanins in and between all domains. All applications residing in the network must be able to interact with the

protocol. The protocol will have to be customized for each specific network, in order to ensure optimal function. When the information is going to or through a network with reduced capacity the amount of data should be reduced to adapt to the low capacity network, a filter would be needed for this task.

Layered service discovery is where each network can utilize the protocol that best suits the given network, but all networks will have to be connected using an overlying protocol that receives and pushes data to all the connected protocols.

Service discovery gateways can be viewed as an intermediary of the previous two. Each network will utilize the network protocol that is best suited for the network topology and capacity. The inter-domain communication is handled by the local network themselves, but all incoming and outgoing information will travel through a gateway which ensures that the data is on a standardized format and understandable for all other nodes. The gateway can also customize or adapt incoming data to match specific criteria of the local network, for example, a disadvantaged grid with low bandwidth and high error rates.

After evaluating all three options, comparing performance, scalability, reliability and ease of use and implementation, the conclusion was that the gateway alternative was very promising, and that it could do a great job with only minor adjustments and adaptations. The most important discovery to take from this is that there has to exist a repository inside the gateway that can store information when the services are transferred between different domains and discovery mechanisms.

Due to the discoveries in the previous related work, this thesis focuses on an approach where federation of information between gateways is essential. This allows the system to operate in several different instances, which can communicate to each other across the Internet. This is possible due to the use of gateways that offers a standardized interface for both incoming and outgoing data. This solution will facilitate a modular and easily scalable solution.

2.4 Requirements summary

Through the previous sections and chapters we have reviewed standards, frameworks, applications and other relevant technologies for this task. There has also been some discussions and assessments of the previously mentioned technology and its role in this project. Based in this information a list of system specifications has been crated and the importance of each specification requirement has been evaluated based on its impact on the finished product. can be seen below and contains three rows, a specification name, a short description and the level of importance for the given specification. The specifications in the table forms an important part of the task and the requirements to the finished product.

What	Description	Importance	Nr
Easily expandable	It should be easy to add new functionality and other frameworks to the system.	High	1.
Modularity	The system should consist of different modules that can easily be reused or replaced.	High	2.

Admin functions/settings	Settings regarding sharing of services across the WAN and other preferences.	High	3.
Well documented/easy to understand	The system should be easy to understand and it should be a trivial task for another developer to continue working on the project.	High	4.
XML messages across WAN	All messages going between the gateways through the WAN, should be formatted using XML. The XML should be based upon a schema, enabling other XML features to be utilized.	High	5.
Develop in accordance with SOA principles [48].	Strive to follow the SOA principles in the project, making it easier to maintain, adapt and develop for future use. Following the SOA principles may also increase robustness and performance of the application.	High	6.
Follow relevant standards, avoid proprietary solutions.	Try to take all relevant standards and as a part of the application, and avoid unnecessary proprietary solutions, which makes the code harder to use, expand and adapt.	Medium	7.
Ability to interconnect multiple instances of the application via a WAN mechanism.	One of the main purposes of the system is to enable sharing of Web Services across different domains and between multiple users. This is done through deploying one instance of the application per domain, and enable communication between these through a WAN mechanism. The WAN mechanism is therefore an essential part of the application.	High	8.
Ability to choose what to share	The ability to let the user decide which services to share and which not to share to the other users utilizing the application.	High	9.
Interchangeable WAN mechanism	Create the application so that the WAN mechanism used to communicate between each instance of the application can easily be replaced with any alternative the user may choose.	High	10.
Interchangeable WS-Discovery mechanism.	Create the application so that it can be used with different WS-Discovery mechanisms, which can be changed with very little effort.	High	11.
GUI	Implement a GUI that allows users to manually administer the application without any prior knowledge to the application	Medium	12.
Scalability	Create the application robust and scalable, making it able to handle multiple simultaneous instances, more than 25, preferably 50, with 5 nodes per instance.	High	13.
Possibility to add service information	A feature that allows the user to update local and remote services with additional information about that service	Medium	14.
Scale to NATO PfP-network size	The application should be able to be used as a part of a NATO Partinership for Peach network, which will consist of approxximatly 28-50 [52] [53]countries, each sharing their services.	High	15.

Low response times	The response times for the application should never exceed 10 seconds for a full update, as long as the number of nodes is not more than 50.	High	16.

Table 4 A summary of the requirements for the system.

3. Design and Implementation

In order to avoid delays in the development process, the design of the application attempts to cover every possible aspect and issue that may arise during implementation. In this chapter the design is presented. The Design is a result of the previous chapters and the contents therein, such as premises and requirements. The design is presented in different granularities, starting by breaking in down into coarse grained pieces explaining the overreaching functions and features of each component. Continuing from this each coarse grained piece into smaller, more fine grained pieces, as well as explaining the functions more thoroughly

When the overreaching presentation of the different components is completed, a set of sequence diagrams are presented to illustrate the information flow for a given scenario in the system.

3.1 Design

In order to avoid timely delays in the development process, the design of the application will try to cover every possible aspect and issue that may arise during implementation. In this chapter the design will be presented. The Design is a result of the previous chapters and the contents therein, such as premises and requirements. The design will be presented in different granularities, starting by breaking in down into coarse grained pieces explaining the overreaching functions and features of each component. Continuing from this each coarse grained piece into smaller and more fine grained pieces, explaining the functions more thoroughly and so on.

When the overreaching presentation of the different components is completed, a set of sequence diagrams are presented to illustrate the information flow for a given scenario in the system.

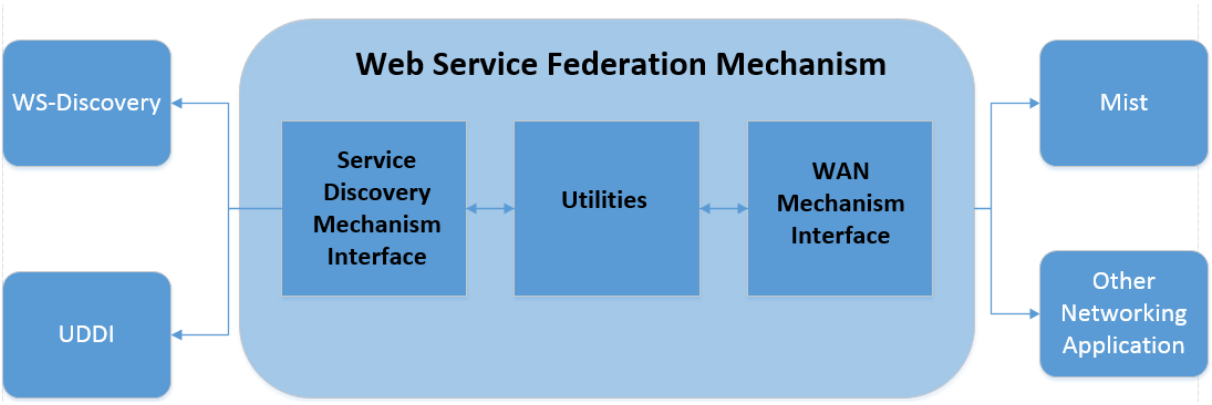

Figure 6. A schema depicting an overall view of the project design.

Figure 6 presents a high level overview of the system. The granularity is very coarse, and the only intent is to show the most basic concepts of the system. The focus here is on the components inside the “Web Service Federation Mechanism” box. The four, smaller boxes outside this box are examples of systems in which the federation mechanism is able to communicate with Even though there are two boxes outside on each side, the intent for the system is to be able to work with only one connected system per interface at a time. It supports several different external mechanisms, but only

one mechanism per interface at a time. This has been decided to ease the implementation work and due to the fact that there are very few scenarios where this feature would be necessary. This decision can be traced back to the system requirements, especially the requirement number 11. The loose design of the application is also a result of the requirements given, especially numbers: 1, 2 and 6.

In the center of the figure the three main components are. The two interfaces are designed to enable connection for different types of WAN mechanisms and Web Service Discovery Mechanisms. These interfaces expose methods that can be used in an implementation of a new WAN mechanism or Web Service Discovery Mechanism, making this an easy and predictable task. This is done to support the requirements: 2, 11 and 12.

In between the two interfaces there is a utilities module. This consists of the repository, as well as a few smaller classes that are responsible for handling the data in the system . The repository is not a part of the requirements, a more thorough review of this decision can be read in section 3.1.1 Design decisions. The specifics around all the modules inside the system will be further examined and assessed in their own sections below.

3.1.1 Design decisions

Through the work with this application several minor and major questions about the design have arisen, most of these questions are answered in their respective section. But the more general topics that does not fit into a specific section are discussed here.

It is not a given requirements that states that the application should consist of a central repository, but there are some very compelling reasons to include it. A central repository will in short contain information about all local and remote services, as well as the users own comments and notes for these services. The first and most important argument for using a repository instead of just saving services in the connected Web Service Discovery Mechanism is the difference in service representation. As seen in requirement nr 13, the system should be able to store additional or supplementary information about a service. If a service comes from a service discovery mechanism that stores more information about the service than the receiving Web Service Discovery Mechanism, the excess information would be lost. By using a repository this excess information can be stored in the repository. Another important argument is the fact that a repository will reduce the bandwidth consumption of the application. In order to reduce the amount of unnecessary data sent between different instances of the system, a repository could check all information it receives from both the connected Web Service Discovery Mechanism as well as the WAN mechanism against the information is has stored. This feature enables the system to compare new data and old data, and discard the new data if it is similar to the previous data stored. Requirement number **Error! Reference source not found.** is also easier to resolve with a repository design, as you need to store the decision of whether or not a service is to be shared somewhere. This cannot be stored in most Web Service Discovery Mechanisms, therefore it is easiest to store in the application.

Implementing a repository also has some drawbacks that should be mentioned. A repository means that the system will have to consist of more code. More code takes more time to create, and is harder to maintain. Omitting a repository would also force the application to implement a solid foundation for forwarding information acquired from other nodes to the connected Web Service Discovery Mechanism. Another challenge of implementing a repository, is the fact that the data from

all different Web Service Discovery Mechanism will have to be transformed into a common format that can hold the services.

Although there are reasons both for and against a repository, the benefits by having a repository by far outweighs the benefits by of not having one. Due to this, the system will be designed around a repository that can maintain the service information and state.

The decision to use gateways for communication between the nodes was made based upon previous research conducted in the field. The report [32] has researched very similar topics, making it a good foundation for decision making, you can read more about this report and the conclusion made therein in section: Related Work.

3.1.1.1 Service repository

Every service that is registered through either the WAN mechanism or the Web Service Discovery Mechanism is initially added to the repository, before the event is forwarded to other interested parties listening to repository changes. The repository is also a natural foundation for a system Graphical User Interface(GUI), as it holds all information and all settings available for the application. This enables the GUI to be designed to only interoperate with the repository, which helps creating clear lines between the different modules of the application.

The repository is made to hold information about all Web Services residing in the local network, as well as information about all services located in other instances of the application that is connected to the network. Using a repository ensures that all instances of the application are equally important peers, performing the same job and having the same responsibilities. This design is similar to the design used in peer to peer applications, as each peer is an equal to all other peers in the network. Peer to peer applications are known for their high level of redundancy, as well as very good scalability. This is due to that no node or peer has any elevated rights or responsibilities, which means that for the network to go down, every single node will have to go down. However, as mentioned earlier, the resilience is also greatly dependent on the WAN mechanism. Different WAN mechanisms are thoroughly examined in Appendix B, different techniques to mitigate the impact of WAN mechanism failures are also discussed there.

The communication between the different instances is going to be based on a notification messaging system, this means that all nodes will receive a notification when a change occurs, and then act accordingly. The main problem about this approach is if a node goes down, and the message containing this information is not sent. This will leave the other nodes thinking that all is fine, while the lost node is unreachable, even though it is listed as OK.

This leads us to some of the drawbacks emerging using distributed Web Service Discovery Mechanisms, most relevant are the liveness and availability problem as presented in chapter 2.2.3.2 WS – Discovery 6th section.

These problems can be mitigated using a fixed interval automatic response check, this check will send a ping message to all other nodes in the network, assuring that they are responding as they should. If a node does not respond, the services hosted by this node will be removed from all other nodes repositories. The value of this timer can be set through a configuration file, in a very stable environment the chance that a node goes down or disappears unexpectedly is low and the timer can

be set to long intervals. As well as vice versa in a more dynamic network. This solution will be the preferred solution for this system, as it facilitates reliability for the data in the system, as well as reducing the amount of overhead data created by the system.

3.1.1.1.1 Service repository richness

In the Java-WS-Discovery and UDDI section in Appendix B you read about the differences between WS-Discovery and UDDI. The repository is going to be universal for all connected service discovery mechanisms, meaning that the number, names and types of variables held within each Web Service listing will have to be adaptable and useable for all mechanisms following the current standards.

WebService variable name	WS-discovery name	WS-discovery datatype	UDDI name	UDDI datatype
ID	UUID	Integer	serviceKey	Integer
Description	Scope	List<URI>	Descrs	List<ServiceDescr>
Address	xAddress	List<String>	bindingTemplates	List<BindingTemplate>
Interface	portType	List<QName>	BindingTemplate	List<BindingTemplate>
businessKey	-	-	BusinessEntity	BusinessEntity
Name	-	-	Name	<List>ServiceNames
Parameters	-	-	-	-

Table 5 The different names and datatypes of the new WebService class, WS-Discovery and UDDI.

As you can see from Table 5, there are no standards or consistency in how the services are stored in the different Web Service Discovery Mechanisms. Due to this, the best solution for storing date from multiple different Web Service Discovery Mechanisms is to create a new Web Service class, which is compatible with the majority of the different Web Service Discovery Mechanisms datasets. It has been decided to solve this problem with a new class instead of using for example the dataset from UDDI. A separate class allows for more customization and adaption to fit the specific needs of this system. A good example of this, is mentioned in section 3.1.1 Design decisions and in system requirement nr **Error! Reference source not found.**. In order to keep track of what services are being shared, a “share” variable must be present in the service representation class, this in not available in any standardized Web Service Discovery library.

As the WebService class is going to be universal, supporting any type of Web Service Discovery Mechanism, it is required to support as much data as possible, a good documentation is also an important aspect as the translator class will work by assigning the right service variables to its corresponding WebService variable.

A more thorough review and explanation about the structure of the Web Service Class and the decisions made when designing it can be seen in section 3.1.2.2

3.1.2 General design

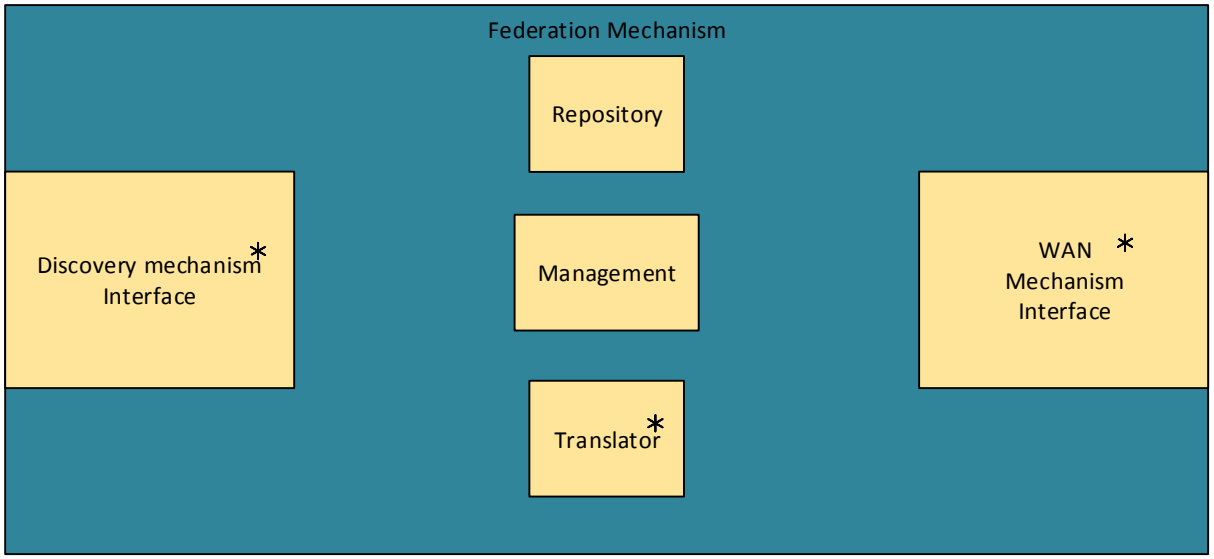

Figure 7. A general overview of the components in the system, the mechanism specific components are marked with a star.

The general design is made to offer the simplest and most efficient solution to the requirements given for the application. The first and most important requirements are requirements number one and two. They require the application to be easily expandable and modular. In order to achieve this a design where most functions are put in different modules has been chosen.

As you can see in in Figure 7, the system will consist of five different components. Some of the components will be the same for each instance of the mechanism, regardless of what Web Service Discovery Mechnism or WAN distribution mechanism that is connected. However, some components will have to be specifically made to each discovery mechanism or WAN mechanism. The Discovery mechanism interface, the WAN mechanism interface and the translator are all marked with a small star, this is to indicate that they are specific for the current setup. This is because they will have to serve as an intermediary connector between the application and the connected module. This is due to that there is no common, standardized way to represent and save Web Services. That is what this application aims to change, through creating a standardized representation of Web Services. WS-discovery and UDDI have different types of interfaces, value names and number of values for each service, hence the easiest and most modular way of implementing support for these standards is to create an interface that enables the users to create their own intermediary class.

In order to comply with requirements number 10 and 11, open interfaces that enables the user to connect any WAN mechanism or Web Service Discovery Mechanism to the system, is needed. These interfaces will ensure that there is a standardized way to send and receive information through the connected mechanism. This has been done by creating one interface in each end of the application, the interfaces and their intent is described below.

To the very left, we have the Discovery Mechanism interface. It is responsible for the incoming and outgoing communication to the connected Web Service Discovery Mechanism. As mentioned earlier, not all discovery mechanisms have the same richness and interface, hence this module needs to be different for each type of discovery mechanism.

To the very right, is the WAN mechanism interface. This interface serves the same purpose as the Discovery Mechanism Interface, but instead of connecting the system to a discovery mechanism, it transmits the data from the system to other instances of the system.

As mentioned in chapter 3.1.1 Design decisions, the system will feature a repository that allows the user to store the information and metadata about each service. The repository will also be related to the management module, but this has been distinguished into a separate module to facilitate running the system both with and without a GUI.

In the top center we have the repository, the repository is responsible for storing information about the services in the network. This storage function facilitates bandwidth optimization and custom flags that can be used in sharing context. A big design choice for the system is the structure for the data in the repository. Since the repository is to be able to store all available service discovery mechanism, it needs preset variables that fits the most widely used Web Service Discovery Mechanisms, as well as a metadata variable that can store unanticipated variables and values.

In the very center we have the management module. This module is responsible for the management of the system. The size and complexity of this module is not a given. The most basic functionality being the ability to choose what services you share across the network, through a simple configuration or settings file. However, a wider, visual interface for this module is desirable for an improved user experience. A visual interface will make the application more intuitive to use and it will also make the application easier to present the application to new users. Both advanced and basic users will be able to see what the application can do.

Through a user interface the user will be able change settings and information without affecting usability. Through this module the user will be able to set several parameters that affect the way the application works. The most basic setting is to be able to choose what services are being shared to other instances via the WAN. It is also desirable to be able to control what services you receive from other nodes. Just because another node wants to share a service with other nodes, does not necessarily mean that you are interested in that service. This is a feature that is increasingly important and helpful as the number of connected nodes increase. To be able to select what services to share in real time is also a good solution as it allows you to temporarily take down a service for maintenance or any other reason. Republishing the service is done through a simple click.

In order for the application to support multiple different Web Service Discovery Mechanisms, a way to transform service data from one format to another is needed. This has been solved through a translator class.

In the bottom center is the last system component, the translator. This module is specific for each connected Web Service Discovery Mechanism. It is closely related to the Discovery mechanism interface, as its purpose is to “translate” the data going between the repository and the connected Web Service Discovery Mechanism. Each interface handles different variables and variable names. The previously mentioned issue concerning the richness of different Web Service Discovery Mechanisms makes it impossible to make this a static class that can handle all types of discovery mechanisms. A good solution for this module, though, is to create a parent class containing the most common functions and features, and for every new module you want to connect, you can extend this class. Through this design, you will only have to implement the functions that are specific to the connected module, while the more general purpose functions will be inherited from the parent class.

3.1.2.1 The WebService Class

As mentioned earlier, the repository is a list of the WebService class. The WebService class is designed to store each Web Service that it receives from both the WAN mechanism and the Web Service Discovery Mechanism.

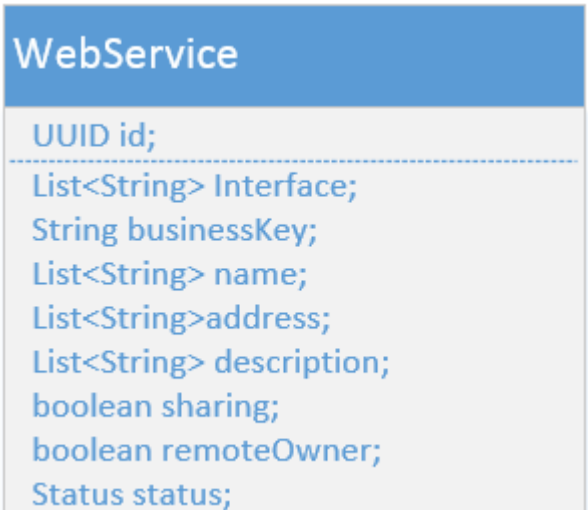

Figure 2 The variables of the WebService class

The easiest way to describe the WebService class is to explain one variable at the time. The first variable is the universal unique identifier (UUID), this variable is set to a unique id for each service, even though it has no knowledge of the other UUIDs used in other instances of the system. This variable allows the system to check if they receive a new service or an update when they receive data.

The next five variables are used to describe the service and where it can be located. Some of these variables are lists of data, enabling the service to hold for example multiple names, addresses and so on. This is also compliant with how WS Discovery represents a service, allowing multiple names, addresses, descriptions and interfaces. The three last variables are the most interesting ones and have been added as a custom field that helps the inner workings of the application.

The boolean sharing is a variable that is used to set whether or not a locally acquired service, either through the connected Web Service Discovery Mechanism, or through the GUI, is to be shared. If this variable is set to true, the service will be announced to all other instances of the system. They receive updates and also receive a message if the service has been removed by the user.

The next value is set to help the system to keep track of where the service is coming from. If it has been added through the GUI or the connected Web Service Discovery Mechanism, this variable is set to false. If the service is added from another instance of the system, this variable is set to true. This feature is used when the system receives incoming refresh requests, in order to only answer on behalf of services it has deployed itself.

```
1  package com.thuen.enums;
2
3  public enum Status{
4      NEW,UPDATE,DELETE,REFRESH
5  }
```

Figure 3 The last variable of the WebService classis an custom made enum.

The last variable of the WebService Class is the enum Status, as seen in Figure 12, it has four possible values. This enum is used to support the WAN mechanism. When a WebService is sent to another instance of the application, this enum helps the receiving instance to find out what to do with the service. When the service is received, it is unmarshalled, before if gets switched on this enum, deciding what to do next.

Repository synchronization

Although the synchronization is handled by the WsRepository Class, it is the different components of the WebService class that enables this feature.

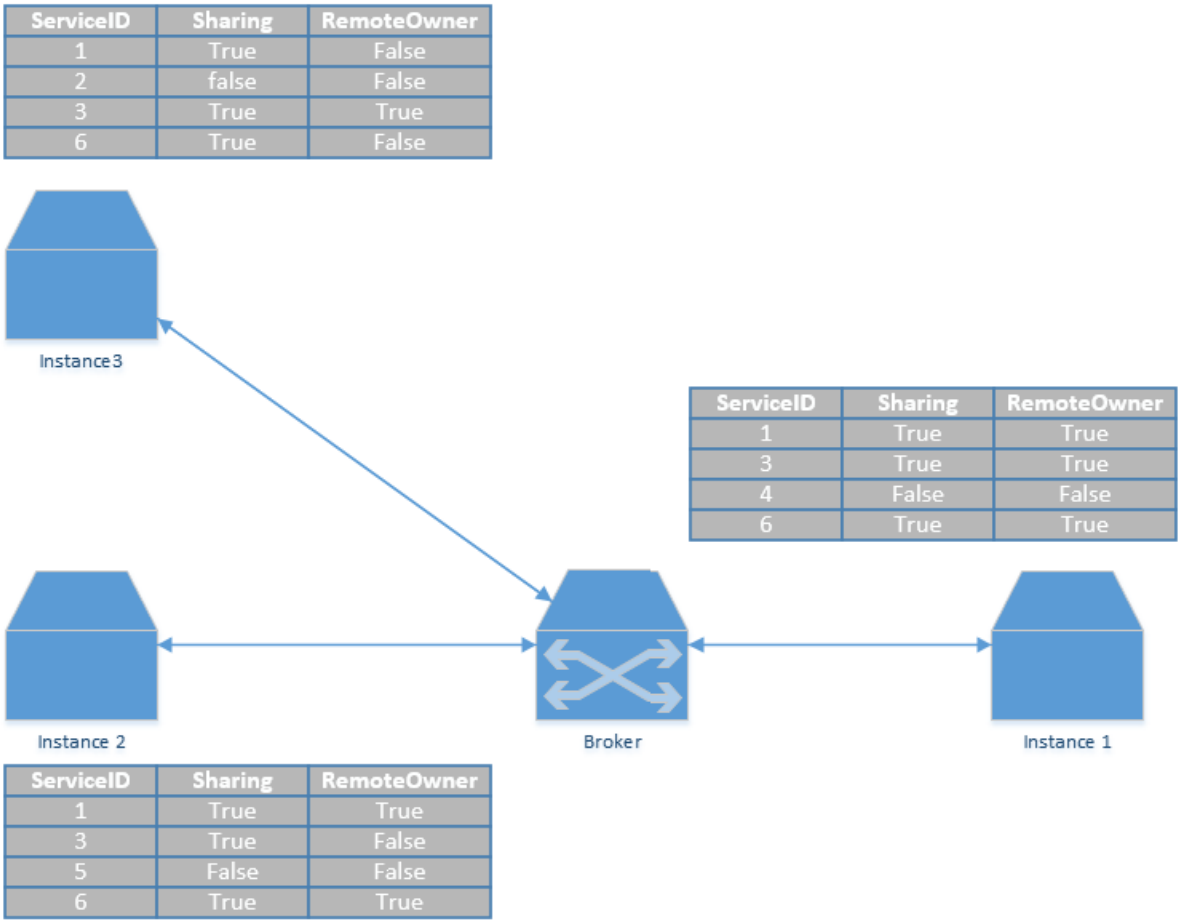

Figure 10 A network of three instances, each holding 4 services in their repository.

Figure 6 depicts a small system running three instances of the application. If Instance 1 is to send a update request, the following would happen:

1. Instance 1 deletes the services 1,3 and 6 from the repository, as they have remoteowner=True.
2. An empty, dummy WebService object is created by Instance 1, this is then sent to Instance 1,2 and 3. This webservice has the status REFRESH.

```
<webService>
    <id>8d983b9e-f98c-4f2f-98a6-4acbfd602eaf</id>
    <parameters/>
    <remoteOwner>false</remoteOwner>
    <sharing>true</sharing>
    <status>REFRESH</status>
</webService>
```

3. All instances receive this dummy WebService.
    a. Instance 1 finds out it is sent from itself, and therefore discards the message.
    b. Instances 2 and 3 unmarshall the message in to a WebService object.
4. Instances 2 and 3 then do a switch on the status value, discovering that it has the status REFRESH
5. This triggers the instances to resend their services.
    a. Instance 2 sends service number 3, this time with the status NEW
    b. Instance 3 sends services 1 and 6, also with the status NEW.
6. Instance 1 receives these messages and the services to the repository.
7. Instance 1 now has the same services as prior to the refresh, but with updated information.

Another common scenario is if an instance updates a Web Service.

1. Instance 1 updates the service with ID=1
2. When the service is saved, it is sent to the broker, now with the status UPDATE.
3. All instances 1, 2 and 3 receives the message.
    a. Instance 1 finds out it is sent from itself, and therefore discard the message.
    b. Instances 2 and 3 unmarshall the message in to a WebService object.
4. The WebService object is then handled based on the status of the object, since this has status UPDATE, instances 1 and 2 checks their repositories for services with matching id.
5. If they find a service with a matching ID, this service is removed from the repository, before the updated version is inserted.
6. If they do not find a service with a matching id, the updated service is added as a new service.

Both the message origin check and incoming message handling is done by the repository. This is done in order to separate as much of the logic as possible from the WAN mechanism, this makes it easier to implement new mechanisms.

Having only one object type to send, also makes the implementation of any WAN mechanism easier, as it only have to deal with one type of message, which again is handled and interpreted by the repository.

Whenever a repository receives a WebService with UPDATE status, the old WebService with a matching ID is deleted, before the updated version is inserted. This could also have been done by comparing every single variable of the new WebService to the old WebService, and then replace the ones that had changed, but this would require a comparison on every single variable in the object. When deleting the old service and inserting the new one, the amount of logic necessary is reduced, and the code is easier to understand and maintain. One option here could be to only send the serve ID and the updated variables, but then the sending service would have to know what variables the receiver had for all different WebServices, in order to know what variable to send, making the process much more timely and advanced than necessary.

3.1.2.2 Information flow

In the previous section the different components of the system were presented and a quick introduction to their functions and contents were given. In this section the aim is to present the information flow and interaction between the modules. Initially, the different work flows of information between the modules will be depicted through a series of sequence diagrams.

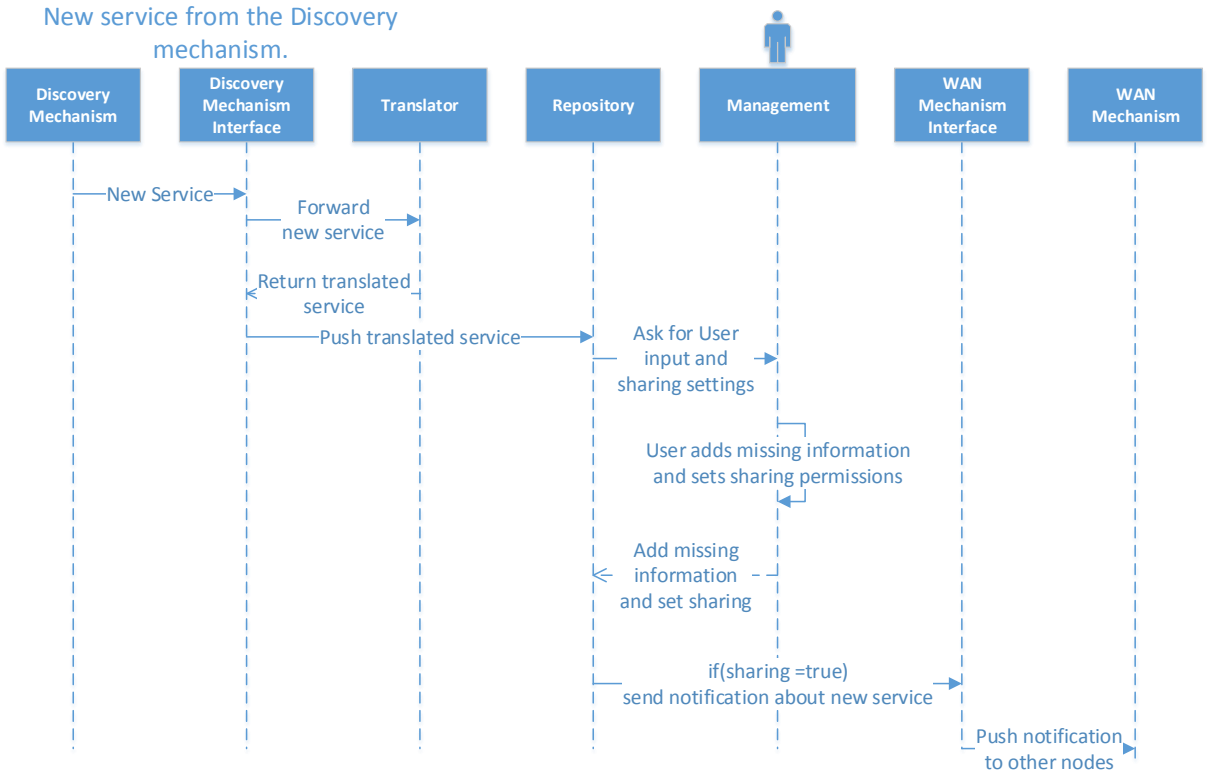

Figure 11. Adding a new service through the connected discovery mechanism.

Figure 11 depicts a very common scenario for the system. A new service is added or discovered in the connected discovery mechanism. The system receives a notification from the connected discovery mechanism, this notification informs that there is a new service available. The new service is then translated to match the repository data format. The service is then added to the repository, before the repository forwards the new service to a queue in the management module. The user then has to add missing information about that service, and decide if the service is to be shared to the other nodes. The service is available in the repository before the user edits it, but it will not be shared. When the user has decided what to do with the service, the process will either be terminated if no sharing is set for the service, or if it is said to be shared, it is forwarded to the WAN mechanism interface. The WAN mechanism interface pushes the service to other nodes.

After the new service leaves the WAN interface, the system has no control or responsibility for the service. Therefore there is no guarantee that the service is actually received by any other nodes, just a best effort solution. A possible improvement here is to implement an acknowledgement system, where the repository kept sending updates until it received an acknowledgement from another node. For this system to be effective it requires the repository to keep track of the other nodes and their identity. However, if a WAN mechanism using TCP is implemented, this can support message confirmation, but at a lower network layer.

There are also some assumptions made for this scenario, where one being that the connected service discovery mechanism has got a feature that allows it to send a notification when a new service is published. If no such functionality is present, this has to be achieved through requesting a full list of published services from the connected mechanism at a fixed interval.

Another assumption that has been made is that the size of the complete system is not bigger or more subject to change that it is an overcome able task for the system manager to keep track of incoming

and outgoing services in the system. If the number of nodes or changes is so great that the queue just keeps growing, one would have to implement automatic rules for sharing and accepting incoming services.

For this scenario to work, it is important to agree how the internal notification system should work. A simple example of this system is the event where a new service is added to the repository. It is not important where the service is coming from. What is important is how it is handled. How can we be sure that the connected discovery mechanism or the other instances of the application will discover and add the new service to their listings/repositories?

There are two main modes for managing notifications in a software system, notification based and polling based. The notification model works by having the changed element sending a notification by either broadcast or to all subscribing listeners. In the previous example it would have been the repository sending a notification to all parties that would need to know there was a new service. The benefits of this structure is that it reduces the amount of excessive network traffic, and there is very little or no delay. It also has low computational overhead, and is easy to implement.

On the other hand, notification based has got some shortcomings, it is impossible for a node to know whether there has been no new events since the last event, or if something is wrong with the sender. If the sender was to go down or become unavailable, the other nodes would essentially never find out about it. This can be handled in different ways, but the most common way to overcome this problem is to use a fixed interval ping request, to make sure that the node sender is still available.

This leads to the other option when it comes to notifications, the polling based model. This model assumes that the part of the system that needs information about changes, will have to discover these changes for itself. The node will send a request at a fixed interval, asking if anything has changed since the last request, if yes, send the new data, of no, send a false response. This is a simple approach that diminishes the problem of the missing host, as if the client can not reach the target it is pinging, it knows that something is wrong.

However, there are two challenges with this approach. The first is more of a shortcoming than a challenge; the system can have quite a delay to it. If a new service is added, the other nodes will not know until they ask, depending on how often they ask, this can be a problem. The second problem is somewhat related to the first one. To keep the system updated at all times, the update frequency has to be set to quite high, this will cause a significant amount of unwanted network overhead, taking up bandwidth from other parties.

The best solution for our application where bandwidth sometimes can be an issue is therefore a hybrid solution. The instance that includes the service discovery mechanism that initially adds the service, will always be responsible for informing the other instances of a new service. This can be done due to the fact that whenever it receives a notification from its connected service, it compares the content of this notification to check if there has been any changes to the services, if yes it forwards the service. There is also be periodic ping messages, where each instance can ask other instances for their list of services. The timer for this ping function can be set by the user. This way the user can select an appropriate value based on their current network capacity and the level of reliability they desire.

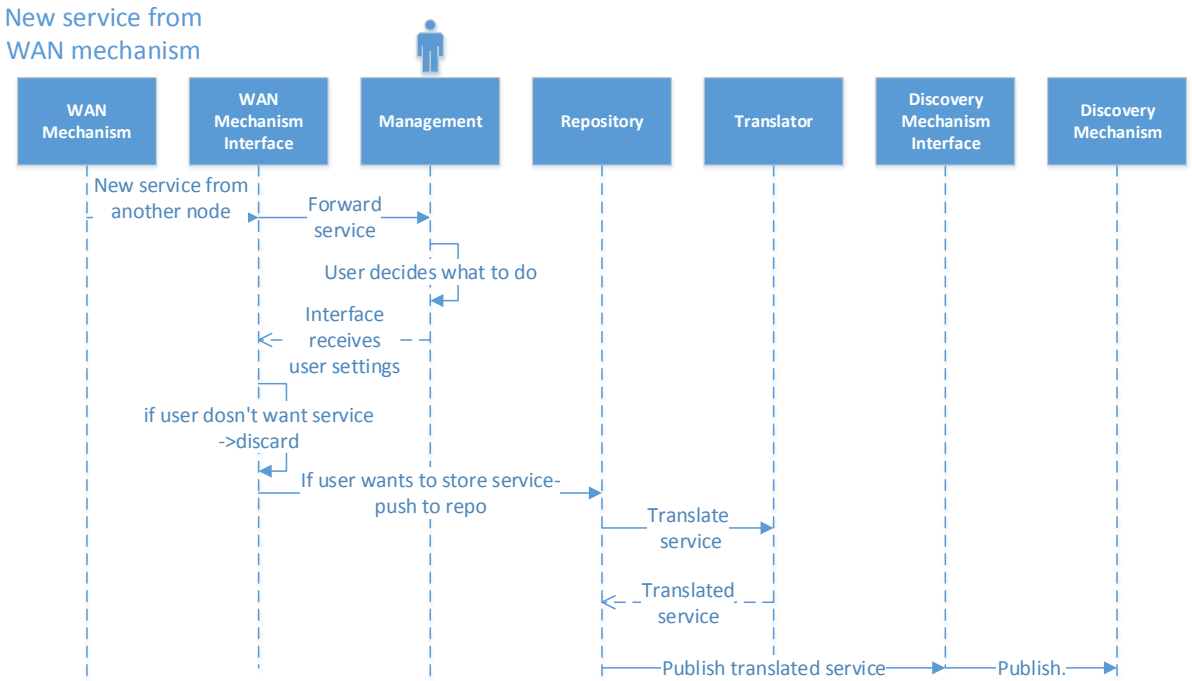

Figure 12. Adding a new service through the WAN mechanism

This scenario depicted in Figure 12 is also very common. The diagram has been inverted, but the workflow is very similar to the previous diagram. This is because the WAN mechanism and the Web Service Discovery Mechanism has very similar tasks, only in different directions. A new service or a service update is received from the WAN. The WAN mechanism interface handles the new service and forwards it to the Management queue. It remains in this queue until the manager of the system has either approved the service or dismissed it. If the service is approved, it is forwarded to the repository for saving. The repository will then forward the new service to the translator for translation. The translator returns the translated service object, which in turn is pushed to the connected service discovery mechanism, via the discovery mechanism interface.

There will also have to be a function that handles looping when it comes to new services. When the new service is added to the discovery mechanism, the next notification, stating that the discovery mechanism has got a new service, will have to be suppressed. There will also be a built in test in on the repository, checking if a service is already listed in the repository.

This scenario also makes some assumptions for it to work as planned. The first one being that all received services from other nodes in the network are already fully populated with values for all variables. This can be ensured by not allowing to share a service from the repository before all variables are set. The assumption concerning the size of the system from the previous figure is still a valid assumption.

An important premise for this diagram is if the interfaces for both the discovery mechanism and the network transport application is not connected to several instances of a discovery mechanism or network transport application at once. This premise has also been mentioned as a constraint in chapter 3.1 Design, third section.

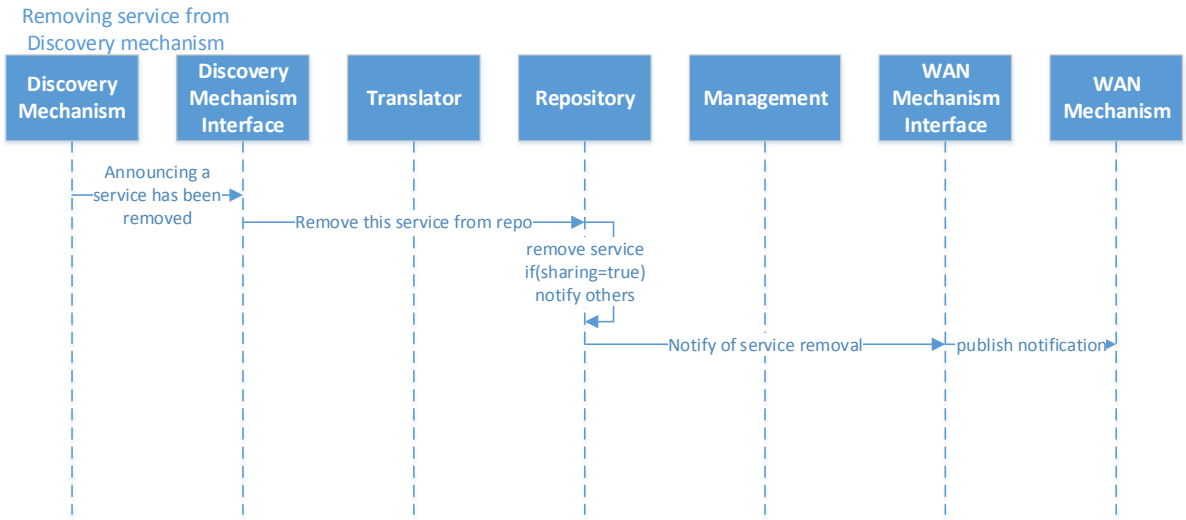

Figure 13. A diagram depicting removing a service from the connected discovery mechanism

Figure 13 shows a service being removed from the connected Web Service Discovery Mechanism. The discovery mechanism sends a notification saying that the service has been removed, the repository finds the matching service by ID and removes is. If the service was being shared, the notification is forwarded to the WAN mechanism interface and pushed to the other services, giving them the opportunity to remove it as well.

There is no guarantee that the other nodes in the network will receive this update, which could result in a liveness problem. This problem could be mitigated by the use of acknowledgements, which was discussed in the second paragraph below Figure 13.

This model is also made based upon some assumptions, the first one being that the connected Web Service Discovery Mechanism supports some sort of notification functionality where it can alert other nodes of changes. Resolving this potential issue has been discussed in the third section below Figure 11.

Another thing to keep in mind for this diagram, that many Web Service Discovery Mechanisms will only remove a listed service if it is actively unlisted by a user, making them prone to the liveness problem. For the system to be able to remove a service from the repository, it is essential that it is first removed from the service discovery mechanism.

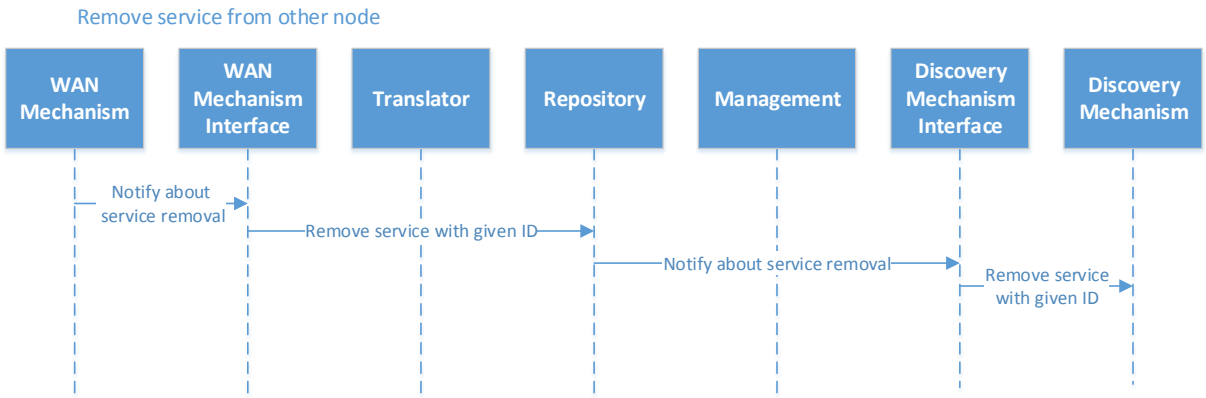

Figure 14. When a service is removed from another node and the local node receives a notification about it.

This scenario in Figure 14 is similar to the previous one, but the information flow goes in another direction. The WAN mechanism receives a notification that a service has been removed and is no longer available. It forwards this information to the repository, the repository then checks if it has got the given service, if yes, it will be removed and the notification will be forwarded to the connected Web Service Discovery Mechanism, where it is also be removed. This process does not involve the translator as it only compares the service ID, not the object, and no translation is necessary.

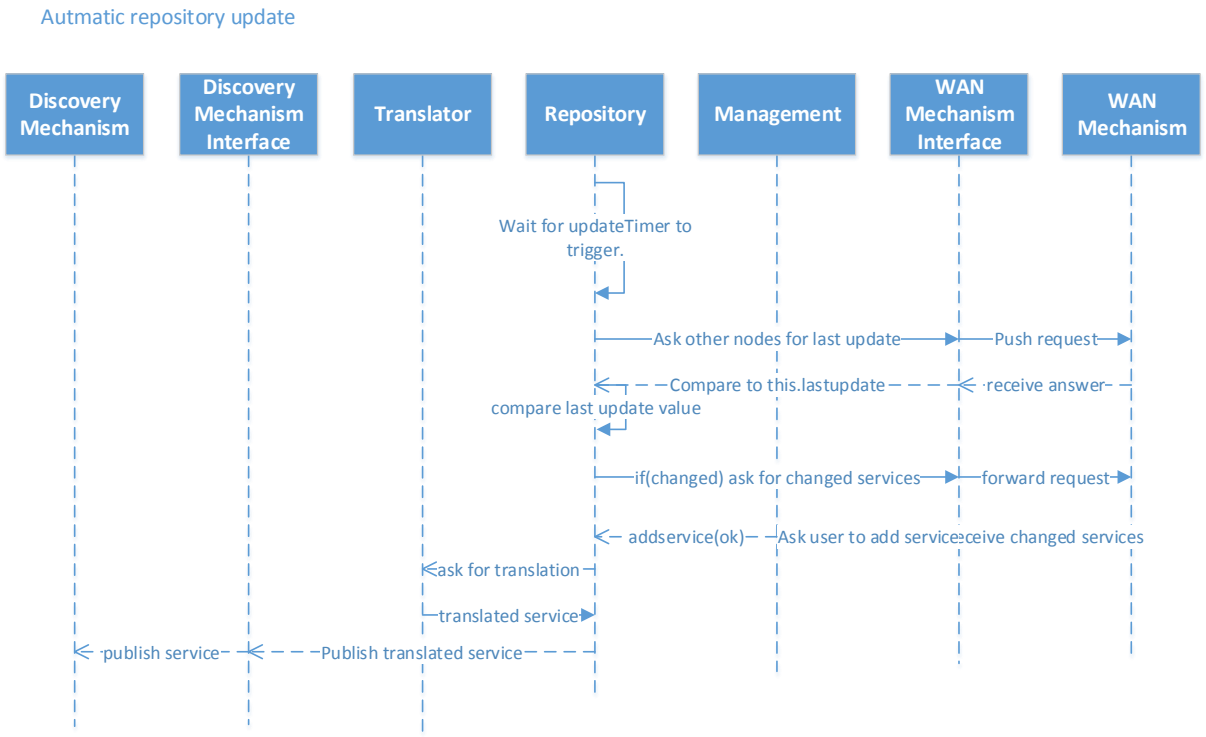

Figure 15. Automatic timer for repository update.

To ensure redundancy and to mitigate the impact and number of occurrences for the liveness problem it has been decided to implement a function that will work as an addition to the notification status updates between the modules. As seen in Figure 15, this function will poll the other nodes at a fixed interval and check for any un-notified changes. This also reduces the need for an acknowledgement on the updates, as there will be regular updates anyway.

The repository has a timer, where the timeout is set by the manager through the management console. When the timer has run out, a request is sent through the WAN mechanism, asking for the last update time of the other repositories, when the request is answered, the timestamp is compared to the one in the local repository. If the other repositories have newer timestamps, a new request is sent to the other repositories, requesting all services that have a timestamp which is newer than the local repository timestamp. All new services is then be returned and added to the management queue. When the user has approved the new services they are added to the repository. When the services have been added to the repository, the repository translates the new services and pushes them to the connected Web Service Discovery Mechanism.

It has been assumed that no extra check between the repository and the connected Web Service Discovery Mechanism is going to be performed at this stage, as it would increase the complexity a lot. If there are multiple nodes connected through the WAN mechanism, the node performing the update will then have to compare its timestamp to a list of other timestamps, preferably a key->Value map, where the node address is the key and the timestamp is the value. This will allow the node to query the specific node with the newest timestamp, for its services.

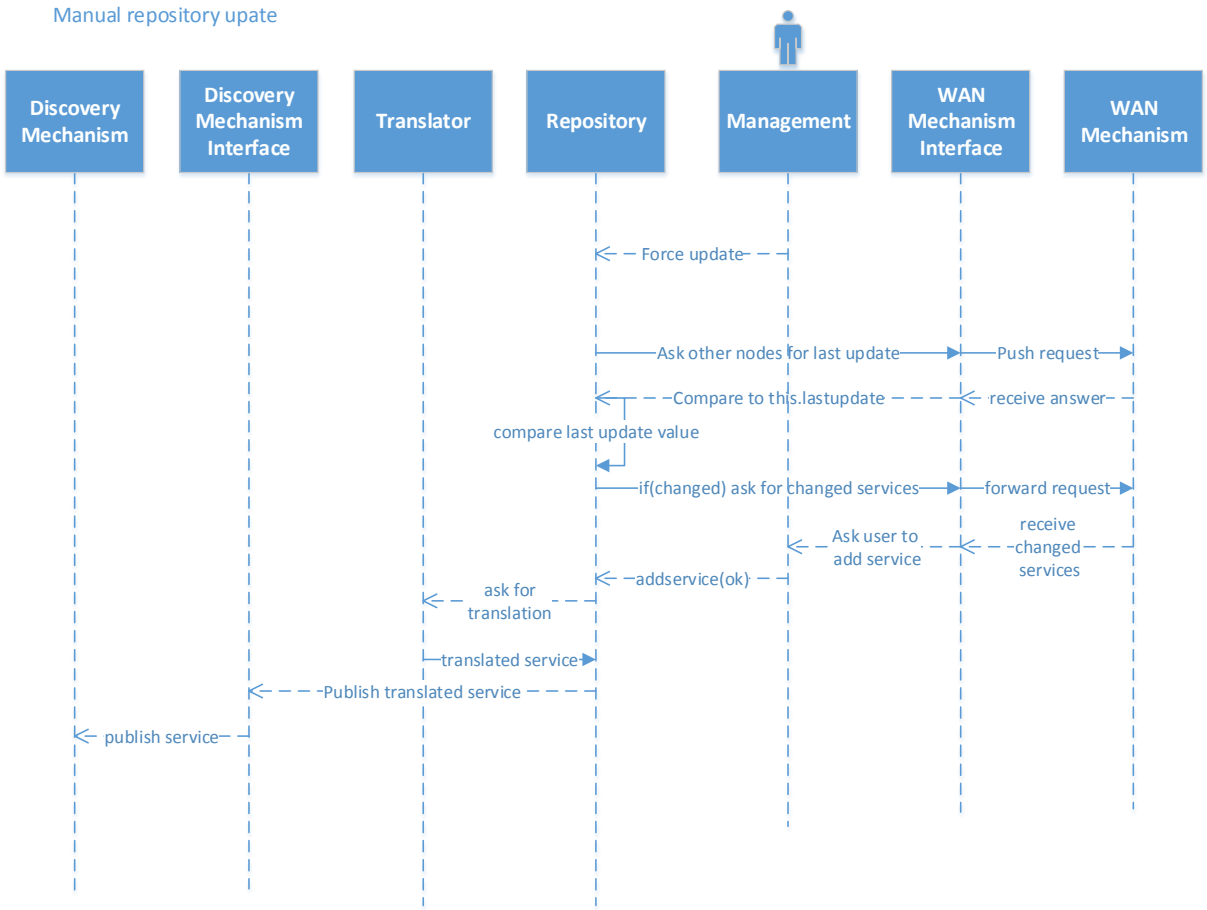

Figure 16. User initiated repository update.

This sequence diagram in Figure 16 is the same as the previous one except for the fact that this one starts when the user pushes the update button in the GUI, forcing the system to perform an update.

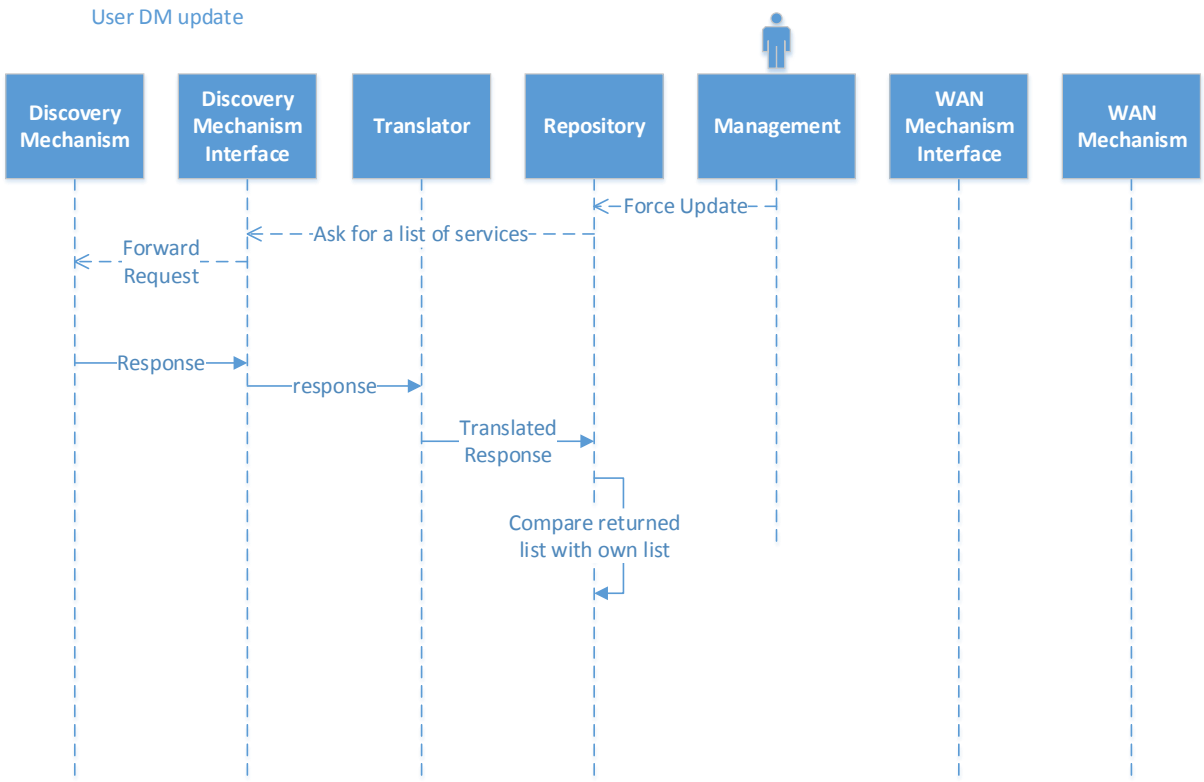

Figure 17 Automatic and manual Discovery Mechanism update

Figure 17 shows the process of manual/automatic update of the connected Web Service Discovery Mechanism. This will be modeled and implemented as a separate function, but it can be included in other functions to use it as for example a part of the WAN update mechanism.

The user pushes the update discovery mechanism button, and a request is sent to the discovery mechanism interface. This request is then forwarded to the connected discovery mechanism. Since there is no guarantee that the connected discovery mechanism is able to support the previously mentioned timestamp method, the request will simply ask for a list containing all the available services in the connected discovery mechanism. This list will be passed through to the translator, where it will be translated into a list that can be compared to the repository. All the services in the list is then inserted into the repository, replacing the unchanged ones and adding the new ones.

If a service is not previously known to the repository, and additional information is needed, the user will be prompted for this information through the GUI.

This process can also be automated through a timer that will trigger the event, the information flow will be exactly the same, but the trigger will be automated instead of user imitated.

3.1.2.3 Interfaces/Modules

Each of the components in the federation mechanism are going to be made after the following predefined interfaces. This interface contains information about the methods of the class, followed by a brief description of what each method does. This design enables easy swapping of the different parts of the system. The different modules can be changed with other modules, implementing the

same interface. Hence, developers can create new modules, without any prior knowledge of the code, and they will still fit right in.

Discovery Mechanism Interface

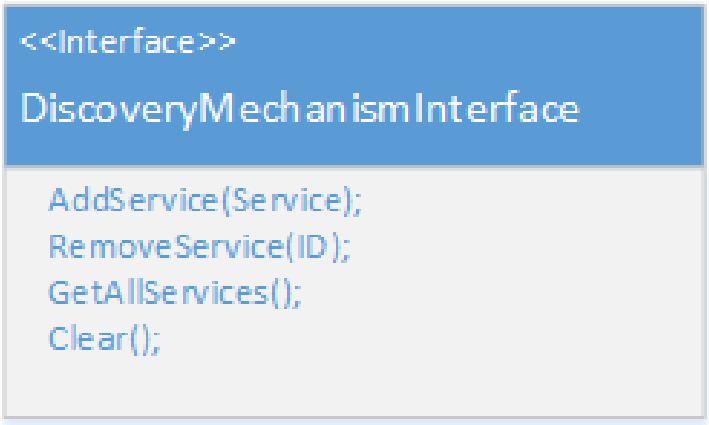

Figure 18. A simple visual representation of the Discovery Mechanism Interface

The DiscoveryMechanismInterface is a connector module, that aims to serve as an intermediary node between the system and the connected Web Service Discovery Mechanism. Since this component is going to be changed for each type of different Web Service Discovery Mechanism, it enables the translator and repository to only adhere to one interface, and what is going on beyond that interface is insignificant for the modules in the other end. All communication going from and to the discovery mechanism will pass through this interface.

Translator

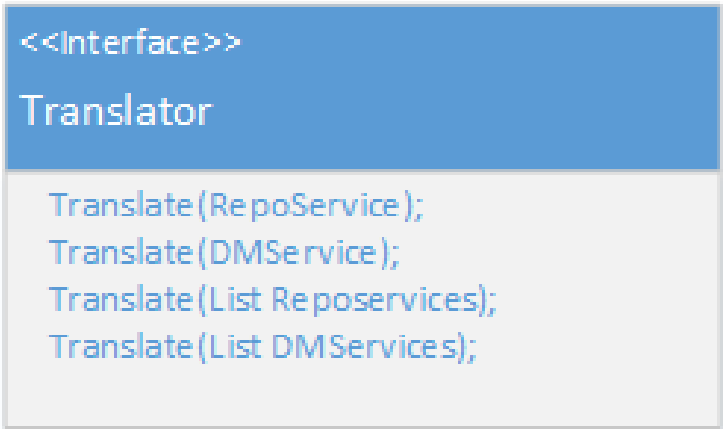

Figure 19. Translator, responsible for translating services between the DM and repository.

The Translator is a support class, it works in the background, and can not be seen from the outside, but it is still essential for the function of the application. This component will also have to be customized and adapted for each different discovery mechanism, it is essential for the efficiency and ease of use that all services stored in the repository are of the same type. Different types of Web Service Discovery Mechanisms can have different interface richness, hence some may have ten variables pr. Web Service, whilst others may just have five. Furthermore, even if some of the

variables represent the same values they may still not have the same name, enforcing the need for a Translator.

Repository

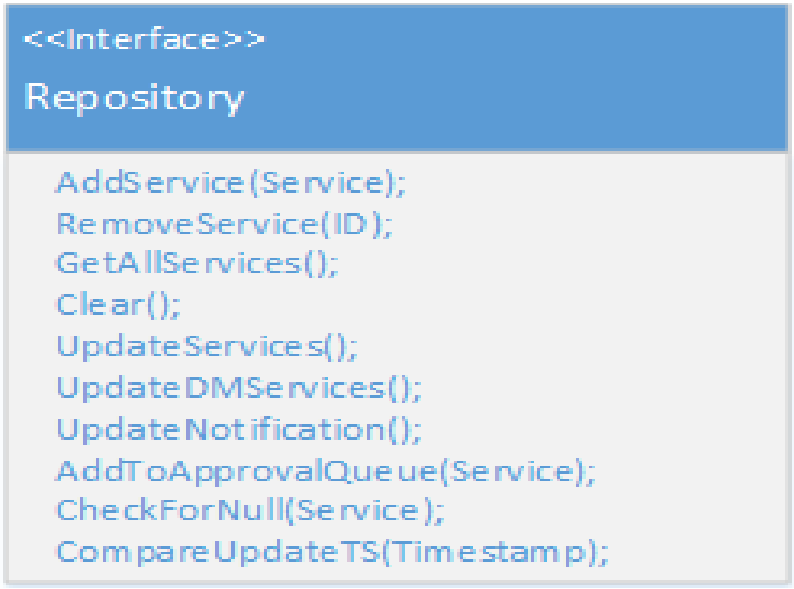

Figure 20. Repository Interface, responsible for saving all Web Services.

The Repository is the heart of the application. It is responsible for keeping track of all available Web Services, and their status. Through the Repository the user can add, remove and update services. The repository will also be responsible for keeping itself and the other nodes up to date. This will primarily be done through notifications sent whenever there is a change in the repository. It will also contain a settable “auto update” feature, that polls the other nodes for any changes, this is to increase the robustness of the application. The repository will be a list containing WebServices, the Service class is going to be implemented to best fit the needs of the application. The repository will also be able to ask the user for additional input on services, and whether or not to share a specific service with other nodes. A new service coming from either the connected Web Service Discovery Mechanism or the WAN, will not be added to the repository before it has passed manual inspection, the manual inspection requires the user to add missing information about the service, and whether or not they want to share or accept the service.

Management

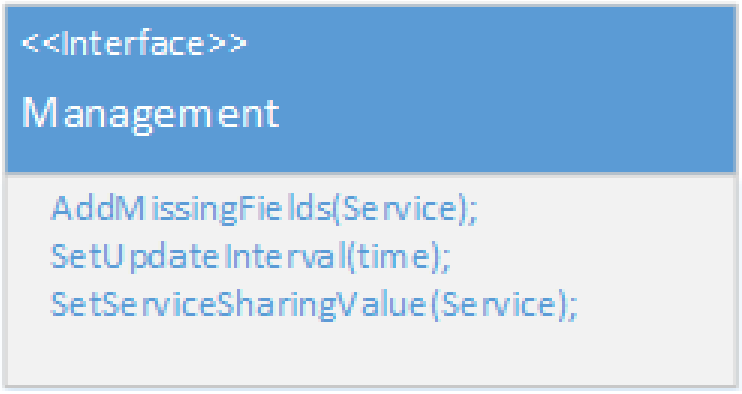

Figure 21. The management interface is the graphical user interface of the system.

The management module is an interface where the user can specify application settings and do monitoring. All of the functions offered by the Management module are executed on user requests or actions. The size and complexity of this module will vary with the amount of time and effort put into developing it. The very basic functions are quick and simple to implement, whilst more advanced options can be included if time allows it.

The management module will be the only graphical user interface for the application, making it the applications face for many users. This is an aspect not to be underestimated, as it can dramatically affect the user's impression of the application.

WAN Mechanism Interface

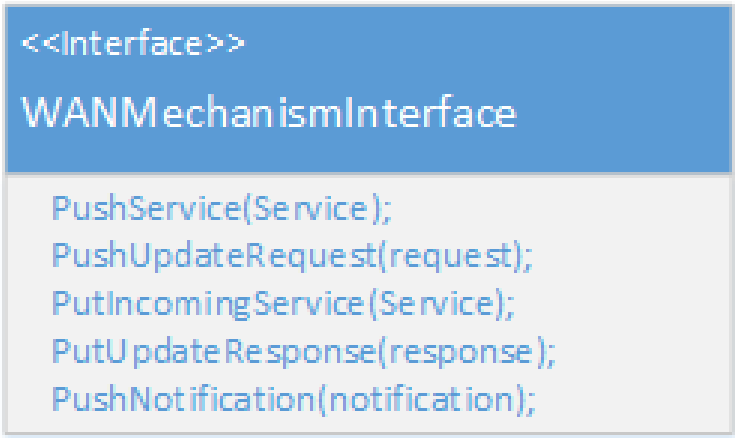

Figure 22. The WANMechanismInterface is the intermediary between the system and the WAN mechanism.

In order to prepare the system for any type of WAN mechanism, a WAN Mechanism Interface has been defined, the purpose of this is to offer the other components in the system a stable and predictable interface that they can use regardless of the connected Web Service Discovery Mechanism.

This makes the system modular and makes it easy to develop new modules, that can easily be swapped with the existing ones.

This module is responsible for forwarding all push, pull and update requests coming from the system going to the other nodes in the WAN. There has been set a requirement that all data passed to the WAN mechanism should be XML serialized, and therefore the implementation will have to create a XML schema that can be used for validating, marshalling and unmarshalling the objects being passed to this module.

3.2 Implementation

The implementation is described below, and the differences from the application design is emphasized and explained for those cases where it was necessary to deviate from the original design.

The general overview implementation matches the design well, although some minor changes have been made during development. Some of the functions that were supposed to be inside the five main modules have been moved to separate classes in order to facilitate modularity and code re-usage. A utilities package has also been added, containing data for logging, configuration files and other helper functions.

A GUI has also been created. This can be used to monitor the different services available in the network, and view information about who owns the services and where you can find them. The GUI also enables the user to add, change or remove services to the repository. The user can also select if a service is to be shared to other nodes connected to the network. It also enables the user to change the settings of the application, such as IP address, port number and network protocol for the WAN interface to use, although these settings will not be changed until the application has been restarted. A version of the application without the GUI has also been created, this version is started as a java.jar file, and runs as a background process, it is able to publish services from a configuration file, and work as a part of a network. There will however not be any opportunities to add, change or remove services during runtime from this mode, as there is no way to interact with the application, to do this the application has to be restarted. This headless mode includes a fully functioning version of the application, running both a Web Service Discovery Mechanism and a WAN mechanism, this is a handy tool for quick setup, testing purposes or running on nodes without a GUI, such as servers.

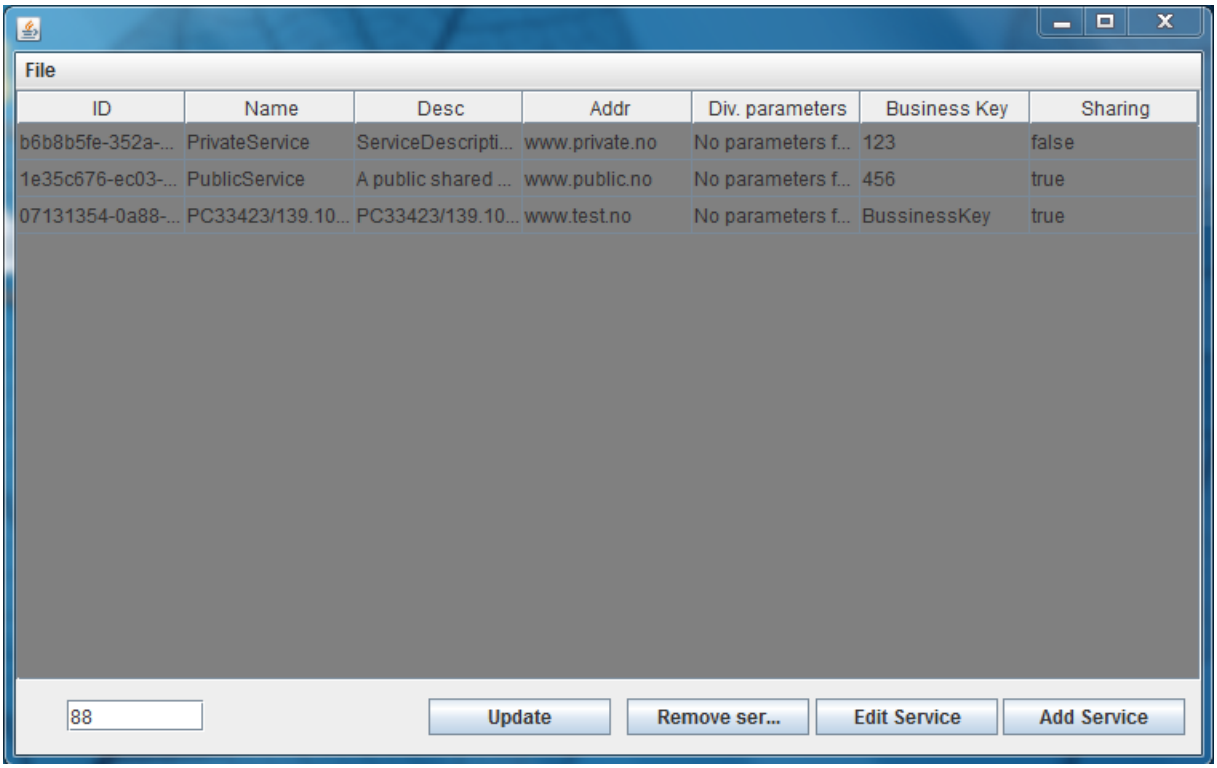

Figure 23 The application GUI.

Figure 23 depicts the application GUI containing three services. The two top services are hosted by the instance running the GUI, one of them are shared to other instances, while the other is not shared. The bottom service is hosted by another instance of the application, running on the same machine, but with a headless client, this client is predefined to publish a service named after the computer it is hosted on and the IP address of said computer. This is a very handy setup for testing purposes, as each node will publish a unique service and respond to any requests sent to it. The deployment of such a service is done with a single click. The number in the lower left corner is intended for performance testing purposes, and measures the time from the “Update” button is pressed, until the response is received from the neighboring nodes. This process is described in greater detail in chapter 3.1.2.1 The WebService Class, under the Repository synchronization section.

The GUI is a very small part of the application, most of the work done was on the back end of the application.

The Discovery interface and the Wan interface are designed in accordance to the designs given in chapter 3.1 Design, three methods has been added to ease the implementation work, as well as reduce the amount of unnecessary code. A listener has been added to both the WAN interface and the Web Service Discovery Mechanism interface. These listeners allow them to act when a change occurs in the repository, and stay idle when nothing happens. This facilitates the principle of loose coupling, as the only link between the Web Service Discovery Mechanism and the WAN mechanism is this listener. This also improves the resource usage of the application, as no actions are taken before it is needed, prompting the service to use less resources when nothing is happening.

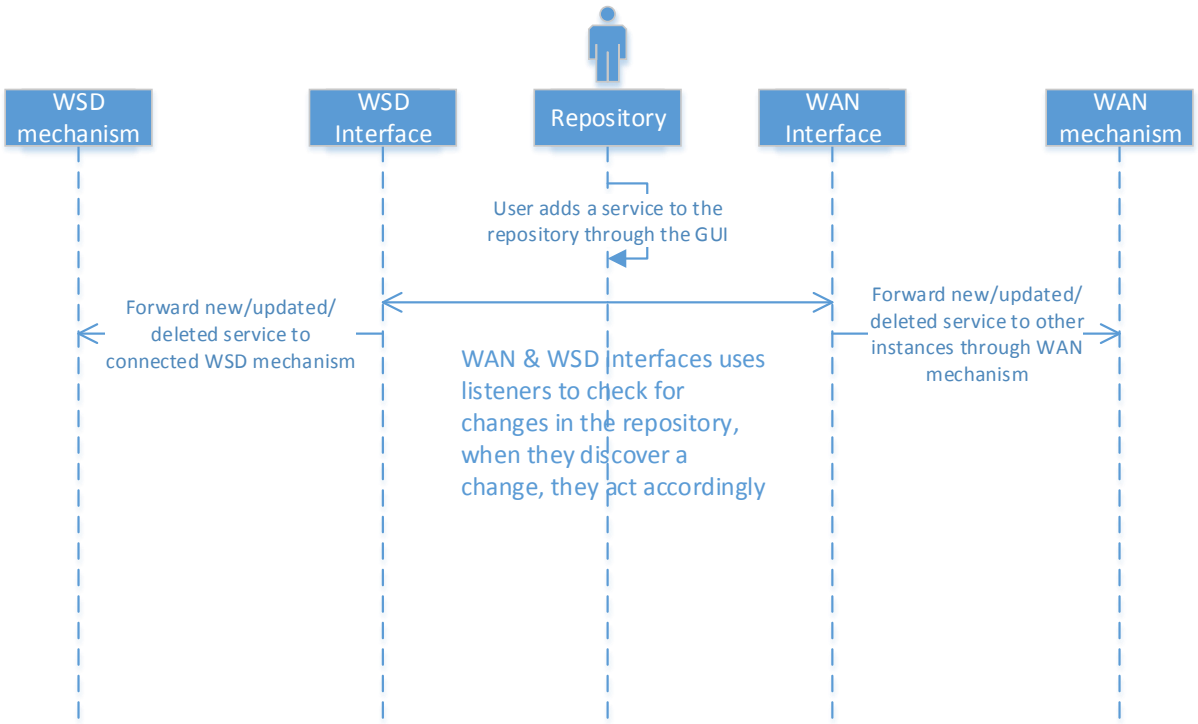

Figure 24 The implemented workings of adding a web service.

A publish(service) function has been added, as the implementation proved that the easiest and most efficient way to publish a service, is through the GUI. This means that instead of having to add services directly to the connected Web Service Discovery Mechanism, adding services is now done through the GUI. As seen in Figure 24 the service is first added to the repository, which again forwards the new service to the WAN mechanism as well as the Web Service Discovery Mechanism. The Web Service Discovery Mechanisms then publishes the service locally, and the WAN distributes it to the other instances, where it is added to the repository and published to the local Web Service Discovery Mecahnism. This means that if a Web Service is added through the connected Web Service Discovery Mechanism, it will not appear in the repository. This allows the user to have full control over what services are stored in the repository, as well as what is being published to the connected Web Service Discovery Mechanism, in accordance with requirement number 9.

In the chapter 3.1.2 General design, the section Management describes what now has turned into be a part of the GUI. However, the GUI is still reading and writing these values from and to configuration files, enabling a user to change them without going through the GUI. The designated tasks of this module have been incorporated as a part of the GUI module, but some of the operations have been moved a little. A service from the another instance, that has been received through the WAN mechanism will now automatically be added to the repository. This omits the step where the user would have to approve and fill inn the blanks of a service before it was added, these functions are still available, but must now be performed after the service is added to the repository. This saves both users and developers a lot of time, as it requires less code, and the application will be able to run unattended and still work. This enables the headless application to run on a server, which it could not have done if each new service would have to be approved. On the other hand, this may increase the amount of work needed to administer the application when there are many nodes and instances, but this is outweighed by the benefits.

In the design of the application section 3.1.1.1 Service repository, 5th section, a redundancy mechanism that periodically checks for new data was proposed, this function has not been implemented. This is due to the fact that the implemented WAN mechanism is running over TCP, and this has proven to be very efficient for the applications reliability. In addition, the function that performs this update has already been implemented, the only difference is that instead of having a timer triggering the refresh requests, they are now triggered by the user. This will save bandwidth as the user only have to refresh if they suspect something is wrong. If this feature proves to be needed it is very easy to implement.

The translator module, which was responsible for translating services going between the repository and the connected Web Service Discovery Mechanism and vice versa, is implemented as planned. A translator will only work for one type of discovery mechanism, and additional translators will have to be created when connecting other Web Service Discovery Mechanisms. The current translator implemented as part of the application can translate messages from and to Java WS Discovery. As mentioned in chapter 3.1.2 General design section 11, a new translator will have to be implemented for every connected Web Service Discovery Mechanism.

In the third section below Figure 11 a problem concerning the ability for the many types of Web Service Discovery applications to be able to send notification to the repository when changes are made to a service. In order to avoid this problem, a slightly different setup for the message update system has been selected. There are no timestamps to verify the version of a service. Each service is identified by a universal unique identifier (UUID) that is set when the service is first created by a user. Whenever this service is sent to another instance, it just checks if this UUID can be found in the repository. If it can, the old will be deleted and the new one will take its place.

This logic is also applied to the refresh repository function which can be seen in Figure 10. Whenever a user refreshed the repository, all remote services are deleted from the repository and the connected Web Service Discovery Mechanism, before it request copies of all services from all other nodes, the services received is then added to the repository and published. However, this function is meant as a supplementary function for the notification system of the repository. A notification is sent if a new service is added, if a service is changed or if a service is deleted, this notification also contains the new, changed or deleted service, so that all the other instances receive the latest data. Due to this, the most common usage of the refresh function is for a instance to send a refresh request when it joins the network, this ensures that the new instance receives all the previously shared data from the other nodes.

The fourth section below Figure 15 addresses a problem where an instance of the application is responsible for forwarding updated information it has received from other nodes. This has not been implemented in the application as the other nodes should also have received this information at the same time, making this forwarding excessive.

A validation to remove any possibility of endless message loops have also been implemented through the use of a origin test. If the message received has the same origin ID as the node, saying the message came from the node itself, it will discard the message.

A resources folder has also been added to the project. This contains three .properties files: log4j.properties, config.properties and services.properties, the first is a configuration file for the logging framework used in the application. Log4J [56] is a well known logging framework enabling multiple logging levels and customizable outputs. The use of a logging framework in the application simplifies the development process, and is a great advantage for other developers who may want to continue to work on the application, as it makes is easier to debug and expand.

The config.properties file is used for application specific settings, this includes many options for the connected WAN mechanism, what IP it should connect to, if it should start its own server, the topic it should subscribe to and many other settings. This enables the user to easily change these configurations when the application is being deployed, which is of great help during the testing and other deployment phases.

The last file, services.properties contains services that you want to automatically deploy to your repository when the application boots, this is very handy if you want to be able to start the application fast and easily, or if you cannot or will not use the GUI.

Four different helper classes have been introduced to the project, which was not a part of the original design. These have been introduced in order to ease the development process and increase code reuse in the application.

The first one is a PropertiesReader class, which allows the user to fetch the values from the different properties files, and use them as variables in the code. The second class is a utilities class, containing small helper functions that are used in multiple locations in the code, by gathering these in a utilities class, the code can be reused and it also facilitates code maintenance, as you only have to change the code one place, instead of all the places it is being used. The third helper class is the JAXBParser, which is responsible for marshalling and unmarshalling the web services to and from XML documents, as the services required to be sent as an xml schema through the WAN mechanism. The last helper class is a Java Main class, called Application Startup, this is used by both the GUI and when the headless nodes start without the GUI. This enables the user to start the application from any context they like, which makes the application easier to integrate as a part of other software projects.

Except from the changes and features mentioned above, the implementation of the application is done in accordance to the design described in chapter 3.1 Design.

The entire thesis, application, tests, test results and any other artifacts associated with this thesis and the work herein has been made publicly available through google code under a GNU public license [7] at https://code.google.com/p/java-ws-discovery-federation/.

4. Testing and Evaluation

In order to ensure that the application meets the requirements set in section 0, as well as testing that the software works as intended, different tests has been created.

Testing is a part of the Software Development Lifecycle (SDLC) [57], this makes it a major part of the software development process, and it may consume as much as 40-50% [58] of the resources invested when developing an application. There are several different approaches to take when testing an application. These different approaches can be broken down into smaller pieces, which makes them easier to understand and differentiate between them.

The most coarse grained pieces you will encounter in software testing are the two alternatives, automated and manual testing. The names are rather self-explanatory, but the benefits and drawbacks of each of these are not as obvious.

Automated testing is when another application is used to test the software. Software testing is very well suited when the performance under load or during a large number of repetitive actions are to be performed. A computer can easily run the tests and record the results for each iteration. The tests can be run an infinite number of times, only limited by the amount of time that is available for testing. Since no employees have to perform each test, the tests are time and resource effective. They can cover a large amount of the code although this depends on the type of application.

Manual testing is when a tester or developer manually tests code component by component, the tests are very flexible, and almost any piece of the code can be tested, but can be limited by the amount of time and resources set aside for testing.

A very common approach to this is to utilize both automated and manual testing, this allows the developers and testers to automate as many of the tests possible in order to save time and money. The more complex and specialized tests are done manually, to ensure the quality of the application. This is also the approach chosen for this project, if a test is automated or manual will be stated in the introduction about each test or test group.

Another aspect to consider when talking about software testing, is the tester's knowledge of the system. Because the end user is most likely going to be a person without specific expertise about the application, it is also a reasonable idea that some of the tests should be executed by a tester with a similar knowledge to the inner workings of the application. There is a name for the different levels of knowledge about the inner workings of the application, in testing. It is referred to as White Box and Black Box testing.

White Box testing or Glass Box testing refers to a transparent system, where everyone can see the inner workings of the system. A tester or a developer with a good understanding and knowledge of the inner workings of the system usually does white box testing. Using this knowledge, the test can be optimized to trigger specific features or paths inside the application, and test these. A drawback using white box testing is that it requires a user with good knowledge of the system and how it is built, meaning it cannot be done by anyone, making the process more expensive.

Black box testing is where the user does not know, and can not see what goes on inside the system. The user simply inputs data to the system and records the output from the system. Due to the limited knowledge of the system, the user is more likely to input unexpected values, and therefore

testing the system. One advantage of black box testing is that you do not need a tester with special knowledge about the system, and can therefore be outsourced to other parties.

The testing of this system will be white box only, as the only person responsible for testing, is also the developer that created the application, this makes it hard to develop the tests data in an unpredictable way, as there will always be knowledge about the systems inner workings.

The different test types mentioned above can be utilized in many different test scenarios, in the next sections three common test scenarios are listed, as well as why they are good tests to perform on this application. The tests chosen for the development of this application are listed below and will be performed during different stages of the development process. The unit tests and the functional tests will be manually conducted, whilst the performance tests will be automated to ensure as reliable results as possible.

4.1 Unit Tests

Unit testing has been briefly mentioned in the last section of chapter three. The unit test are the most basic form of testing. A unit test usually focuses on a single unit of code, like a class or a method. This unit is then tested independently of all other code for the application, to make sure that it works. Most units tests conducted are categorized as manual. The unit tests can either be positive or negative, whereas a positive tests gives the unit a valid value and then checks that the returned value or the finished product is valid, if it is, the test will pass. A negative test is a test where you give the unit an invalid value and checks that the unit handles this correctly, either by throwing an exception, or notifying the user that the given value is not valid and then return 0, for example. A good mix between these two types of tests allows for a thorough check of the code, before any deployment or actual running, saving a lot of time and any surprises.

The application implements unit tests on most major components of the project. The tests are structured so that most application classes has a corresponding test class. Each of the test classes contains tests that utilize the functions of the of the application class.

One of the major benefits of unit tests is that they can be implemented early in the development process, enabling the developer to discover and resolve bugs as early as possible. The earlier a bug is resolved, the less time consuming the process is. Unit tests are also quite simple tests, which is why writing unit tests can be outsourced to less experienced developers, without the need for a thorough understanding of the complete workings of the project.

But while unit tests ensure the developers that each component of the application is working as intended, there is no guarantee for that the components as a whole will work with each other, making unit test an important part of a more extensive test regime for an application.

Because it is the most low level, and basic test type to perform on a system, several different unit tests have been implemented as part of a parallel test project. The most basic and important parts of the application was therefore tested through the use of the tests depicted in Figure 25.

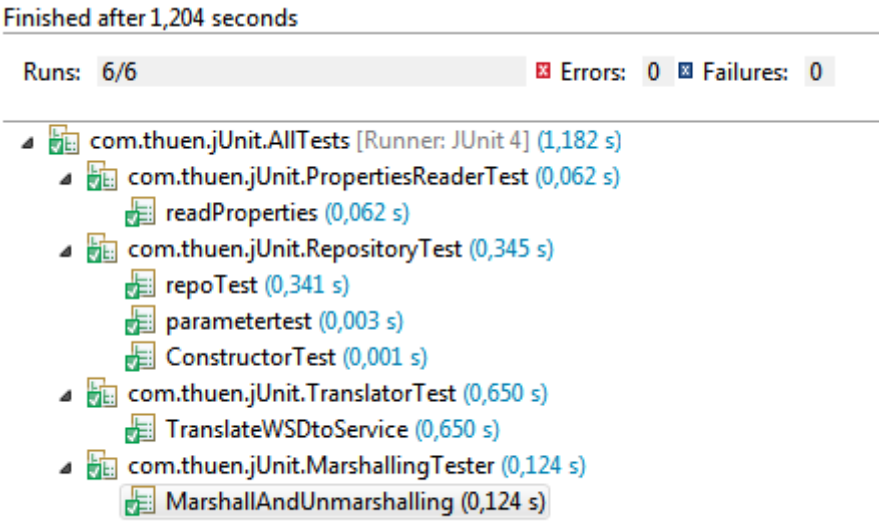

Figure 25 The results from the JUnit tests.

The repository test class creates as repository, and two mock Web Services. The first service is added to the repository, before it is retrieved back out from the repository, and the BusinessKey variable is compared to the one set on the mock service before it is added to the repository. If these variables are not identical, the test will fail. Further, the services has a function where you can add a list of addresses, as well as adding single addresses, to make sure that an address is not overwritten when a list of addresses I inserted to the service. The service now has three different addresses, one added directly, and two inserted into a list, which then is added to the service. The test checks that the service has got three addresses, and that the directly inserted service is not overwritten by the list of added services.

The second service is then added to the repository twice, this is to check that a service can not be added two times to the repository. If the number of services in the repository after this operation, is more than two, the mechanism to prevent duplicate insertions of services will have failed.

Further a similar test, checking that adding a hash map of parameters, that this does not overwrite the previous variables in the parameters hash map, stored within the Web Service. Most variables in the Web Service is stored as Lists of variables, because most of these values can be stored as lists in some Web Service Discovery Mechanisms. Therefore most of these variables contained in the Web Service, implements a set method, that sets an entire list of variables into the Web Service, and an add method, that adds the single variable to the end of the current list of variables in the Web Service. These two methods will have to cooperate and work seamlessly alongside each other. Due to this, these functions will have to be tested thoroughly.

The final part of the repository test class adds a few more services to the repository, some are removed, and the test checks that the number of services in the repository is correct after these operations.

The translator test is a simple, yet important test. The test creates a WebService object, populates it with values, before it transforms it to a WS-Discovery Service. Then the class extracts the variables from the WS-Disocvery Service, and compares the values from the variables of the WS-Discovery service to those entered to the initial WebService.

The MarshallingTester tests the marshalling and unmarshalling functionality of the system. A WebService is created and populated with values. It is then parsed to XML string, before this string is then parsed back to a WebService object. The initial values are then compared to the values retrieved at the end of the test.

The PropertiesReader Test tests that the built in properties file and the class responsible for reading it works as intended. A testvarialbe is set in the config.properties file, the test then reads it and checks if it is the same as in the config.properties. A summary of all unit tests can be seen in Table 6.

Test Name	Description	Result
MarshallAndUnmarshalling()	Testing that the marshalling and unmarshalling function works	Success
readProperties();	Testing that the application can successfully read values from the config.properties file.	Success
ConstructorTest();	Testing that the constructor for the WebService class is working as intended.	Success
Parametertest();	Testing that the generic parameter hash table is working.	Success
repoTest();	Testig that the repository constructor and built in functions work.	Success
TranslateWSDtoService();	Testing that the translator can successfully transform a WebService to a WS-Discovery object	Success

Table 6 A quick summary of the Unit tests for the system and their result.

4.2 Functional tests

The second types of testing are the functional tests, these tests exists in order to ensure that the application is functioning as desired, and that it produces the desired result. All of the requirements given in chapter 0 will have to be evaluated and assessed through the use of functional tests. These tests can either be created in a similar manner to the unit tests, where you look at a part of the application, define adequate input and output values for it and then proceed by testing it. Another, simpler, yet still as effective approach to testing is to use the requirements defined for the application and simply use a checkbox “sheet” to evaluate the application against these requirements. The functional tests are designed to answer questions like: “Can the user do this?”, and is simply answered yes or no. It does however, not measure how or “how well” it is done, only if it can be done. Furthermore, there is no guarantee that it will still work when 10 or 100 simultaneous users try to perform the action.

In order to verify that the application is working as intended, functional tests can be applied to verify that everything works as it should. The execution of these test can be automated, but in smaller applications, it is easier to manually conduct them. The idea is the test the main functionality against a set of predefined criteria to ensure that everything works as expected.

A series of different functional tests for this application has been designed and run on the application.

- **Adding** a service. When a service is added through any connected node in the network, and sharing of this service is enabled, the service become visible to other nodes connected to the same network.
- **Removing** a service. Whenever a service is removed from the node that initially published the service it should also automatically be removed from all other nodes in the network. If a node that did not host a service decides to remove the service, it will be removed from the repository of this node, and will be unavailable for this node until a full refresh is conducted or the initial host changes the service, which again will trigger an updated version of the service to be broadcasted to all nodes in the network.
- **Updating** a service. A previously published service can be updated for a number of reasons, when a service is updated by the node that initially posted it, the updated version of the service will be pushed to all nodes and the updated data will appear in the repository and GUI of all nodes in the network. If a node that did not initially post the service, updates a service, the service will be updated in this nodes repository, but these changes will not be sent out through the network, these changes will stay like this until the original owner removes or updates the service.
- **Refreshing** a repository. Any node can, at any time, refresh their repository. This will delete all remote services in the repository, leaving only those who were initially published by the node itself. The node will then send an update request to all other nodes in the network, where it asks the other nodes to resend their local services to the other nodes in the network. The node will then receive updated data on all services available in the network. This can for example be done when a node joins an existing network, already containing several services and nodes. This function should however be used with care, as it generates quite a bit of network traffic.
- **Introducing a new node.** Whenever a new node joins an existing network of nodes, the new node will receive all new messages sent after the node has arrived. In order to receive a fully updated image of all services in the network, a refresh request must be sent.
- **Removing a node.** Whenever a node leaves the network, it is set up to send delete messages for all the services it is hosting. This works as intended if the exit button of the application is pressed. However, if the application is terminated or loses Internet connection, this message will not be sent to the other nodes, creating an availability problem.

If these six actions can be performed on a node, and the result is as described above, the application has met the requirements set for the functional tests. A simple matrix has been created in order to easily verify that the application meets the functional test requirements.

Test Number	Test name	Test description	Test success(T/F)
1.	Adding	Adding a service, checking if it appears on the other instances.	True
2.	Removing	Removing a service, checking if it disappears on the other instances.	True
3.	Updating	Updating a service, checking if the update appears on the other instances.	True
4.	Refreshing	Refreshing a repository, checking that it relists all the services from the other instances.	True
5.	Introducing	Whenever a new instance joins the network, any initial services this node holds, will appear in other repositories.	True

6.	Removing	Exiting a node, checking that its shared services disappears from the other instances.	True

Table 7 A list of the different functional tests.

4.3 Performance tests

The last type of testing, are the performance tests. These tests are usually the last tests to be conducted on a system, and the result accuracy depends on optimizations being completed before the tests start. The main idea behind these tests is to test the system under the load that it is assumed to experience in production environments. These tests can be automated, which makes them easier to conduct, able to run a high number of iterations as well as maintaining accuracy as the environment and execution is exactly the same every time.

The most important aspects of large-scale performance for this application are response times and function. The application must keep functioning even with a large number of nodes, approximately 25-50 individual nodes, all offering their own services as well as receiving service information from other nodes connected to the network. This is in accordance with the premises given in requirement number 15. This requirement does not imply the need of a specific test class, but rather visually confirmed when the application is deployed to multiple test nodes. The functional aspect measures that all data are received and delivered across the WAN mechanism and to the different instances of the application. If this requirement is not met, debugging must be performed to find the cause of data going missing or corrupt. The response time aspect is the reacting time of the application. How long time does it take from a service being added at a repository, and until it is visible in the other repositories. This is important to find out, as the application needs to be fast and contain reliable, updated data.

The performance tests are conducted using an internal java stopwatch function. This stopwatch is started when a function is called, and it will keep counting until the function has completed its job. All times are then written to a file that can be reviewed to compare and analyze the durations of the different functions. For this application, the WAN mechanism and the amount of traffic generated on the network when the number of nodes increase, is a particularly interesting topic that should be examined.

The next criterion is the response time of the application, a very relevant example is when a user refreshes his service repository. When the refresh button is pushed, all services that is not owned by the node, is deleted and an update request is sent to all the other connected nodes. When this request is received by the other nodes, they will resend all services they own to the node requesting the refresh. The requesting node will then receive services from all other nodes in the network, and add these to its own repository. The amount of time this takes is a very good measurement to determine the scalability of the application, as it involves all nodes in the network, adding services to the connected Web Service Discovery Mechanism, and adding and removing said services from the repository.

The main focus is to examine the response time of each node when performing different actions. The results are then compared and evaluated. Requirement number **Error! Reference source not found.** states that the finished prototype should support approximately 50 nodes, each sharing approximately 5 services.

The test for each environment is run at least 100 times each, as this ensures the integrity of the data and minimizes the impact of deviations. However, if the data set returned from the testing is inconclusive and shows no clear pattern of response time, additional test cycles are run. The test is then repeated until a sufficient data set is established.

4.4 Test environments

These tests are conducted in different environments and different settings before the results are compared and analyzed.

Through deployment in these three environments, all aspects of the application are tested and evaluated against its criteria for each environment.

An important part of the application is the modularity, this has been demonstrated through the implementation of two different WAN mechanisms, the experimental Mist and standardized AMQP (using ActiveMQ), connected to the application. Due to the fact that ActiveMQ is the only alternative of the two that is eligible for use in WAN networks without multicast support, the tests conducted in this chapter are all based on the ActiveMQ implementation.

4.4.1 Test tools

In order to gather as much information as possible, different test tools have been utilized. The response time tests are conducted using the built in Java StopWatch class is being used. It is capable of recording times as low as nanoseconds, making it a good tool to test application performance.

The packet capture part of the testing is done in Unix environments, using a bash build in function called tcpdump v.4.1, this application can be set to record packets for any interface and port, reducing the amount of unwanted data recorder.

The files generated by tcpdump is then analyzed using Wireshark V.1.12.4, this enables the user to view every packet sent, its source, destination and content. Filters can be applied to the tcp traces, filtering out only the most interesting data from the packet trace.

Another packet capture analyzer, captcp V1.9, has also been used, this is a command line application that offers statistical data for the entire trace.

All data in the following sections are recoded and retrieved using these tools.

4.4.2 Small scale testing

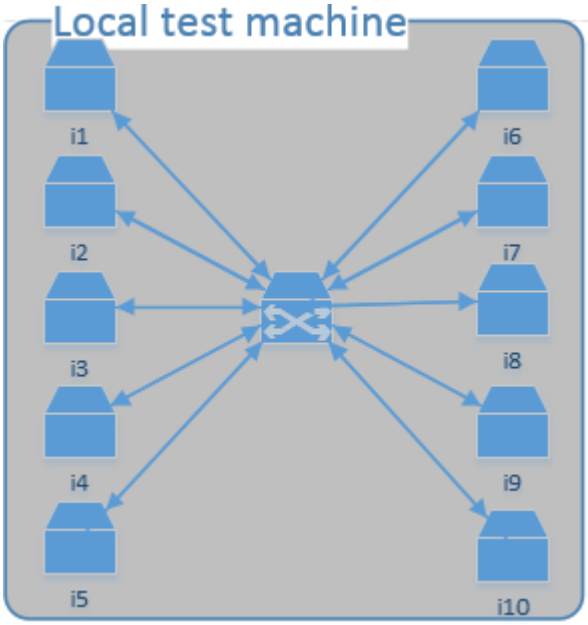

Figure 4 A schematic depicting the small scale test environment.

The small test environment is running on a single computer, the application is not sending any data across the network, but it uses the network adapter as a connector. The value of this data is therefore limited with regard to the network and large scale performance, but they fulfill another important aspect of the testing. It allows for testing the application in a more functional manner, thus identifying and resolving errors that could cause great delays in testing at later stages of the application testing. Due to this, the local, small scale testing is just as important for the development and testing process as the larger scale testing.

The small scale testing environment is set up with the following specifications, running ActiveMQ as WAN mechanism:

A laptop running Intel Core i5-3320M @ 2,6 GHz, 8 GB DDR3 RAM, Intel Centrino Wireless Adapter (N-300Mbps), Running Windows 7. As all instances of the application were running locally, there was no need to disable the firewall or forward ports to ensure the connection between the nodes. The application was deployed in ten different instances on the machine. Nine of these instances where headless nodes, only running as a background service with a predefined service deployed on them. The last node also featured a GUI, that enabled the user to monitor all services and nodes on the network, as well as adding, removing or altering services. Through this GUI, the application also monitors incoming and outgoing messages from the service. The time consumption for a full update, described in the section “Repository synchronization”, is timed and saved to a comma separated file. This file can then be opened in a spreadsheet to compare results and create graphs. The performance testing is very narrow for the small scale testing scenario, because the main intent is to test the function. The data will not be sent over any network connection, and the speed and capacity is therefore not very relevant. Another important point to take notice of is that all data sent over the WAN mechanism is treated equally by the WAN mechanism, whilst the repository routes and handles the messages. This is explained in the section “Repository synchronization” section, and involves that there is only WebService objects being sent, but with a different “status” parameter, which helps decide what the receiver should do with it. This means that the time a new service uses to travel from one node to another, is also the time a deleted service or an updated service will use.

Due to the fact that the first “level” of testing, is more functional related than performance related, the number test iterations has been greatly decreased. The table below contains the results for the first test conducted on the application.

Node/ Test	1	2	3	4	5	6	7	8	9	10	AVG:	Total Time
0	700	894	1025	1134	1254	1359	1454	1552	1652	1769	1279,3	1769
1	182	357	446	529	612	707	775	848	921	998	637,5	998
2	180	290	363	442	539	592	648	708	772	849	538,3	849
3	195	294	420	618	731	868	981	1085	1192	1313	769,7	1313
4	262	388	486	592	735	817	940	1048	1195	1290	775,3	1290
5	221	340	439	521	614	710	800	942	1038	1121	674,6	1121
6	120	196	366	468	552	633	724	815	903	1011	578,8	1011
7	229	396	482	562	657	745	840	931	1018	1098	695,8	1098
8	190	308	391	474	570	654	747	834	917	1002	608,7	1002
9	232	331	413	503	600	692	779	879	963	1052	644,4	1052
10	222	323	413	508	600	687	775	861	946	1028	636,3	1028
11	214	343	425	509	607	699	792	882	969	1057	649,7	1057
12	106	299	442	525	618	700	789	875	955	1053	636,2	1053
13	220	334	415	495	585	688	782	873	961	1050	640,3	1050
14	205	307	387	480	574	690	783	870	952	1039	628,7	1039

Table 8, results from the initial small scale performance test.

Table 8 contains the results from the first performance tests. The test was conducted with ten nodes, numbered from 1-10, the test was run 15 times, as seen in the Y column, where the tests are numbered from 0-14. All times in the table are in milliseconds.

The results from each test are written in a single row, the times written are the time elapsed from the update request was sent from the GUI node until the response is received from one of the headless nodes. The time is not reset between each new incoming response, as the time just keeps counting until all nodes have responded, and the test is complete.

The results for each test is summarized in the last two columns of the row, where the first column contains the average response time for all nodes. The last result is the total time from the refresh request was sent until all nodes had responded. As all the refresh requests and responses are performed in parallel, the total update time equals the response time of the slowest node.

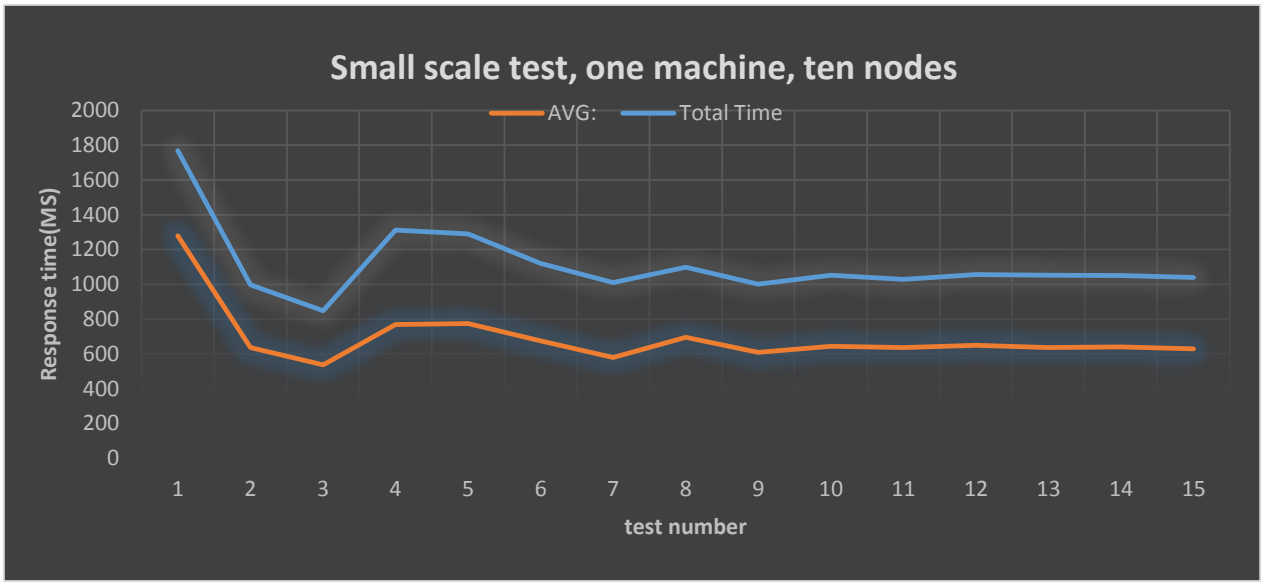

Figure 27. A graph summarizing the results from the first test.

As we can see from Table 8 and Figure 27 the response time of the first test is noticeability higher than the other tests. The later tests are more similar and at test number 9 the system is fully stabilized and the results from the test are very even. The average of the sums of “Total time” is 1115 milliseconds.

There could be several reasons for this run-in period to occur before the results stabilizes after a few tests, the three main plausible reasons are: The java hotspot compiler, establishing a network connection or loading the application in the computer memory/processor.

The java hotspot compiler optimizes and improves the most run sections of an application in order to ensure performance and minimize the load put on the host computer. However, the number of cycles these test run the application is not enough to improve the application in such a way.

The next option is the establishment of a network connection between the broker and the 10 clients. Even though each node registers with the broker on the startup of the application at startup, the nodes will need some time to get the transportation process started.

The last and by far most plausible explanation is the computer and its inner workings. Even though the application is already started and running when the first test is conducted, not all necessary resources are loaded into the system. When the resources have been loaded on the first test, they will be located in the computer RAM and as well in the processor cache. This enables the computer to reacssess the resources very quickly.

The functional tests, described in Table 7, were all successfully performed for this system.

4.4.3 Medium scale testing

The medium scaled test environment is included to assess both the challenges met in large scale testing, as well as the challenges met in small scale testing in the same test. The challenge from the small scale test scenario involves the more functional aspect, where connection between the nodes and successful message exchange. The large scale challenges includes testing the speed of the messages and updates between the different nodes.

The test environment is run on a multi-hop local network, which more closely resembles a full size network, it is also a very valuable asset when it comes to discovering and resolving issues related to the network connection part of the application. Since the full scale testing is a finite resource, it is crucial that the smaller scale tests are run prior to the full scale test, in order to ensure a smooth and efficient full scale test.

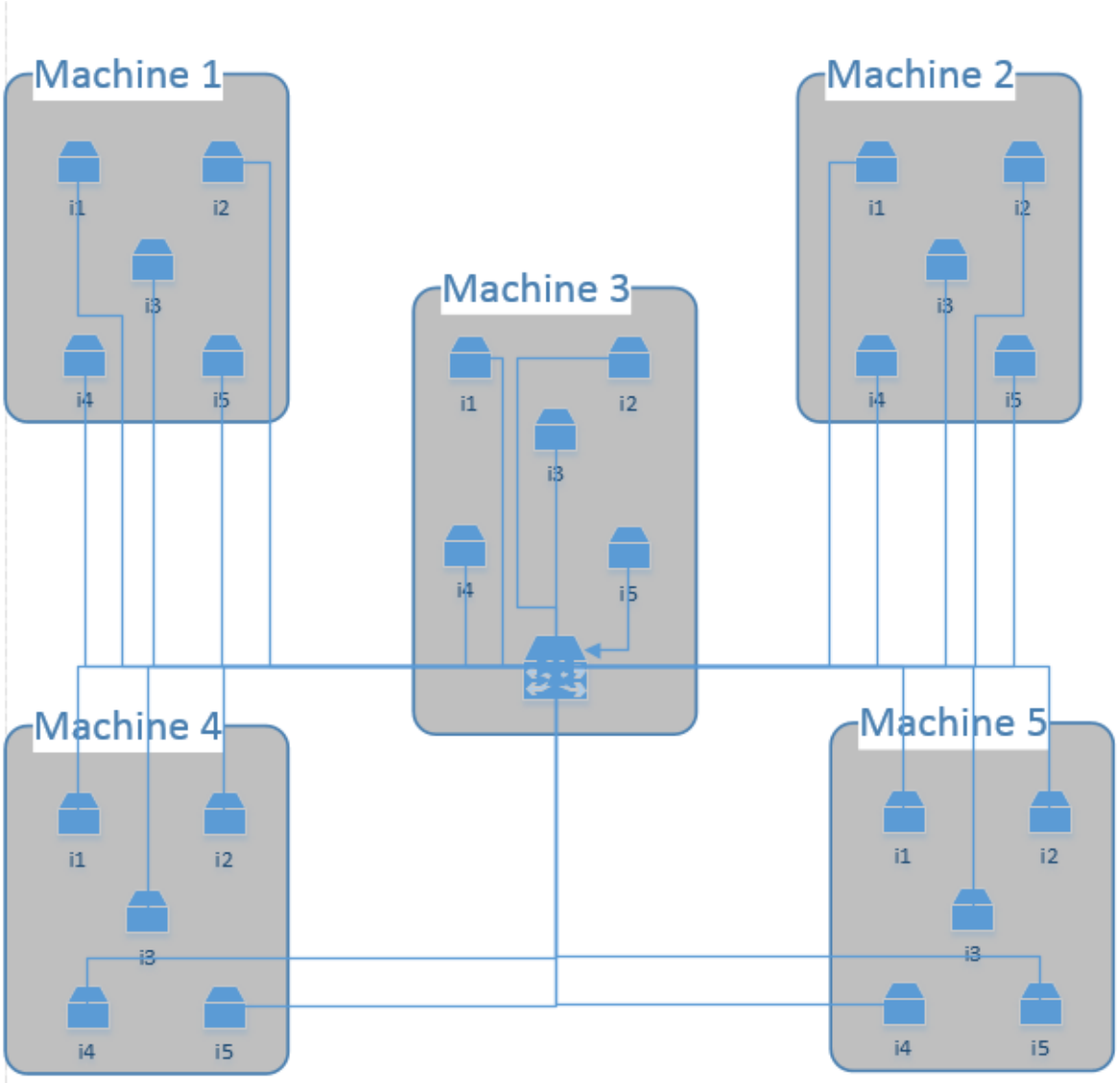

Figure 28 The setup for the medium scale test environment, 5 machines, 5 nodes on each.

The test are run with five different machines, each running five instances of the application. One of the machines will also host the broker, which will work as an intermediary for all communication between the nodes. The machines are a mix of laptop and desktop computers, running Windows 7, Windows 8 and Linux RedHat operating system. This ensures that the application works as intended on both Windows as well as Linux operating systems.

Since the communication between the different nodes now was across network boundaries, the firewalls of the respective clients would have to be opened for traffic on port 61616, as this is the default port for activeMQ packages. In order to test data transfer over the Internet, the port 61616 was also opened and forwarded to the machine hosting the broker. One of the nodes was then started setting the broker address as the server public IP address, and managed to successfully subscribe to the broker through the Internet address, which is good news for the full scale testing.

The tests conducted are the same as in the small scale testing, but in this environment the test was run approximately 200 times to ensure more reliable results. There was also a packet tracing application running for the full duration of the test, capturing all packets whose source or destination is port 61616. The full test data can be found in Appendix B.

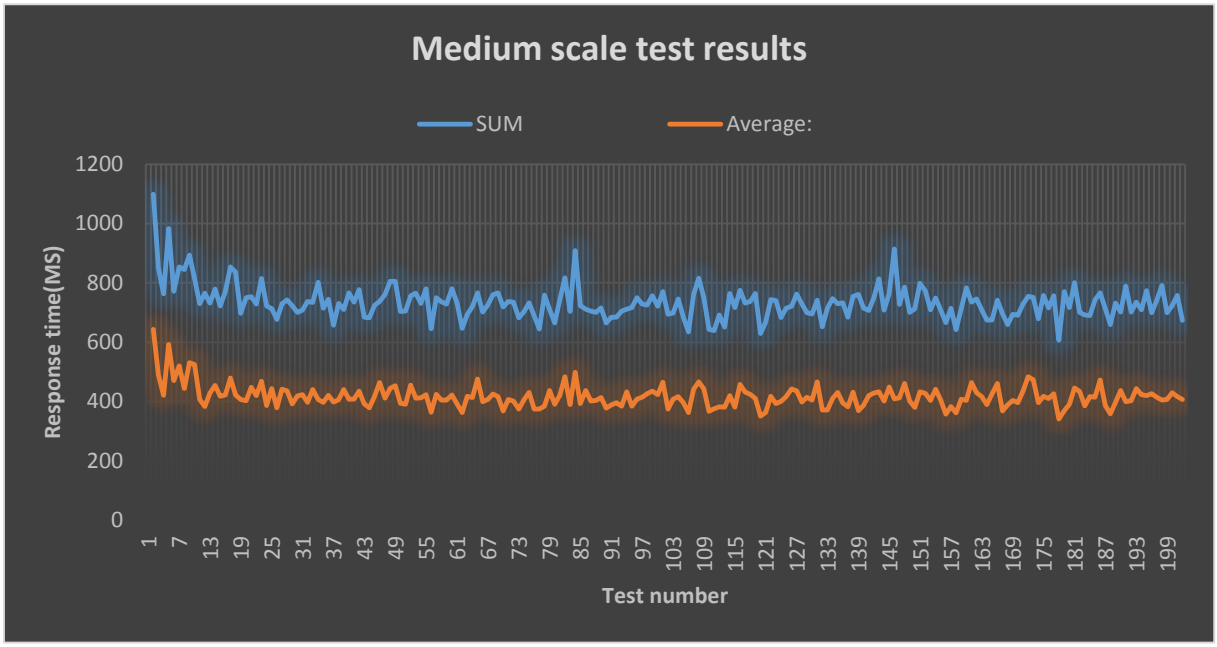

Figure 29 A graph displaying the test results from the medium scale testing.

Figure 29 depicts the results from the medium scale testing. The most obvious difference between this figure and Figure 27 is the number of entries, whilst Figure 27 is based on 15 tests, this graph contains data from approximately 200 tests. The total average of "Total time" is 734,11 milliseconds. If we compare the results from the small scale tests and the medium scale test, the average of total time, which is the time it took for the slowest node to answer, is 734,11 vs 1115 milliseconds. This means that the medium scale test scenario performed better than the small scale test scenario, even though it features more nodes and more instances. This means that on average, the nodes used 734,11 and 1115 milliseconds to reply to the message.

The reason for this increase in performance can be explained by the increased number of machines involved, and the decreased number of instances per machine. If a machine hosts ten instances , each of these ten instances will have to publish ten services to their connected Web Service Discovery Mechanism, this means that each machine hosts ten instances, which each hosts one Web Service Discovery Mechanism, which again hosts ten services each. When all these services are to be removed and/or updated at the same time, it creates a spike in CPU usage, which slows the overall speed of the system down. The average of each node is 8-10% CPU consumption, which will slow the computer down if a node hosts more than five instances. Distributing the system over multiple machines reduces the load on each machine, but increases the network load. Due to the packet tracing application running on both the medium scale and the large scale test, we can compare the network usage and application behavior in both environments.

Display

Display filter:	openwire
Ignored packets:	0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	616648	381873	61,927%	0	0,000%
Between first and last packet	3101,175 sec	3101,174 sec			
Avg. packets/sec	198,843	123,138			
Avg. packet size	302 bytes	375 bytes			
Bytes	186025213	143315717	77,041%	0	0.000%
Avg. bytes/sec	59985,394	46213,371			
Avg. MBit/sec	0,480	0,370			

Figure 30 The summary of the results from the packet trace application.

Figure 30 is the summary of the packet tracing application that was running on the machine hosting the broker during the experiment. It was started before the test and terminated after the test was done. The average network traffic is 0,48 MBit/sec for the node the application was running on, this number includes both up and down network traffic.

The summary also contains some other very interesting data about the packet traffic generated by the application and the WAN mechanism. A filter has been applied, filtering the packets on their protocol type. There are two types of packets in this packet trace, TCP packets and OpenWire packets. The TCP packets are those generated by the application and the messages, while the OpenWire packets are generated by the WAN mechanism as overhead.

In Figure 30, the numbers in the “Captured” column is for all packets, while the “Displayed” column is for the OpenWire packets. We can then see that the OpenWire packets make up for over 61% of the packets sent, and over 77% of the data sent. This is a very large overhead, which can be explained by the very low amount of data, and that the overhead size in MB will stay almost constant, regardless of the amount of data sent. This is good news, as it means the network application will scale very well, tests performed by ActiveMQ indicates that a throughput of over 20000 messages per second is possible [59].

The functional tests, described in Table 7, were all successfully performed for this system.

4.4.4 Large Scale Testing

The large scale tests was run in Nornet Core [60]. Nornet Core is a large scale WAN test bed incorporating nodes spread across Norway, Germany, China, USA and Sweden. All nodes are connected to the Internet via multiple Internet service providers and offers high connection speeds and stable performance. Each researcher is assigned a number of virtual machines which works as the different nodes of a network, communicating across the Internet.

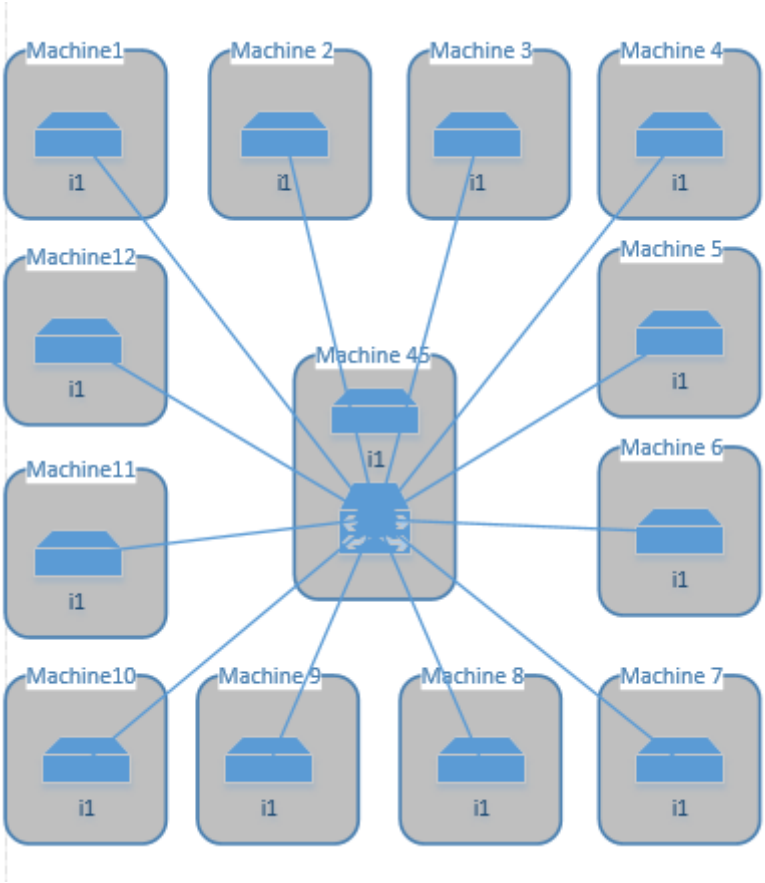

Figure 31 The large scale system consisted of 45 machines each running one node each, for demonstration purposes the number of machines have been reduced to 13 in the figure.

We were assigned 45 machines in Norway and Germany, the application was installed on each of these machines and a broker was started on one of the machines. When the broker was started, the application could be started on all the other nodes. When all the nodes were connected to the network, the packet tracing application was started, and at last the test was started. The test conducted was the same as in the medium scale test environment, where one of the node asked for updates at a regular interval. The test ran for 1000 cycles over approximately 4 hours.

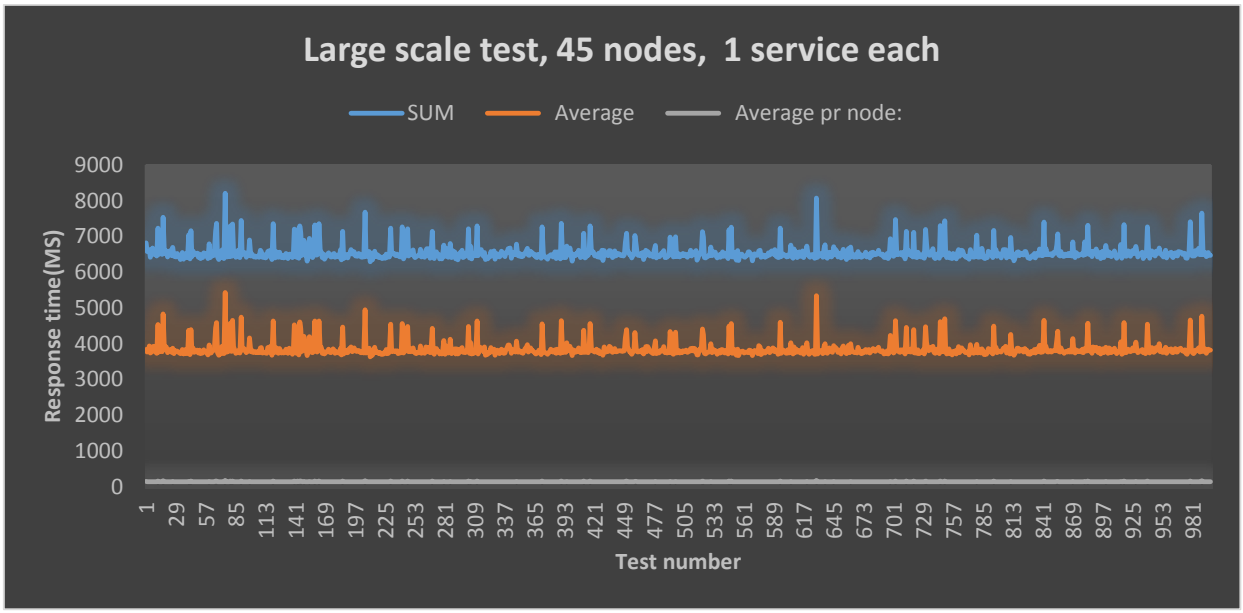

Figure 32 The test results from the Large scale testing.

The graph displayed in Figure 32 is a summary of the results from the large scale testing. A full refresh of the system, querying all nodes and receiving updated information about their services took on average 6,54 seconds and never more than 8,21 seconds.

If we look at the results from Figure 32 and compare them to the statements given in chapter 2.2.7 Response times as well as requirement number 16, the requirements has been fulfilled. At no point is the 10 second mark broken, and all possible requests made within the system is completed in less than 10 seconds. However, this is the most time consuming request that can be made, and during normal operation, this request should only be necessary during the initial startup, in order to fetch an updated view of all available services in the network.

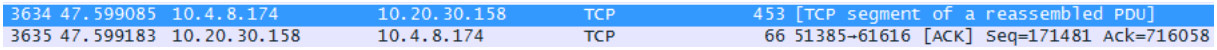

Figure 33 Captured packages going from the Broker and to a node running the application.

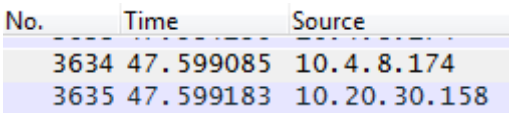

Figure 34 A enhanced caption of the timings of the packets depicted in Figure 33.

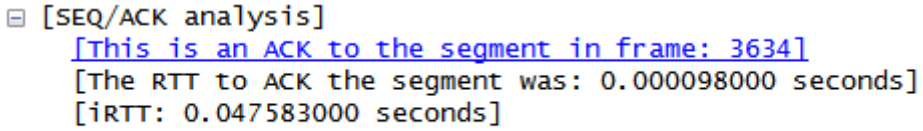

Figure 35 An expanded view of the content of packet number 363.

Figure 33 displays a sequence of a packets sent between the broker and a node connected to the network. In Figure 34 you can see an enlarged image of the timings in Figure 33. Packet number 3634 is the refresh request, and is sent after 47.599085 seconds, the next packet is the acknowledgement

for the previous package, this is timed at 47.599183 seconds, which means that the packet takes 0.000098 seconds from sending to acknowledgement, this can also be written as 98 microseconds.

This means that when a user adds a new service in their instance of the application, this service appears in all the other instances of the application within an acceptable timeframe. This number will not be affected by the number of nodes, and as long as the network capacity is good enough it will remain very fast.

As the tcp packet capture application was running on the full scale tests as well, a comprehensive amount of data from the packet traffic during the test has been logged. Initially, we will compare the summary of the packet capturing with the summary from the medium scale tests.

Display

Display filter:	openwire
Ignored packets:	0 (0,000%)

Traffic	Captured	Displayed	Displayed %	Marked	Marked %
Packets	444530	168107	37,817%	0	0,000%
Between first and last packet	15098,005 sec	15097,952 sec			
Avg. packets/sec	29,443	11,134			
Avg. packet size	272 bytes	267 bytes			
Bytes	120817840	44880906	37,148%	0	0.000%
Avg. bytes/sec	8002,239	2972,649			
Avg. MBit/sec	0,064	0,024			

Figure 36 Summary of the packet capture data from the large scale tests.

The duration of the large scale testing is roughly 5 times as long as the medium scale tests, although the test data from both tests are quite stable, and it is reasonable to assume that a longer duration on the medium scale test would not have yielded any different results.

The average packet size is slightly higher on the medium scale test compared to the full scale tests, this is due to the way ActiveMQ works. Because the medium size test system is running 5 instances, each instance hosting one service each, but all on the same IP address, whilst the large scale testing scenario is running 45 instances, each hosting on service, but all instances on different IP addresses. ActiveMQ regards acknowledgements as TCP traffic, and since the number of acknowledgements is higher, the average packet size is lower. This is due to the small average size of acknowledgement packets.

Display

Display filter:
tcp.flags == 0x0010

Traffic	Medium Scale tests Captured	Displayed	Displayed %	Large Scale tests Captured	Displayed	Displayed %
Packets	616648	200152	32,458%	444530	127644	28,714%
Between first and last packet	3101,175 sec	3101,137 sec		15098,005 sec	15097,957 sec	
Avg. packets/sec	198,843	64,541		29,443	8,454	
Avg. packet size	302 bytes	100 bytes		272 bytes	228 bytes	
Bytes	186025213	19986250	10,744%	120817840	29155860	24,132%
Avg. bytes/sec	59985,394	6444,815		8002,239	1931,113	
Avg. MBit/sec	0,480	0,052		0,064	0,015	

Figure 37 Statistics for the large scale tests compared to the medium scale tests, filtered to show only acknowledgements.

Figure 37 is a comparison of the statistics summary for the large scale tests, compared to the medium scale tests. The “Captured” column is the entire dataset from the packet trace, while the “Displayed” column is a filter that filters to show only acknowledgement packets.

For both tests, the number of acknowledgement packets is approximately 30%, percent of the packets sent, but in the large scale tests, the acknowledgement packets are on average 228 bytes in comparison to 100 bytes on the medium scale tests. Due to this, the acknowledgements take up only a little over 10% of the medium scale tests bandwidth, but almost 25% of the large scale bandwidth. This portrays the increased reliability of a local network over the Internet.

	Medium scale			**Large scale**			
	UP	**DOWN**	**SUM**	**UP**	**DOWN**	**SUM**	
Number of Packets	14633,90	13954,57	28588,48	5022,67	4855,78	9878,44	Packets
Total bytes sent	6348290,71	2458231,33	8806522,05	1612741,22	1072099,67	2684840,89	bytes
Application layer throughput	14249,84	4491,21	18741,05	678,78	396,93	1075,71	bit/s
Retransmissions	37,52	21,52	59,05	80,53	20,42	100,96	Packets
Retransmissions %	0,2563%	0,1542%	0,2065%	1,6033%	0,4205%	1,022%	Percent

Table 9 A comparison chart of the traffic per node in the medium scale test compared to the large scale test.

is a comparison chart of the amount of packets and data sent per node in the medium scale test compared to the large scale test. As discussed in the section prior to the table, total bytes sent is approximately 3 times higher than in the large scale tests. The sum of Total bytes per packet is 8806522 for the medium scale, versus 2684840 bytes per packet for the Large scale tests. 8806522/2684840=3,28. The average medium scale test node, produced 3,28x more data than the average large scale test node, but it holds 5 times as many services.
The table also contains the average number of packet sent and the average number of retransmissions per node, if we compare these numbers for the Medium scale and the Large scale tests. The medium scale tests are run in a local network, and therefore there is a lower percentage of packet loss, this is perfectly normal, as a local network is more stable and reliable than a WAN like

the Internet. This also leads to the fact that the number of packets sent is not three times as many, but 2,89 (28588/9878) times as many.

The functional tests, described in Table 7, were all successfully performed for this system.

5. Conclusion and future work

5.1 Conclusion

Through this master's thesis the entire process from idea to a finished prototype of a Web Service Discovery federation mechanism has been documented and discussed. The aim was to create a federation mechanism that could enable two or more different Web Service Discovery Mechanisms to communicate over a WAN like the Internet. The intended use is in a coalition network consisting of NATO and PfP member nations. Assuming each nation deploys one gateway towards the federation WAN, the application implementing the federation mechanism had to scale to 50 nodes. The application should also be modular, meaning that any part of the application could easily be swapped for another part performing the same job.

- The application implements a repository that can store information about Web Services. The Web Services can be added through a configuration file or through the GUI of the application. The application does not support retrieving information from the connected Web Service Discovery Mechanism, the information about a service will not change in the repository if a change is done in the connected Web Service Discovery Mechanism.
- The different instances of the software each holds their own repository, the information in each repository is synchronized between the repositories.
    - An update/refresh function has been implemented. This enables a node to ask all the other nodes for updated information. This is implemented as a publish/subscribe mechanism, all nodes are listening to the broker, if a node wants an update, it sends an update request to the broker which is then forwarded to the other nodes. In the federation mechanism the AMQP framework ActiveMQ was used.
    - There is also a listener function that will look for changes in the repository, when a change is detected, an updated version of the changed service is sent to all other repositories, and then forwarded to the connected Web Service Discovery Mechanism of the repositories. This is also a publish/subscribe mechanism.
- During the test phases there are three different scenarios that have been run.
    - A small scale scenario, running on a single machine, with ten instances of the software connected to each other.
    - A medium scale scenario, where five physical machines were running five instances of the software each. One of the machines was also hosting the ActiveMQ broker, which was used as a hub for the communication between the nodes.
    - A large scale scenario, where 45 different physical machines, each running one instance of the application, communication via a WAN. One of the machines was also hosting the ActiveMQ broker.
- The tests run on the system were:
    - Unit tests which has been run during development.
    - Functional tests, which has been run in all the test scenarios.
    - Performance tests, which has been run in all test scenarios.
    - Network traffic has been logged in the medium and large scale tests.

The application fulfills the requirements listed in section 0, and is working as intended. The application is intended to be easy to use, and through the performance tests it has been shown that it falls within the responsiveness frame expected of an interactive application. But, in order to take advantage of the application like it is intended to be used, knowledge about how to create and

publish a Web Service is required. In conclusion, the application fulfilled the requirements stated and the thesis goal was reached.

5.2 Future work.

The application is working as intended, but in order to make it a fully federated system it must be expanded. The most important feature to achieve full federation, is to enable retrieving data from the connected Web Service Discovery Mechanism. As the application is today, data can be added or deleted from the connected Web Service Discovery Mechanism. In addition to these features, the application must be able to subscribe to changes made in the connected Web Service Discovery Mechanism, and updated the services in the repository accordingly. This also means that a service added through the Web Service Discovery Mechanism, should automatically appear in the repository. When this feature is implemented, full federation will be achieved.

When the repository will automatically receive data from the connected Web Service Discovery Mechanism, a slight change to the WebService ID function must be made, as this has to be settable to the ID that the service had or has in the connected Web Service Discovery Mechanism. However, both WS-Discovery and UDDI uses the UUID format for ID, which is already supported in the application.

As well as the prior changes, more Web Service Discovery Mechanisms could also be implemented. This will make it more interesting to use for different parties. This is more urgent than new WAN mechanisms, as the application already implements two different WAN mechanisms, while it only supports a single Web Service Discovery Mechanism. It is also possible to expand the number of features that the application supports, for example it could be possible to set up hot swapping of the WAN mechanism or the Web Service Discovery Mechanism. This would enable the user to change to a different Web Service Discovery Mechanism while the application is running, given that both the new and the old mechanism had already been implemented through an interface. With this approach, the application would become easier to use and the different branches implementing different mechanisms could be merged into a very handy tool for anyone wanting to test or connect many different Web Service Discovery Mechanisms or WANs.

Finally, when using AMQP as the WAN mechanism one should consider a multi-brokered topology setup for redundancy. In the conformance and performance tests a single broker setup was used, and this constitutes a single point of failure. However, this was done to evaluate the worst case scenario of a single broker having to handle all the publish/subscribe traffic. The broker performed admirably and within the required timeliness constraints. In a multi-broker setup one could expect both resilience towards partial failure as well as potentially even lower response times due to the traffic being distributed across multiple brokers.

Figures:

Figure 1 Multiple independent LANs connected via a WAN 9
Figure 2. The focus of this task is to develop the federation mechanism that you can see inside the red circles, this is a utility enabling interconnection between different Web Service Discovery Mechanisms. 16
Figure 4. The SOA Triangle, with the service contract in the center [72]. 17
Figure 5. WS-Discovery implemented in ad-hoc mode without discovery proxy. [28] 23
Figure 5. Federation of two independent WS-Discovery (WS-D) domains [32] 32
Figure 6. A schema depicting an overall view of the project design. 37
Figure 7. A general overview of the components in the system, the mechanism specific components are marked with a star. 41
Figure 11 The variables of the WebService class 43
Figure 12 The last variable of the WebService classis an custom made enum. 43
Figure 10 A network of three instances, each holding 4 services in their repository. 44
Figure 11. Adding a new service through the connected discovery mechanism. 46
Figure 12. Adding a new service through the WAN mechanism 48
Figure 13. A diagram depicting removing a service from the connected discovery mechanism 49
Figure 14. When a service is removed from another node and the local node receives a notification about it. 50
Figure 15. Automatic timer for repository update. 50
Figure 16. User initiated repository update. 52
Figure 17 Automatic and manual Discovery Mechanism update 53
Figure 18. A simple visual representation of the Discovery Mechanism Interface 54
Figure 19. Translator, responsible for translating services between the DM and repository. 54
Figure 20. Repository Interface, responsible for saving all Web Services. 55
Figure 21. The management interface is the graphical user interface of the system. 56
Figure 22. The WANMechanismInterface is the intermediary between the system and the WAN mechanism. 56
Figure 23 The application GUI. 58
Figure 24 The implemented workings of adding a web service. 59
Figure 25 The results from the JUnit tests. 63
Figure 29 A schematic depicting the small scale test environment. 69
Figure 27. A graph summarizing the results from the first test. 71
Figure 28 The setup for the medium scale test environment, 5 machines, 5 nodes on each. 72
Figure 29 A graph displaying the test results from the medium scale testing. 73
Figure 30 The summary of the results from the packet trace application. 74
Figure 31 The large scale system consisted of 45 machines each running one node each, for demonstration purposes the number of machines have been reduced to 13 in the figure. 75
Figure 32 The test results from the Large scale testing. 76
Figure 33 Captured packages going from the Broker and to a node running the application. 76
Figure 34 A enhanced caption of the timings of the packets depicted in Figure 33. 76
Figure 35 An expanded view of the content of packet number 363. 76
Figure 36 Summary of the packet capture data from the large scale tests. 77
Figure 37 Statistics for the large scale tests compared to the medium scale tests, filtered to show only acknowledgements. 78
Figure 41 A Basic single broker to N nodes ActiveMQ implementation. 94

Figure 42 A ActiveMQ broker to broker network setup. 95
Figure 43 A ActiveMQ network with Master and Slave Brokers. 96

Tables:

Table 1 The different fields of the WSDL document. 20
Table 2 – Comparison chart for different Web Service Discovery Mechanisms. 25
Table 3 WAN Mechanism Evaluation 30
Table 4 A summary of the requirements for the system. 36
Table 5 The different names and datatypes of the new WebService class, WS-Discovery and UDDI. . 40
Table 6 A quick summary of the Unit tests for the system and their result. 65
Table 7 A list of the different functional tests. 66
Table 8, results from the initial small scale performance test. 70
Table 9 A comparison chart of the traffic per node in the medium scale test compared to the large scale test. 78
Table 10 88
Table 11 The different variables set to each service in WS-Discovery 90
Table 12, The list of variables and elements from the technical specification [26]. 92

References

[1] Dictionary.com, "Definition of Federation," [Online]. Available: http://dictionary.reference.com/browse/federation. [Accessed 09 01 2015].

[2] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, "Web Services Architecture," 11 February 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/. [Accessed 23 September 2014].

[3] C. Janssen, "Techopedia - Wide Area Network (WAN)," Techopedia, [Online]. Available: http://www.techopedia.com/definition/5409/wide-area-network-wan.

[4] I. A. Education, "Service Orientation Design Principles," Arcitura, [Online]. Available: http://serviceorientation.com/serviceorientation/index. [Accessed 27 01 2015].

[5] S. Denning, "The Springboard," Boston, London, Butterwhort Heinmann, 2000, pp. chapther 4-7,11-12..

[6] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young and P. J. Denning, "Computing as a discipline," *Communications of the ACM,* pp. 9-23, 1 Jan 1989.

[7] GNU, "GNU Public Licence," [Online]. Available: https://www.gnu.org/licenses/licenses.html.

[8] P. Bartolomasi, T. Buckman, A. Campbell, J. Grainger, J. Mahaffey, R. Marchhand, O.Kruidhof, C. Shawcross and K. Veum, "NATO network enabled capability feasability study," 2005.

[9] J. Eckert, "Answering Questions on the Future Mission Network," 29 March 2012. [Online]. Available: http://www.act.nato.int/article-2013-1-16.

[10] P.-P. MEILER, Writer, *SOA at the tactical level - NATO IST-118 Approach.* [Performance]. NATO, 2013.

[11] F. T. Johnsen and T. Hafsøe, "Experiments with web services at combined endavour. Paper ID 002," 2010. [Online]. Available: http://www.dodccrp.org/events/15th_iccrts_2010/papers/002.pdf.

[12] NATO, "NATO ACT TIDE," NATO, 17 October 2014. [Online]. Available: http://www.act.nato.int/tide. [Accessed 27 Jan 2015].

[13] K. Lund, A. Eggen, D. Hadzic, T. Hafsøe and F. T. Johnsen., "Using Web Services to realize Service Oriented Architecture in military communication Networks.," *IEEE Communications Magazine ,* no. October, 2007.

[14] R. T. Group, "SOA Recommendations for Disadvantaged Grids in the Tactical Domain (IST-118)," NATO, [Online]. Available: https://www.cso.nato.int/ACTIVITY_META.asp?ACT=2293.

[15] M. Gudgin, M. Hadley, N. Mendelsohn, Jean-Jacques, H. T. Nielsen, A. Karmarkar and Y. Lafon, "SOAP Verison 1.2 W3C reccomandation," 27 April 2007. [Online]. Available: http://www.w3.org/TR/soap12/. [Accessed 24 April 2015].

[16] H. Haas and A. Brown, "Web Services Glossary," 11 Feb 2004. [Online]. Available: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice.

[17] T. Bray, J. Paoli, C. Sperberg-McQueen and F. Y. Eve Maler, "Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation," 26 November 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/.

[18] A. DuVander, "JSON'S EIGHT YEAR CONVERGENCE WITH XML," www.programmableweb.com, 26 Dec 2013. [Online]. Available: http://www.programmableweb.com/news/jsons-eight-year-convergence-xml/2013/12/26. [Accessed 05 Feb 2015].

[19] J. Schneider, T. Kamiya, D. Peintner and R. Kyusakov, "Efficient XML Interchange (EXI) Format 1.0 (Second Edition)," 11 February 2014. [Online]. Available: http://www.w3.org/TR/2014/REC-exi-20140211/. [Accessed 14 October 2014].

[20] T. Hafsøe, F. T. Johnsen, K. Lund and A. Eggen, "Adapting Web Services for limited bandwidth tactical networks.," in *12th international Command and Control Research and Technology Symposium*, Newport, RI, USA, 2007.

[21] Oracle, "Sun Java System Message Queue 4.1 Developer's Guide for Java Clients- The SOAP message object," 2010. [Online]. Available: http://docs.oracle.com/cd/E19717-01/819-7757/aeqfi/index.html.

[22] R. Chinnici, J.-J. Moreau, A. Ryman and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0," 26 June 2007. [Online]. Available: http://www.w3.org/TR/wsdl20/. [Accessed 01 October 2014].

[23] F. Zhu, M. W. Mutka and L. M. Ni, "Service Discovery In Pervasive Computing Environments," 2005. [Online]. Available: http://www.informatik.hs-furtwangen.de/~hanne/Pervasive/ServiceDiscovery.pdf.

[24] L. Clement, A. Hately, C. von Riegen and T. Rogers, "UDDI Version 3.0.2," 19 July 2002. [Online]. Available: http://www.uddi.org/pubs/uddi-v3.0.2-20041019.htm.

[25] A. Grangard, B. Eisenberg and D. Nickull, "ebXML Technical Architecture Specification v1.0.4," 16 February 2001. [Online]. Available: http://www.ebxml.org/specs/ebTA.pdf.

[26] V. Modi and D. Kemp, "OASIS Web Services Dynamic Discovery 1.1," 1 July 2009. [Online]. Available: http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html.

[27] F. Johnsen, J. Flathagen, T. Gagnes, R. Haakseth, T. Hasføe, J. Halvorsen, N. Nordbotten and M. Skjegstad, "Web Services And Service Discovery," FFI -2008/01064, Lillestrøm, 2008.

[28] F. Johnsen, J. Flathagen, T. Gagnes, R. Haakseth, T. Hasføe, J. Halvorsen, N. Nordbotten and M. Skjegstad, "Web Services and Service Discovery - FFI rapport 2008/01064," 2008. [Online]. Available: http://www.ffi.no/no/Rapporter/08-01064.pdf. [Accessed 2015].

[29] "Internet Systems Consortium - DHCP," [Online]. Available: https://www.isc.org/downloads/dhcp/.

[30] "What is DNS?," [Online]. Available: http://www.webopedia.com/TERM/D/DNS.html.

[31] "What is a domain controller?," [Online]. Available: http://www.webopedia.com/TERM/D/domain_controller.html.

[32] "Denial of Service Attack," [Online]. Available: http://www.webopedia.com/TERM/D/DoS_attack.html.

[33] "Distributed Denial of Service," [Online]. Available: http://www.webopedia.com/TERM/D/DDoS_attack.html.

[34] S. Cheshire and M. Krochmal, "Multicast DNS - Memo," February 2013. [Online]. Available: http://tools.ietf.org/html/rfc6762. [Accessed Sept 2014].

[35] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.," Microsoft Research Ltd, St. George House, Houston, Texas.

[36] Y. Chen, "Distributed hash table based routing algorithm for wireless sensor networks.," 2014. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6977632. [Accessed 2015].

[37] TechTerms.com, "Peer to Peer," [Online]. Available: http://techterms.com/definition/p2p. [Accessed 2015].

[38] "FreePastry," [Online]. Available: http://www.freepastry.org/.

[39] B. Y. Zhao, J. Kubiatowicz and A. D. Joseph, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing," Computer Science Division University of California, Berkeley, California, 2001.

[40] G. Tribhuvan, "A BRIEF INTRODUCTION AND ANALYSIS OF THE GNUTELLA PROTOCOL(??)," University of Freiburg, Freiburg.

[41] "Gtk-Gnutella," [Online]. Available: http://gtk-gnutella.sourceforge.net/en/?page=news.

[42] G. T, "A robust and scalable peer-to-peer publish/subscribe mechanism," *Communications and Information Systems Conference (MCC), 2012 Military ,* pp. 1-6, 2012.

[43] [Online]. Available: https://www.fkie.fraunhofer.de/.

[44] J. Fernandes, I. Lopes, J. Rodrigues and S. Ullah, "Performance and evaluation of RESTful web services and AMQP protocol," IEEE, Da Nang, 2013.

[45] M. Skjegstad, F. T. Johnsen, T. H. Bloebaum and T. Maseng, "Mist: A Reliable and Delay-Tolerant Publish/Subscribe Solution for Dynamic Networks," IEEE, 2012.

[46] O. C. Christoph Schrot, "Brave New Web: Emerging Design Principles And Technologies as Enablers of a Global SOA," SAP Research CEC, St. Gallen, 2007.

[47] F. T. Johnsen, T. H. Bloebaum and K. Lund, "Enabling service discovery in a federation of systems: WS-Discovery case study FFI-rapport 2014/1454," Norwegian Defence Research Establishment, Lillestrøm, 2014.

[48] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middelware)*, Heidelberg, Germany, 2001.

[49] A. Rowstron, A.-M. Kermarrec, M. Castro and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure.," in *NGC2001, UCL*, London, 2001.

[50] F. T. Johnsen, "Pervasive Web Service Discovery and Invocation in Military Networks," FFI - report number: 2011/00257, Kjeller, 2011.

[51] J. F. T. H. M. S. N. K. Frank T. Johnsen, "Interoperable service discovery: Experiments at combined Endavour 2009," FFI- 2009/01934, Lillestrøm, 2009.

[52] C. 7. N. J Busch, "NATO NNEC Core Enterprise Services," NATO, 2009.

[53] NATO, "The SOA baseline profile Version 1.7," NATO, 2011.

[54] "Apache Log4j," [Online]. Available: http://logging.apache.org/log4j/2.x/.

[55] Tutorialspoint, "SDLC - Overview," Tutorialspoint, 2015. [Online]. Available: http://www.tutorialspoint.com/sdlc/sdlc_overview.htm. [Accessed 13 04 2015].

[56] L. Luo, "Software Testing Techniques," [Online]. Available: http://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf. [Accessed 2015].

[57] ActiveMQ, "ActiveMQ Performance," Apache, [Online]. Available: http://activemq.apache.org/performance.html.

[58] [Online]. Available: https://www.nntb.no/nornet-core/.

[59] J. Nielsen, "Usabilit Engineering," 1993. [Online]. Available: http://www.nngroup.com/books/usability-engineering/.

[60] J. Nielsen, "Response time: The 3 important limits," Nielsen Norman group, 1993. [Online]. Available: http://www.nngroup.com/articles/response-times-3-important-limits/.

[61] M. Skjegstad, "java-ws-discovery," November 2011. [Online]. Available: https://code.google.com/p/java-ws-discovery/. [Accessed October 2014].

[62] "ActiveMQ," [Online]. Available: http://activemq.apache.org/.

[63] R. J. Rob Godfrey, "AMQP," OASIS, 2014. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=amqp. [Accessed 04 2015].

[64] [Online]. Available: http://activemq.apache.org/kahadb.html.

[65] [Online]. Available: http://www.computerhope.com/jargon/m/mesh.htm.

[66] F. Johnsen and T. Hafsøe, Artists, *Føderert service Discovery.* [Art]. FFI, 2014.

[67] M. Erik Christensen, I. R. Francisco Curbera, M. Greg Meredith and I. R. Sanjiva Weerawarana, "www.w3.org," March 2001. [Online].

[68] T. Buckman, "NATO network enabled capability feasibility study," October 2005. [Online]. Available: http://www.dodccrp.org/files/nnec_fs_executive_summary_2.0_nu.pdf.

[69] M. z. Muehlen, J. V. Nickerson and K. D. Swenson, "Developing web services choreography standards—the case of REST vs SOAP," 29 9 2005. [Online]. Available: http://ac.els-cdn.com/S0167923604000612/1-s2.0-S0167923604000612-main.pdf?_tid=5abf23ae-38e6-11e4-85e0-00000aab0f01&acdnat=1410352454_be1428b8f4b78b6a4959604f6887f057.

[70] R. Schmelzer, Artist, *Soa Triangle.* [Art]. http://www.zapthink.com/2004/05/28/how-to-think-loosely-coupled, 2004.

[71] T. N. D. R. Establishment, "Om FFI," 03 03 2014. [Online]. Available: http://www.ffi.no/no/Om-ffi/Sider/default.aspx. [Accessed 29 10 2014].

[72] C. v. Riegen, T. Bellwood, D. Ehnebuske, Y. L. Husband, A. Karp, K. Kibakura, J. Lancelle, S. Lee, S. MacRoibeaird, B. McKee, T. Nordan, D. Rogers, C. Tomlinson and C. Tosun, "UDDI Version 2.03 Data Structure Reference," 19 July 2002. [Online]. Available: http://www.uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm. [Accessed 5 December 2014].

[73] A. Thuen, Artist, *WS-Federation Overview.* [Art]. UiO, 2014.

[74] P. Mag, "PC Mag Encyclopedia," PC Mag, [Online]. Available: http://www.pcmag.com/encyclopedia/term/51181/service-discovery. [Accessed 08 01 2015].

[75] H. H. M. N. ,. C. ,. F. D. O. David Booth, "Web Services Architecture," W3C, 11 February 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/. [Accessed 6 February 2015].

[76] [Online]. Available: https://www.nntb.no/nornet-edge/.

[77] E. G. Peixiang Liu, "Recursive Routing in the Cognitive Packet Network," 2007. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=4444727&queryText%3Drecursive+routing.

Appendices

Appendix A – List of abbreviations

Table 10 List of abbreviations

Abbreviation	Full name
C2	Command and Control
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance
COTS	Commercial off the shelf
DDOS	Distributed Denial Of Service
DHT	Distributed Hash Table
Disadvantaged grids	Communication grids that are disadvantaged by line-of-sight connections, low bandwidth, intermittent availability, etc.
DOS	Denial Of Service
ebXML	Electronic Business using eXtensible Markup Language
EXI	Efficient XML Interchange
FMN	Federated Mission Networking
GUI	Graphical User Interface
IST-118	NATO work group on SOA Recommendations for Disadvantaged Grids in the Tactical Domain
LAN	Local Area Network
LANs	Local Area Networks
MAN	Metro Area Networks
MANET	Mobile Ad Hoc Network
MANETs	Mobile Ad Hoc Networks

MC	Military Committee
mDNS-SD	Multicast Domain Name System- Service Discovery
MN	Mission Network
NAC	North Atlantic Council
NATO	North Atlantic Treaty Organization
NBD	Network based defense.
NNEC	NATO Network Enabled Capability
QOS	Quality of Service
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TIDE	Technology for Information, Decision and Execution Superiority
UDDI	*Universal Description, Discovery and Integration*
W3C	World Wide Web Consortium
WAN	Wide Area Network
WS-D	Web Service Discovery
WSDL	Web Service Description Language
WS-I	Web Service - Interoperability
XML	Extensible Markup Language

Appendix B – Technology Basis

Technology basis

There are some frameworks and standards that are more important for the workings of the system than other. In this section you can find a thorough review and explanation of the most essential external software used as a part of the application.

Java-WS-Discovery

The Java-WS-Discovery release [61] is an implementation that aims to follow the WS-discovery standard [27]. It is an open source project, which benefits it by having a rather large user group. This contributes to the implementation through a lot of valuable feedback and reports from the community.

The Java-WS-Discovery implementation supports SOAP over UDP and a proprietary SOAP over UDP using EXI or Gzip. The OASIS WS-discovery standard only contains guidelines for SOAP over UDP, however the SOAP over UDP with compression may come in handy for disadvantaged grids, as it reduces the payload size and adds checksums to the transmitted content. This makes the the transmission less prone to errors and more bandwidth efficient.

The backbone of the Java-WS-Discovery is the server, this server can as previously mentioned be set to work as either a proxy server, that forwards and redirects the queries sent, or in ad-hoc mode. When in proxy mode it sends a probe out to all its known services, this probe informs the nodes of the new server. And encourages all nodes in the network to switch from multicast to unicast probe messages. In ad-hoc mode, each service that is deployed registers with a local server that does not have a specific endpoint address, it just runs in a separate thread on the local machine. All probes are sent multicast to announce changes to other nodes in the network.

A service registered in WS-Discovery can be described by multiple parameters, *Table 10* contains the name of each service and a description summarizing the meaning of the parameter.

Name	Description
UUID	Universally unique identifier(128 bit unique value)
portType	Service operation- Input and output(LIST)
scope	Service scope, used for matching services and their general use.(LIST)
xAddrs	Address to the implementation endpoint. (LIST)
endpointReference	A String containing the endpoint reference
endpointReferenceType	An EndpointReferenceType Object, containing data about address, portType and name.

Table 11 The different variables set to each service in WS-Discovery

Whenever a service is deployed to a local type of WS-Discovery server it registers with the server by calling the deploy method of the server, this again requires a reference to the server object. It is also possible to create a new server that contains just the one service, but this is neither very efficient nor scalable. If a service is to register with a WS-discovery proxy server this requires the service to send a multicast Hello message, which the server is required to be listening for. The server will then publish the service using all the necessary information contained in the Hello message. The server will list the service until it is removed by a Bye message. A potential problem in a dynamic network may be the liveness problem, where the service is listed by the proxy, but not available for invocation.

UDDI

This is the introduction from the UDDI v2.03 data spec [26]*:*

-"The programmatic interface provided for interacting with systems that follow the Universal Description Discovery & Integration (UDDI) specifications make use of Extensible Markup Language (XML) and a related technology called Simple Object Access Protocol (SOAP), which is a specification for using XML in simple message based exchanges.

The UDDI Version 2.0 API Specification defines approximately 40 SOAP messages that are used to perform inquiry and publishing functions against any UDDI compliant service registry. This document outlines the details of each of the XML structures associated with these messages.

The purpose of UDDI compliant registries is to provide a service discovery platform on the World Wide Web. Service discovery is related to being able to advertise and locate information about different technical interfaces exposed by different parties. Services are interesting when you can discover them, determine their purpose, and then have software that is equipped for using a particular type of Web Service complete a connection and derive benefit from a service.

A UDDI compliant registry provides an information framework for describing services exposed by any entity or business. In order to promote cross platform service description that is suitable to a "black-box" Web environment, this description is rendered in cross-platform XML." [26].

UDDI has a very rich interface, which can be found in the data spec [26], even though it describes the same service as WS-Discovery, it does so using a far wider perspective, UDDI can hold vast amounts of information about the service owner, the service itself and other service related data. In order to maintain a somewhat neutral profile on the Web Service data that will be held in the repository, not all of these variables can be taken into account when designing the Web Service class, a more specific discussion about this subject is located in section 3.1.1.1.1 Service repository richness.

Table 11 is included to give a simple overview of the different variables and their datatype:

Field Name	Description	Data Type	Length
businessKey	This attribute is optional when the businessService data is contained within a fully expressed parent that already contains a businessKey value. If the businessService data is rendered into XML and has no containing parent that has within its data a businessKey, the value of the businessKey that is the parent of the businessService is required to be provided. This behavior supports the ability to browse through the parent-child relationships given any of the core elements as a starting point. The businessKey may differ from the	UUID	41

	publishing businessEntity's businessKey to allow service projections.		
serviceKey	This is the unique key for a given businessService. When saving a new businessService structure, pass an empty serviceKey value. This signifies that a UUID value is to be generated. To update an existing businessService structure, pass the UUID value that corresponds to the existing service. If this data is received via an inquiry operation, the serviceKey values may not be blank. When saving a new or updated service projection, pass the serviceKey of the referenced businessService structure.	UUID	41
name	Optional repeating element. These are the human readable names recorded for the businessService, adorned with a unique xml:lang value to signify the language that they are expressed in. Name search is provided via find_service call. Names may not be blank. When saving a new or updated service projection, pass the exact name of the referenced businessService, here.	string	255
description	Optional element. Zero or more language-qualified text descriptions of the logical service family.	string	255
bindingTemplates	This structure holds the technical service description information related to a given business service family.	structure	
categoryBag	Optional element. This is an optional list of name-value pairs that are used to tag a businessService with specific taxonomy information (e.g. industry, product or geographic codes). These can be used during search via find_service. See categoryBag under businessEntity for a full description.	structure	

Table 12, The list of variables and elements from the technical specification [26].

Table 11 contains the variables and a brief description of each one. These are the parameters that can be used to describe a Web Service. UDDI also contains several other parameters

and fields containing information about the publisher, similar services and other metadata about the service.

Mist

Mist in intended to be used in MANETS and other loosely coupled and unreliable networks, but it is also intended to be able to connect MANETS with WAN networks, this is possible due to topic routing. Topic based routing divides the messages and nodes into different topics, where a message is only sent to a node subscribing to a matching topic. This reduces network load, which again increases scalability. Mist supports multiple different communication modes, such as multicast, broadcast as well as a point-to-point unicast mode. These options allow the user to optimize the messaging setup to suit the needs of the user and the networking protocol, giving it a wider range of applicable settings.

Mist has proven to be fast and reliable [37] in MANETS and other dynamic networks, whilst still functioning in more static environments. The API of Mist is very easy to use, as it exposes only a few methods to the application utilizing it. This is a great advantage when it comes to implementation, as it has a very low threshold for implementation and starting up. This easy startup can also be explained by the multicast support, which requires very little setup, as most the recommended settings are already set and the only thing missing is to set up a multicasting node and a subscriber. With this easy setup advantage comes also the greatest disadvantage of Mist. Since Mist is intended for MANETS, it becomes cumbersome to set up a stable link across a WAN network, such as the Internet. The initial design and development will therefore be done using Mist, but the work to implement a second WAN mechanism will be facilitated and taken into account when implementing Mist. This method of implementation will also facilitate the modular design that is set as a premise in section 1.3.

ActiveMQ

Another alternative WAN mechanism is ActiveMQ [63]. ActiveMQ is an implementation of the AMQP [64] messaging protocol, which is an Oasis standard. ActiveMQ is designed to support reliable communication between two or more distributed processes. ActiveMQ is created in Java, but can be used in numerous different languages such as; Java, C, C++, C#, Python and many more. It is widely used in enterprise applications where reliability and performance are important aspects.

A very important part of any ActiveMQ networking system is the broker. Depending on the setup and configuration of the service, the broker has different roles and responsibilities in the network. ActiveMQ also supports message acknowledgements, that guarante the delivery of a message, the broker will hold and store the message until it is confirmed that it has been delivered. The broker can store the message in memory or a queue, such as KahaDB [65], the queue is safer and more reliable, at the cost of increased resource consumption for the broker. The standard setup is using a queue, and that will not be changed for this application.

The most basic and resource efficient setup of the network is a centralized broker, all nodes in the network connects to this broker upon startup, the broker works as an intermediary for all the nodes. When a node decides to send a new message, the message is first sent to the broker, which then forwards the message to all nodes subscribing to that topic.

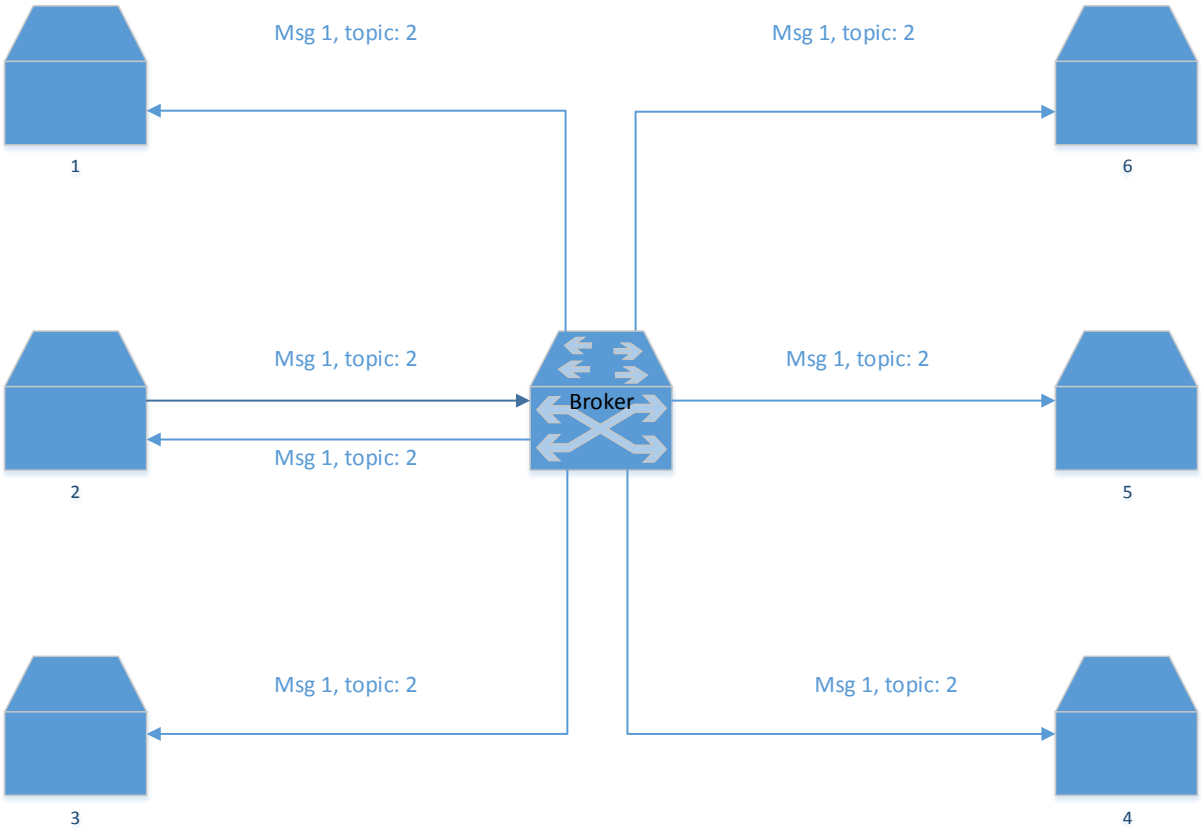

Figure 41 A Basic single broker to N nodes ActiveMQ implementation.

Figure 1 shows a basic diagram of how ActiveMQ works with a single broker setup. Node number 2 sends the message to the broker, the broker then forwards the message to all other nodes subscribing to that topic, included the sending node. In this application all nodes are subscribing to the same topic, as they are all interested in all messages. Due to the pub/sub message distribution, the message is also returned to sender. A simple feature eliminates loops as the sending nodes checks the message origin, and discards the message if it was sent by itself.

This setup has both advantages and drawbacks, one of the main advantages is the simplicity of setup and configuration. All nodes just need a broker address, port and a topic. This makes it easy to set up in large scale environments, as all nodes can use the same configuration options. The owner of the broker can also easily monitor the network traffic through the built in server monitoring feature of ActiveMQ.

There are also a few drawbacks to be considered with this approach, the first one is reliability and lack of redundancy. As all communications pass through a single point of failure, the application will be prone to errors. If the broker goes down or becomes unavailable, all communication will be broken and the application will be useless. Another challenge to this approach is the question of responsibility. The party responsible for maintaining and drifting the broker will have to invest a great amount of time and resources to run and maintain the broker, they are also the scapegoat if anything should fail, this is a responsibility that is not desired by anyone and a could be a cause for disagreement.

The second setup pattern of ActiveMQ is the broker to broker setup, where each node or cluster of nodes has a dedicated broker, and all communication between the nodes or node clusters is handled through their respective broker.

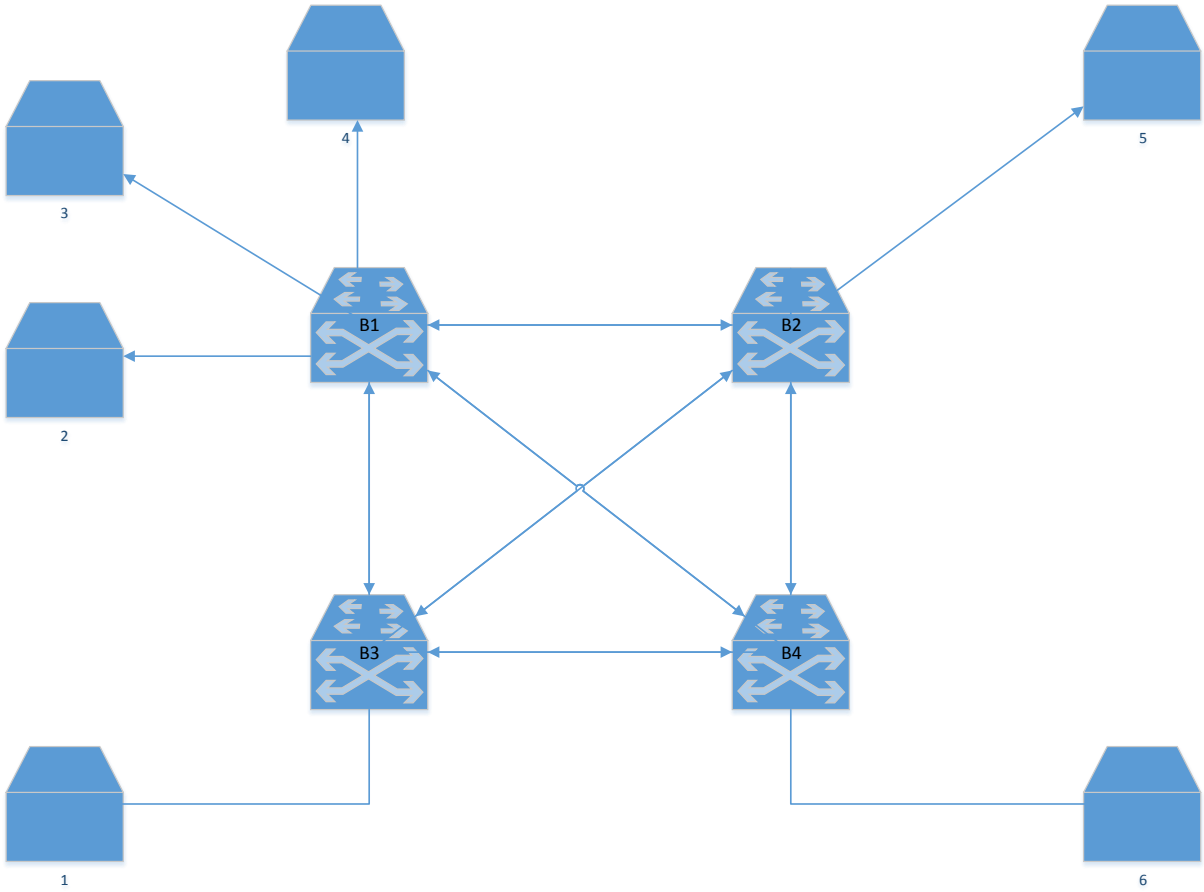

Figure 42 A ActiveMQ broker to broker network setup.

Figure 42 shows a basic ActiveMQ broker to broker communications setup. Each party hosts its own broker, and can then host as many nodes as they desire within their own network. The broker will then communicate with the other brokers and exchange information from and to the nodes, however, it does not have to communicate to all the other brokers, it can also communicate with a subset of the other brokers. This approach utilizes similar mechanics as the first approach, but thorough using multiple brokers, the system becomes more scalable, as the load is divided across them, and it adds redundancy to the system. If a broker is to go down, this will only affect the nodes directly connected to the broker, and they will not be accessible any more. This removes the single point of failure mentioned in the previous setup. Another advantage of this setup is that the responsibilities become clearer, as each party is responsible for their own node, and if their node goes down it will mainly affect themselves, however, services they are hosting will also become unavailable for other parties connected to the network.

A shortcoming with this setup is that it can be quite complicated to manage when there is a lot of brokers involved. This is due to that each broker must connect to all the other brokers in the network, which means that the configuration will not be the same for any brokers. It also suffers a performance degeneration when the number of brokers becomes high, this is because all messages will be sent to all brokers, and a message going from Node 1 to Node 6 will still pass through all brokers, including broker 1 and broker 2, which does not need the message. This creates a lot of

unnecessary network traffic and is therefore considered bad practice to set up the brokers as a full mesh [66] network. But a partial mesh adds a good redundancy with a smaller impact on performance.

Another reasonable setup for adding redundancy at a fair cost, is the master slave broker setup. The setup is similar to that in Figure 41, where there is one centralized broker handling all communications between the nodes, this node is referred to as the master broker, while there is 1 to N slave brokers, that will take the masters place if it goes down or becomes unavailable.

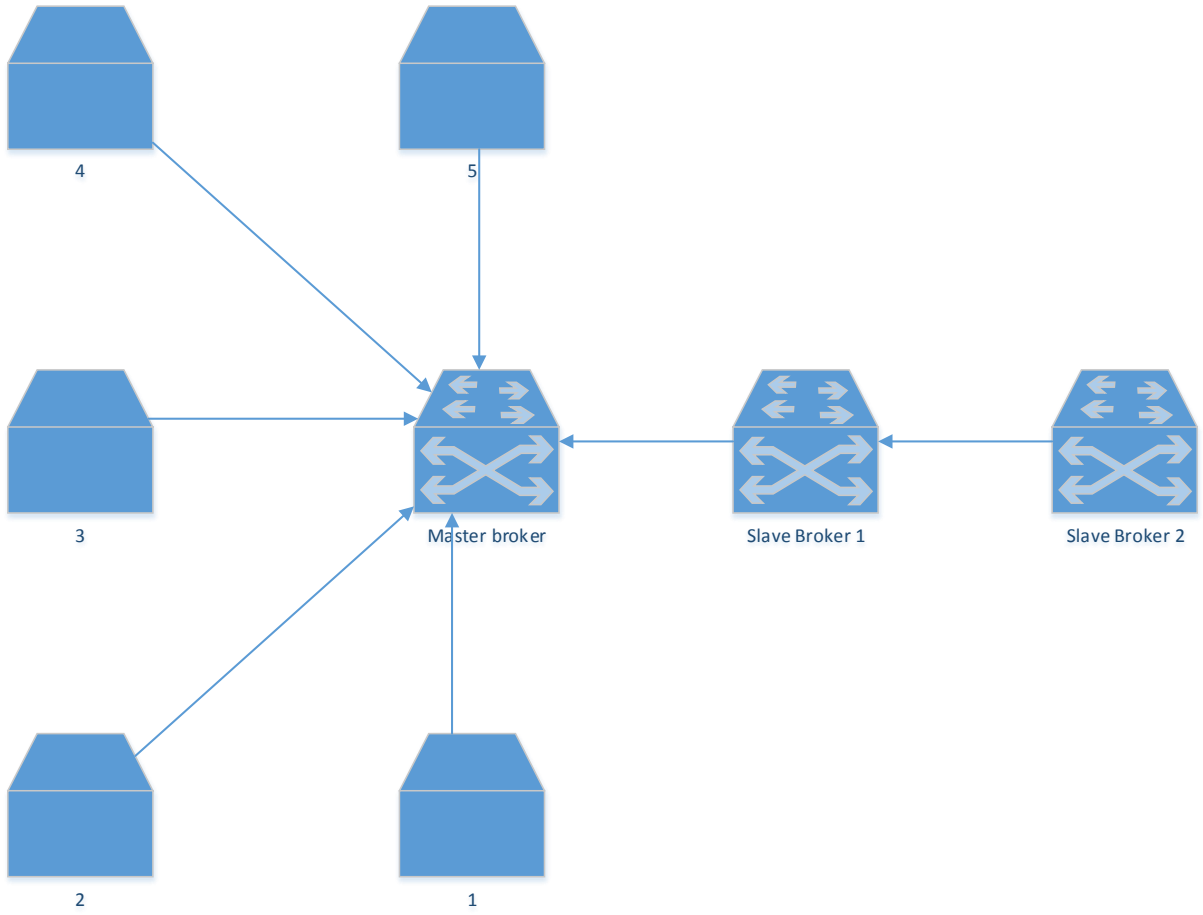

Figure 43 A ActiveMQ network with Master and Slave Brokers.

Figure 43 depicts a master-slave setup, where the master is working like a normal broker, and the slaves are set up to work as a backup for the master. This approach applies redundancy by having backup brokers available on demand. The shortcomings are very much the same as those mentioned in the first broker example, except for the redundancy issue.

The previuos examples, shown in Figure 41, Figure 42 and Figure 43 are all possible setup solutions when utilizing the ActiveMQ messaging client in the project. Due to this being a prototype, decreasing the need for redundancy and scalability as well as the limited amount of time available for testing different networking setups and their performance. The tests in for this application have been conducted using the first implementation, depicted in Figure 41.