PROJECT REPORT

IT2901 - Informatics Project II

Group 02 - FFI Publish/Subscribe

Written by:

Fredrik Christoffer Berg
Kristoffer Andreas Breiland Dalby
Håkon Ødegård Løvdal
Aleksander Skraastad
Fredrik Borgen Tørnvall
Trond Walleraunet

Spring 2015

NTNU

Norwegian University of Science and Technology

Page intentionally left blank.

We would like to extend our gratitude to Frank Trethan Johnsen and Trude Hafsøe Bloebaum from the Norwegian Defence Research Establishment for their excellent collaboration throughout the lifetime of this project. We would also like to thank our supervisor Alfredo Perez Fernandez, for his excellent feedback and help on the process and report parts of the project. At last we would like to thank Marianne Valstad for proofreading and giving feedback on the report.

Abstract

This report describes the work done during the course IT2901 - Informatics Project II. Our customer was the Norwegian Defence Research Establishment, which is a governmental organization responsible for research and development for the Norwegian Armed Forces. The assignment was to make an application that translates between different publish/subscribe protocols used in the Norwegian Armed Forces and the North Atlantic Treaty Organisation. The relevant protocols were the Web Services Notification protocol, Advanced Messaging Queuing Protocol, Message Queue Telemetry Transport and ZeroMQ. FFI and The Norwegian Armed Forces needed such a broker in order to participate in federated mission networking. This report mainly focuses on the development process. However, a thorough description of the product is also included. Sections regarding the implementation of the product assume a basic degree of technical knowledge from the reader.

Trondheim, 28 May 2015

Fredrik Christoffer Berg

Kristoffer Andreas Breiland Dalby

Håkon Ødegård Løvdal

Aleksander Skraastad

Fredrik Borgen Tørnvall

Trond Walleraunet

Table of Contents

List of Tables

List of Figures

Abbreviations

AJAX	=	Asynchronous JavaScript and XML
AMQP	=	Advanced Message Queing Protocol
API	=	Application Program Interface
DDS	=	Data Distribution Service
DVCS	=	Distributed Version Control System
FFI	=	Forsvarets Forskningsinstitutt
FMN	=	Federated Mission Networking
HTTP	=	HyperText Transfer Protocol
JRE	=	Java Runtime Environment
JVM	=	Java Virtual Machine
JSON	=	JavaScript Object Notation
MQTT	=	Message Queue Telemetry Transport
MVC	=	Model View Controller
NATO	=	North Atlantic Treaty Organisation
OASIS	=	Organization for the Advancement of Structured Information Standards
OKSE	=	Overordnet kommunikasjonssystem for etterretning
REST	=	Representational State Transfer
SASL	=	Simple Authentication and Security Layer
SDLC	=	Systems development life cycle
SOAP	=	Simple Object Access protocol
WSN	=	Web Service Notification
XML	=	eXstensible Markup Language

Chapter 1

Introduction

1.1 The course

The goal of the course IT2901 is to give students experience in working on a project with a customer. The students are to work in groups to develop a product related to information technology, with guidance from a supervisor. This project includes all parts of the software development process up to, but excluding, the maintenance/evolution phase.

1.2 The group

Our group consist of six members; Fredrik Christoffer Berg, Kristoffer Andreas Breiland Dalby, Håkon Ødegård Løvdal, Aleksander Skraastad, Fredrik Borgen Tørnvall and Trond Walleraunet. All of the members started a bachelors degree in computer science at NTNU in the fall of 2012. Over the course of the study, the members have gained experience in programming, as well as methodologies in software development.

Fredrik Christoffer Berg

Berg has prior experience in programming from courses taught at NTNU. Additionally, he has some experience with writing papers, both from school and in other areas. He is also interested in, and has a fairly good understanding, of different development methodologies.

Kristoffer Andreas Breiland Dalby

Dalby has prior experience with backend development and server administration. He also has experience with the programming languages Java, Python and JavaScript.

Håkon Ødegård Løvdal

Løvdal has experience with Java, Python and JavaScript from courses taught at NTNU. He also has knowledge of the Python framework Django, HTML and CSS through personal projects. Furthermore, Løvdal has gained experience with human-computer interaction from being a student assistant in the course TDT4180 at NTNU during the lifetime of this project.

Aleksander Skraastad

Skraastad has experience with Java, Python, HTML, CSS and JavaScript, as well as some experience with SQL databases. He had been involved in a few medium scale programming projects before, both commercially, and through volunteering and private projects.

Fredrik Borgen Tørnvall

Tørnvall has past experience from courses at NTNU. Programming experience includes Java, Python, C# and JavaScript. Additionally, he has past work experience working in small to medium sized teams. Finally, he has experience with Android development.

Trond Walleraunet

Walleraunet has experience with Java, Python, JavaScript, HTML and CSS through courses taught at NTNU. Additionally, he has work experience in server and network management. Finally, he has experience with project management working in small to medium sized teams.

1.3 The customer

The customer that we have been cooperating with on this project is the Norwegian Defence Research Establishment (hereby denoted FFI). FFI is a governmental organization responsible for research and development for the Norwegian Armed Forces. In addition, the organization is involved in the long term planning of the armed forces, as well as participating in other non-military research projects. The customer was located at Kjeller in Oslo. Thus, communication was mainly performed with the help of video calls.

1.4 Project Description

FFI was in need of an application aimed for translation between various publish/subscribe communication protocols. The application would include both protocols used internally in the Norwegian Armed Forces and the North Atlantic Treaty Organisation (hereby denoted NATO). The standard protocol for this type of communication is the Web Services Notification protocol (hereby denoted WSN) [1], which was the main focus of FFI.

At the time, FFI did not have such an application. They did however possess an implementation of the WSN protocol, developed during a project at NTNU in 2014. The group's goal was to develop software that could provide efficient communication between vehicles, stations, personnel, unmanned aerial vehicles (UAVs) and international NATO forces. Additionally, it would be a challenge develop, as the criteria given in the requirements could present the need for a deeper understanding of various publish/subscribe technologies. This project could yield an important application that would allow quick adaptation of new devices and tools that use other protocols than the NATO standard.

The task of this project was to implement a multi-protocol publish/subscribe brokering solution. Although there exists several different publish/subscribe frameworks and protocols, our goal was to make one solution which covered, and was able to translate, between several of them. The primary function however, was for the solution to support WSN, Advanced Message Queuing Protocol (hereby denoted AMQP) [2] and Message Queuing Telemetry Transport (hereby denoted MQTT) [3]. ZeroMQ [4] and Data Distribution Service (hereby denoted DDS) [5] were also interesting protocols, but with lower priority in this project. The customer wanted the application to translate to and from all the implemented protocols, regardless of their type. The application also had to include a graphical interface for administration. Note that clients for sending and receiving messages were not part of the project.

Finally, there were no security requirements defined. The group was assured that security would be implemented in the Internet layer (e.g IPSec) or link layer (e.g L2TP).

Chapter 2

Prestudies

2.1 Customer input

In our first meeting with the customer, the group received input about the overall structure of the desired product, as well as the research required. The customer put emphasis on a few important factors when designing the application. The protocol implementation had to be modularized in such a way that additional protocol support could be added with ease at a later time. A graphical administration interface was also required, to ensure ability to review broker data and adjust publisher/subscriber mappings. Additionally, extensive research had to be done, both to learn how the different protocols work and to get an overview of the publish/subscribe pattern in general. These factors had an impact on the group's choice of which sort of solution to develop. They also affected the development and research methodology.

The application was meant to be used internally by FFI researchers in training and experimentation exercises. This meant that the product was targeted towards a very narrow user base. Due to this, less resources were allocated to prototyping and testing of the user interface. Having a continuous dialogue with the customer and being able to adapt to the customers feedback was deemed sufficient to accommodate this area of development.

2.2 Publish/subscribe

The implementation of the product had to be structured in a publish/subscribe messaging pattern. In standard messaging protocols, users send messages directly to other users or groups of users. Publish/subscribe is based on categories, and each message sent is placed in a specific category. Whenever a message is put in a category and sent, every subscriber on the category will receive the message. Publishers and subscribers are separated, as subscribers do not know which publishers, if any, exist. In the same manner, publishers have no insight into which, if any, subscribers exists to the category. A user can be both a subscriber and publisher at the same time, depending on the category in question.

In the publish/subscribe pattern, the category is commonly known as a topic. The concept of topics is a close sibling to other messaging paradigms, like queues. The difference between queues and topics are described in the next section. The general publish/subscribe pattern is visualized in the figure [6] below:

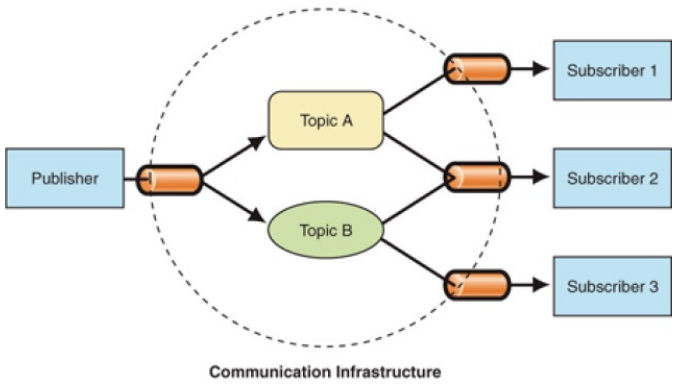


Figure 2.1: Publish/subscribe

2.2.1 Topics and Queues

While both the publish/subscribe pattern and WSN use the concept of topics to describe where a message is supposed to be routed, AMQP uses the concept of queues. As there is a fundamental difference between them, a detailed description of each is needed.

Topic

In a system implemented with topics, an incoming message will be distributed to every subscriber on the given topic. This means that if the topic ”example” has three subscribers, and a message is sent to this topic, the message will be sent to all three subscribers.

Queue

When a system is implemented with the behaviour of a queue, an incoming message will be distributed to *one* of the subscribers on the given queue. This means that if the queue has three subscribers, and a message is sent to this queue, the message will be sent to *one* of the subscribers. The way the subscriber is chosen can be implemented in a number of ways, for example at random or with the round-robin algorithm [7]. A typical usage for queue based messaging protocols is load balancing, which works out of the box when using queues. Additionally, with a queue based implementation, messages sent to a queue without subscribers will be stored until someone subscribes, and then offloaded on to that subscriber.

2.3 Protocols

The following sections describe the protocols that the system was intended to support. The protocols are all based on either the publish/subscribe or the queue pattern, but differ in their structure and implementation.

2.3.1 Web Services Notification

WSN is a messaging protocol developed by OASIS [8]. It is a protocol that implements the publish/subscribe pattern over HTTP [9]. The protocol uses SOAP [10] to send XML [11] messages over the network. The complete standard can be found at `oasis-open.org` [12].

2.3.2 Advanced Message Queing Protocol

AMQP is a binary protocol designed to send messages over a computer network. The protocol is built for handling the queuing of messages and routing (including publish/-subscribe handling) in a reliable and secure way. The main funding and development of AMQP comes from the financial sector. The customer requested that version 1.0 of the AMQP protocol was to be implemented.

AMQP version

AMQP has mainly two versions, with some substantial differences being used. The newest version, 1.0, has fewer public implementations and less documentation and examples. It is radically different from 0.9.1. As 1.0 was a requirement from the customer, little research was done on 0.9.1. It is important to note that the difference gap between the two, are big enough to almost regard them as different protocols. The developers of RabbitMQ (see section 2.4.2), one of the most popular AMQP brokers state: ”AMQP 1.0 is a completely different protocol than AMQP 0.9.1 and hence not a suitable replacement for the latter.”

2.3.3 Message Queue Telemetry Transport

MQTT is a publish/subscribe based protocol. It has the advantage of being lightweight and has minimal overhead. This protocol minimizes its use of bandwidth. MQTT has gained a lot of popularity over the last years, and its properties makes it suitable for everything from embedded systems to big implementations like Facebook Messenger [13].

2.4 Existing solutions

Following is an evaluation of currently existing solutions. These are products that partially cover the requirements, or could be used in the implementation in some way.

2.4.1 Apache Apollo

Apache Apollo [14] is an open source project maintained by the Apache Software Foundation [15] and commercially supported by Red Hat [16]. This application distinguishes itself from the existing solutions presented below, because it is a complete protocol agnostic broker that translates freely between AMQP, MQTT, OpenWire [17] and STOMP [18]. It is an up to date project, that is continually updated by the development team. Apache Apollo can be considered the next generation of the older Apache ActiveMQ broker [19]. It is a complete reimplementation of the old broker, built on more modern programming concepts and has a more efficient message handler. A more comprehensive description and analysis of Apollo can be found in appendix E.

Evaluation

Initially, the group set up an instance of Apache Apollo and read through both the user and developer documentation. Apollo seemed to be a very well written broker, and it covered many of the requirements given by the customer.

The internals of the message queuing and message translation systems are well designed. It uses a non-blocking reactor thread model, which keeps all threads running and consumes tasks asynchronously, allowing a fast message processing rate. Apollo dynamically converts large message queues from actual objects to pointer references when the queue becomes large. This helps mitigate the memory limits of the Java Virtual Machine (hereby denoted JVM), and allows Apollo to maintain queues larger than what a regular in-memory queue system would have. One of the issues with Apollo however, was that a lot of the functionality is written in Scala [20], which none of the group's members had any experience with.

2.4.2 RabbitMQ

RabbitMQ [21] is an open source project created and maintained by Pivotal Software [22], which provides commercial support for the product. The software is mainly written for version 0.9 of AMQP, but supports version 1.0 via an experimental plug-in. There is also plug-in support for MQTT. RabbitMQ provides a large amount of bindings for a wide range of different programming languages. The software and its plugins are written in the Erlang programming language [23].

Evaluation

RabbitMQ and its plugins are implemented in a programming language and paradigm that was not preferred by the customer. Thus, it would not be practical to build upon this solution. Additionally, no one in the group had previous experience with Erlang. Therefore, the group concluded that further research was unnecessary.

2.4.3 WS-Nu

WS-Nu [24] is a project developed as an assignment at NTNU. It is an implementation of the WSN protocol written in Java. It fully incorporates support for a large part of the WSN standard.

Evaluation

WS-Nu is well written, uses common patterns and has quite a decent level of documentation and code comments. From what can be concluded from the prestudy most, if not all, aspects of the WSN protocol standard has been implemented. Additionally, the WS-Nu implementation was recommended from FFI, as it had passed several of their tests. At this level of evaluation, the WS-Nu library was deemed a good fit as a protocol library to use in the project.

2.4.4 Apache Qpid Proton

Apache Qpid Proton [25] is a library that implements the AMQP protocol version 1.0. It is one of the few library implementations that use this version.

Evaluation

The Proton library is recommended by OASIS as it provides language library bindings to over ten popular languages in addition to the native Java and C++ implementation. Proton is widely used and is actively maintained. Due to this, it was chosen as the only viable alternative for implementation.

2.4.5 MicroWSN and NFFIPlayer

MicroWSN and NFFIplayer are both software developed and provided by FFI. MicroWSN is an implementation of a WSN broker. NFFIPlayer is a program for testing WSN by simulating a publisher. This package was created by FFI for testing in the field. However, the implementation is incomplete, and it lacks several features that a complete implementation should have. Since this software was property of FFI, it could not be further distributed by the group (see appendix F).

Evaluation

The package was not considered for further development, due to its incompleteness, and the fact that it is purely a broker for WSN. However, due to the graphical interface provided, it could be used for testing the WSN implementation.

2.5 Overall evaluation

The main outcome of the prestudies phase, was the discovery of Apollo. This proved to be a solution that it was possible to expand upon. One issue with Apollo however, proved to be the massive code base. It was clear that it would be difficult to grasp, given its sheer size and complexity. A lot of the code was hard to understand, and lacked thorough documentation and community usage. Additionally, the web interface and a few core system components were built using Scala. Regardless, Apollo provided what seemed a very dynamic and robust implementation of both the AMQP and MQTT protocols, which were requirements from the customer. In our opinion, expanding Apollo with WSN would be the best solution implementation wise, as it would probably open up time for adding other protocols the customer wanted.

The group chose to introduce a research phase, in order to make a final decision regarding Apollo. Due to this, a risk analysis targeted towards the research of Apollo was created. These risks would lay the foundation of the research process, and define the different areas of focus. The risk analysis is shown below, and a more detailed description of the methodology is explained in section 3.1.1.

Description	**Likelihood (1-9)**	**Impact (1-9)**	**Importance (Likelihood * Impact)**
Apollo being too complex	5	9	45
Too much time required to implement WSN	4	9	36
Apollo being incompatible with a protocol such as WSN	4	10	30
Difficulty in altering the Apollo admin interface	4	7	28
Unable to implement additional modules	3	9	27
Unable to learn Scala well enough	2	6	12

Table 2.1: Risk analysis for building the system based on Apollo

2.5.1 Alternative solution

If the outcome of the research phase would result in discarding Apollo as a foundation for the new system, a plan B was outlined. Plan B was to develop a completely new protocol brokering solution. WS-Nu would then be used as the WSN library of the product.

The two major development tasks were to create an administration interface in addition to the brokering solution itself. The initial architecture outline was strongly influenced by the architecture of Apollo, due to its robustness and the sheer quality of the product. A quick draft was constructed to get an idea of the components needed to build a system from scratch. Influences from Apollo are the components in figure 2.2 marked with green background color.

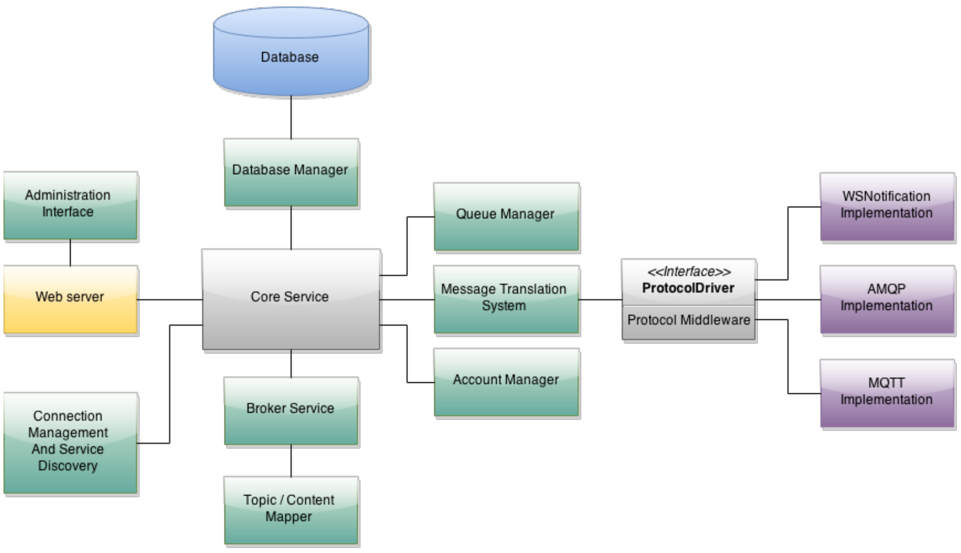


Figure 2.2: Early Pub-Sub broker architecture influenced by Apollo

2.6 Tools

The following section contains descriptions of all the tools used in this project. This includes tools used for communication, document sharing and development.

2.6.1 Skype

Skype [26] was the main tool used for communication with the customer. Meetings were held using the group call feature of Skype. Communication within the group was done using both group calls, as well as the instant messaging feature within Skype. This tool was convenient to use both for the customer and for the group. It provided all the communication features needed, both voice communication and a textual chat.

2.6.2 Mailing list

In addition to Skype, the group and the customer used email for communication inbetween the meetings. To make sure that every group member was up to speed on all relevant information, a mailing list was created. The mailing list was used to send a copy of every email received to the whole group. This was beneficial, as all of the group members received all information both from supervisor and customer.

2.6.3 Java

The customer preferred that the main part of development was done with Java [27]. Java is a programming language widely used in software development. The group chose version 8, which is the newest version, as the target. One of the main benefits of choosing Java and the JVM as language and platform is that the broker could run on every system where JVM was available. This means that the broker would not be affected by different hardware architectures or operating systems as long as they were supported.

2.6.4 Spring Framework

Spring Framework [28] is a model view controller (hereby denoted MVC) [29] framework developed in Java. It is a modern framework used mostly for development of web-based applications. This framework was chosen after a quick evaluation of different possibilities. It was chosen because it was easy to learn, and it provided exactly what was required, with a small amount of work to set up.

2.6.5 Bootstrap

Bootstrap [30] was chosen as a suitable framework for the visual aspects of the web application. Bootstrap provides a solid foundation that includes almost everything a typical website requires, while also being easy and flexible to customize. This foundation provided a large set of tools for page layout and elements, such as buttons, forms, navigation and, most importantly, a layout grid system. Designing it from scratch would have taken a lot of time, and this was less important than the broker solution and the functionality of the web application. Also, based on previous experience, Bootstrap has proven to be a stable framework that is easy to learn. Thus, the framework seemed like an appropriate tool for its purpose.

2.6.6 jQuery

jQuery [31] is a lightweight and fast JavaScript library. The main purpouse of jQuery is to promote simplicity in the use of JavaScript. By using jQuery, things like document object model manipulations, asynchronous JavaScript and XML (hereby denoted AJAX) and event handling is less comprehensive. Thus, the total amount of code needed decreases. It has also made a great impact in the enterprise market. Finally, it eliminates a lot of cross-browser incompatibilities. Different browsers use different JavaScript engines, and jQuery handles almost every one of these inconsistencies.

2.6.7 JavaDoc

The group chose to use JavaDoc, an automatic, comment based documentation tool to provide developer documentation of the code base. JavaDoc is supported in most Java Integrated Development Environments (hereby denoted IDE), and allows easy generation and update of developer documentation. JavaDoc is the standard documentation tool for development in Java. Few good alternatives that covers the same functionality exist, and the tool was an obvious choice in this project.

2.6.8 IntelliJ IDEA

IntelliJ IDEA [32] is a Java IDE. It contains a wide array of useful built-in functions for development in Java. It also includes support for build systems like Maven [33] and Gradle [34]. Version control systems are also well integrated in the tool.

Several different development environments exist. Everything from simple text editors, to fully integrated development environments might be used. Considering development in Java however, one might achieve some advantages by using a big platform. Both code completion, and functions for compiling code in the environment, make the development process easier and faster. The group considered mainly three alternatives when evaluating the environment: IntelliJ, Eclipse [35] and Netbeans [36]. The choice was based on mainly two deciding factors. Both Eclipse and Netbeans suffer from having a confusing amount of plugins for different tasks, where as the built in features of IntelliJ are widely cohesive. Secondly, IntelliJ was the first to release support for Java 8, and has proven to be stable and efficient over time. Thus, IntelliJ was chosen as the IDE.

2.6.9 Apache Maven

Apache Maven is a software solution for handling structuring, building and testing of Java projects. Maven also provides the developers with a package manager, which can be used to fetch libraries and add them automatically to the system. It is well implemented and widely used. Most Java Development Environments (hereby denoted JDE) like Eclipse and IntelliJ support Maven integration.

2.6.10 TestNG

TestNG [37] is a framework used for writing unit tests in Java. Compared to the more commonly used JUnit [38], TestNG is implemented with better handling of tests that require threads, easier parametrized tests and the ability to more efficiently use of data providers in the tests. TestNG also integrates out of the box with both Maven and Intellj IDEA.

2.6.11 Git

Version control is important when working on mid and large scale development projects. The group chose to use Git [39] in collaboration with GitHub [40], a cloud hosted, distributed revision control system that allows all team members to access the code. Git provides complete history and version control, with complete offline support, as the Git working directory acts as a complete repository.

There are other distributed version control systems (hereby denoted DVCS) available. However, the whole group had experience with both Git and GitHub beforehand, and the group agreed that using these services would be the best choice. It would have been counterproductive to research and learn a different version control system. As most of the DVCS accomplish the same tasks, the time should rather be spent on research and development.

2.6.12 Jenkins

Jenkins [41] is an open source continuous integration tool. It serves as a tool to automatize testing, maintenance and deployment. Whenever a change is integrated, Jenkins both runs tests and reports test results, as well as deploys the changes automatically.

Jenkins was used as an automatic test running tool, with full reporting from each build. It also automatically deployed the latest stable release of the system to the test server. Automatic test runs or deployment was triggered by integrating Jenkins with GitHub. Jenkins also provided out of the box integration with Maven and TestNG.

2.6.13 Gantt project

Gantt project [42] is a tool used to create Gantt charts. It is a simple chart creation tool exclusively made for creating clean and structured Gantt diagrams. The group had prior experience with the tool, and it provides simple and understandable diagrams.

2.6.14 Google Drive

Google Drive [43] is a platform for sharing documents. Drive supports all necessary types of files, as well as support for real time cooperation on documents. The simplicity of Drive is what makes it an attractive platform. The ability to simply upload a file to the website, and instantly having it shared, is both efficient and easy to grasp. Compared to using email or storage on a physical medium for file sharing, a cloud solution seemed more suitable. Drive was used to share all text and image files used in the report. It also served as the main platform for sharing and collaborating on time-tables and agendas for meetings.

2.6.15 LaTeX and ShareLatex

ShareLatex [44] is an online platform for collaborative editing of LaTeX documents. Furthermore, LaTeX is a typesetting system, designed for production of technical and scientific documentation.

LaTeX was chosen as typesetting system for the report. Compared to other solutions, like Microsoft Word, it generates cleaner and well formed documents. It also opens the opportunity for automatically generating a front page and table of contents. In the end however, the choice fell on LaTeX due to it being better, and more professional looking than many of its alternatives.

ShareLatex can to a certain degree be compared to Google Drive. The difference is that it is exclusively built for sharing LaTeX documents. It features the ability to cooperate on documents in real time, as well as compile and preview documents.

2.7 Risk analysis

Risk assessment was made early in the planning phase. The table 2.2 shows which risks were most likely to occur, as well as preventive actions. The analysis includes both events applying to individuals and the group as a whole. Upon inspection of each of the risks, a number between 1 and 9 was chosen for the likelihood of the case happening, as well as the potential impact it would have. The final rank of the risk was defined as these two numbers multiplied. When a risk was identified, a remedial action was included, in order to reduce the potential impact it would have.

Description	**Likelihood (1-9)**	**Impact (1-9)**	**Importance (Likelihood * Impact)**	**Preventive Action**	**Remedial Action**
Delivering an unsatisfying product.	3	9	27	Frequent customer meetings. Have a mutual understanding of the goal of the project.	Document what went wrong, and why it happened.
Too little work capacity within the development team	5	5	25	Frequent reviews of the progress. Proper distribution of workload.	Reviewing the time schedule. Add extra work hours each week.
Choosing a wrong/too complex architectural design.	4	6	24	Comprehensive research and planning. Set a deadline on which a final decision must be made on design approach. Constitute an alternative solution.	Fall back to alternate solution within the deadline.
Long term illness $>$ 3 days	3	6	18	Tell the group immediately when you notice illness etc.	In case of severe illness, negotiate with the customer about ambitions and requirements of the project.
Cooperation issues with the customer.	3	5	15	Frequent meetings with the customer. Uphold meeting agendas, and ensure mutual understanding at the end of meetings.	Emergency meeting with customer, assess and find a solution to the issue.

Customer does significant changes to the requirements.	2	7	14	Clearly define project scope and boundaries early on.	Negotiate with customer in order to find realistic goals.
Unable to meet time limit for deliverables.	2	7	14	Uphold time schedule. Review progress during meetings.	Working extra hours during evenings.
Too little domain knowledge within the development team.	7	2	14	Proper planning and research. Continuous research and customer cooperation throughout the process.	Meetings with the goal of cooperatively gathering domain knowledge.
Data integrity (working on different versions).	7	2	14	Use a version control system.	Roll back to a state without conflicts.
Short term leave of absence.	6	2	12	Tell the group immediately when you are aware of leave of absence.	Distribute workload among the other members of the group.
Cooperation issues within the group.	4	3	12	Define each members role in the group. Find solutions to problems early.	Talk about the problems as a group, attempt to find a solution all parties agree on.
Data availability/loss.	1	9	9	Cloud storage, local backup on several devices.	Cooperate to assess damage, redo lost work.
Other subjects occupying work time.	8	1	8	Create a time table. Separate project work from other work.	Extra work time during evenings.

Table 2.2: Risk analysis

Chapter 3

Process and methodology

3.1 Methodology

After the initial research on the project and deciding what sort of system to build, the group faced difficulties choosing which process model and development methodology to use. The main reason for this, was that as a part of the initial prestudy phase, the group also looked at existing solutions similar to what the customer had requested. During this, the group found that more research than initially assumed was required. It became apparent that an application that satisfied two of the requirements specified by the customer existed, as well as some other supported protocols. The option of expanding this product arose, and the group realized that this would have a large impact on the project. Both the quality of the code, and the degree of requirements achieved, would be affected in a positive way.

The project was eventually split into three different phases. The first phase would take shape as a research project, using models and techniques relevant to performing a research project. The second phase would be dependent on the outcome of the research phase. During exploration of the two potential outcomes of the research phase, it became apparent that an iterative, incremental development process was suitable either way. Detailed description of the methodologies, models and techniques used are explained in their respective sections below. The final phase would incorporate tasks needed to finalize the project report, documentation and other formalities related to the project.

3.1.1 Research

The research methodology chosen was a modified version of the phase-gate model [45]. The model was chosen due to the need of a structured way of performing product research, as none of the group members had much experience with such work. After considering some different ways of structuring it, a modified phase-gate model was chosen as the most suitable. This is a model made for testing and prototyping of new products or potential business ideas.

The model describes a time period as a phase. Furthermore, each phase has a gate, which is a goal or evaluation criteria that has to be met. In this case, each phase was a part of the research required, in order to choose whether or not to further develop Apollo. The traditional model commonly has five such phases. The group's modified version used the discovery, scoping, business plan (analysis part), and testing phases. Other phases and/or aspects of the model were not executed. Individually, each group member used these phases as a research cycle, where new aspects or possible problems were discovered. After evaluating whether or not it would be technically and time-wise achievable, further and deeper research proceeded. If an aspect or problem that would not pass the gate evaluation criteria appeared, further research would be aborted and the alternative development plan executed.

In order to cope with the size and complexity of the application, each group member was assigned a different part of the system to evaluate. Using the modified Phase-Gate model, each part of the system was researched and evaluated using the gate criteria stated in figure 3.1. Additional discoveries during research prompted further analysis following the same cycle. As long as no part of the system failed the criteria, given the current understanding, research could progress.

Additionally, a final deadline was set. Due to time constraints, no more time could be allocated to research before development had to start. If none of the gate criteria failed, research would go on until the final deadline before a decision was made. The general gate criteria were taken from the research phase risk analysis (see table 2.1), and discussed internally. The group formed what the Phase-Gate model describes as a steering-committee, that would make the final decision on whether or not to proceed. The committee consisted of the whole group, with the lead developer functioning as a leader of the committee. He was responsible for evaluating the arguments in a factual manner.

The final decision was to develop a new brokering system. The system would be called ”Overordnet kommunikasjonsystem for etterretning” (hereby denoted OKSE). The plan was to use WS-Nu as the main implementation for WSN, and develop a broker based on many of the concepts discovered in Apollo. The name was, however, merely a placeholder and subject to change. The technical aspects leading to the final decision are more thoroughly discussed in chapter 7.1.2.

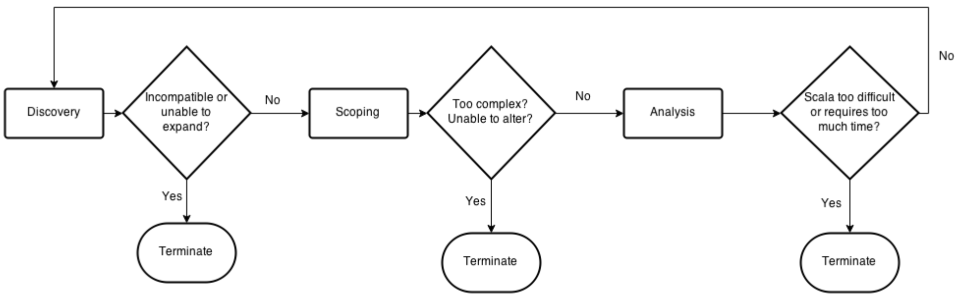

Discovery
Incompatible or unable to expand?
No
Yes
Terminate
Scoping
Too complex? Unable to alter?
No
Yes
Terminate
Analysis
Scala too difficult or requires too much time?
No
Yes
Terminate

Figure 3.1: Modified Phase-Gate model of the research process

3.1.2 Development

The second phase would encompass the actual development of the product. Neither the group nor the customer had a firm grasp of what this kind of product would look like, and how it should function. The customer was eager to provide feedback, both in terms of their own needs and requirements, as well as on aspects such as the user interface. Utilizing this dynamic, an iterative and incremental approach seemed reasonable. Both to secure that the product was as close to customer expectations as possible, but also to allow different techniques and approaches to be implemented for demonstration and feedback. Incremental development would allow the group to continuously expand the product and get instant feedback from the customer. An always up-to-date version of the product hosted live on the Internet was planned to achieve this dynamic.

Scrum [46] was chosen as the agile development method of choice. In contrast to the traditional waterfall model, Scrum is a methodology based on incrementally developing a product. It incorporates sprints, which is a set time interval, often lasting one or two weeks. With each sprint, a new increment of the product is delivered. Additionally, the group used Scrum meetings, where each member gave a quick summary of the work done and problems encountered. At the end of each sprint, the group had a longer meeting in order to evaluate to which degree the planned tasks were completed. A detailed plan of the next sprint was also formed during the meeting.

All the group members had prior experience with this model from previous school projects. Additionally, Scrum was suited for this project because it encouraged frequent updates from everyone. This was important for keeping track of progress, as well as planning customer and supervisor meetings. More generally, using Scrum allowed the group to react to changes in the requirements, either due to problems arising, time constraints or alterations made by the customer.

Elements from test-driven development (hereby denoted TDD) were planned to be used for suitable parts of the development process, mainly models and methods with a high degree of logical operations. When a task had clear testing conditions, the plan was to write tests before the actual program code. Tasks would be considered completed once they had passed all the tests. Additionally, the process of writing tests provided a clear definition of what individual components or units were supposed to do.

Another possibility would have been to use the waterfall model or a variation of it. This model is really simple to use and to manage, something which would have been the main advantage in this case. However, given that the group had some issues both defining, and understanding what to actually develop, it would have been difficult to define beforehand. The technical knowledge of how a broker solution had to be built, was not present at an early stage. Secondly, the model does not adapt to change in a good manner, due to the development plan being clearly defined at an early stage.

Supporting development with tools

Many of the tools listed in section 2.6 were chosen explicitly because they did fit with the development methodology. To provide better efficiency with the Scrum process and the continuous release of new versions of the software, Jenkins (see section 2.6.12) was integrated in the development cycle. Jenkins was used to automatically build and test the code every time a ”pull-request” was created on the GitHub repository. A pull-request can be explained as an action that is done by a developer to signal that their branch of code is ready for integration with the main branch.

Integration with GitHub and Jenkins provides fast feedback displaying any exceptions or test failures. In addition, the group planned to use a strict scheme on how branches should be merged together. The pattern was based on the idea of having a branch called ”develop”, where the latest development code was added. In addition to the develop branch, a branch with the latest code that was considered stable was created, this branch was named ”master”. If a developer wanted to get the code into the master branch, it had to go through develop. The develop branch was only merged into the master branch when it was approved by more than two members of the development team. Another important guideline was that a pull-request could not be merged by the creator, even if the code was building successfully and all the tests succeeded.

In addition to the pull-request building from Jenkins, another job was created to build and deploy the master branch every time it was updated. When a new stable release was merged into the master branch, the code was built and deployed on the test server, publicly available to the group and the customer. This was done frequently, depending on the situation and the need for customer feedback or testing. As a minimum, deployment of the product to the test server was done before the end of each sprint.

3.1.3 Report

The report was one of the aspects of the project that required a lot of resources. Thus, a structured way of writing the report was important. The group used different methods to make sure that the report was up to date. All decisions or goals met were noted in the report. The notes in the report were then elaborated upon by all the group members, to create meaningful sections. Additionally, all new major changes to the report were reviewed by all the members, before they were finally agreed upon.

Design was also a factor when creating the report. The decision to use Latex was partially made due to this. Additionally, it was important to make figures and tables as understandable as possible. The general strategy used to achieve this was to have group members, as well as the supervisor giving feedback.

Before the delivery deadline of each version of the report, the group set aside time for review and discussion. This phase also included consideration of the feedback received from the supervisor and customer. After all input and feedback was considered, a final version was cooperatively written and agreed upon.

To ensure the quality of the language and readability, a third party with good English language skills was involved to read and comment on the report towards the end of the project.

3.2 Project Organization

After discussing with the customer, the team decided to have development iterations lasting two weeks, starting Wednesday February 25. These two weeks gave the group room to plan according to mandatory work in other courses. The customer also recommended and preferred having weekly meetings. The research phase was the main reason for the late development start.

Furthermore, a distribution of roles was made. This was mainly based on the previous experience of each member. However, it was also important to cover the most common roles in software development teams [47], and to cover most of the phases in the system's development life cycle (hereby denoted SDLC) [48]. Since the group only consisted of six people, it was important to declare these roles to prevent any aspects of the development life cycle to be neglected.

3.2.1 Product owner

The product owners were Frank Trethan Johnsen and Trude Hafsøe Bloebaum from FFI. Their main task was to oversee the development process, and provide continuous feedback on the product. Other than providing requirements and defining the scope of the project, the customer did not participate in the development.

3.2.2 Scrum master

Walleraunet was given the role of scrum master. This was due to him having prior acquaintance with the customer, as well as prior experience in this role. A scrum master has the main responsibility for monitoring progress and deliverables, as well as being the main communication link between the group and the customer.

3.2.3 Scrum team

The other five members of the group constituted the development team. The team and the scrum master were responsible for developing the product, as well as completing the other deliverables in time. In order to distribute some responsibility away from the scrum master, the group assigned members to different areas of the project. The group created roles for the remaining five members of the group. The goal was for these roles to cover all the aspects of the project. If any issues arose, one would always have someone to consult. Additionally, roles were distributed to complement the skills of each member (see section 1.2).

Lead developer

The task of the lead developer was keeping control of progress on the development part of the project. This role was given to Skraastad due to his experience from other development projects. Keeping track of inter-sprint progress, version control, development planning and integration were key responsibilities of the lead developer. Additionally, he would be the link between the solutions architecture and the rest of the development team, providing more detailed descriptions and structure of the different system components.

Report and documentation

Berg was responsible for distributing work regarding the report and documentation writing. He was the most experienced when it came to writing scientific reports, and had used some time researching the structure of other reports previously written in this course. The main task of Berg was documenting work to be included in the report, and ensuring progress. He was also responsible for making sure that the written documentation was satisfiable.

Architecture and solutions analysis

Løvdal was given the responsibility of overseeing the architectural design of the brokering system. He was responsible for ensuring that all the aspects from the functional analysis was mapped into respective implementable technical solutions. Finally, he was assigned the task of overseeing that the link between the broker and administration interface were properly constructed.

Testing and configuration management

Due to the nature of the project description, the group's choice of architecture, and the complexity of the final system, testing and configuration management was important. To ensure that none of the relevant aspects of unit, integration and system testing were neglected, Dalby was given these responsibilities. His main tasks were to ensure proper configuration of the system, as well as ensuring that testing of different configurations and settings were thoroughly executed.

Design, user experience and functional analysis

One of the main features of the system was the administration interface, with the ability to perform topic mapping and administer configurations. To ensure that this part of the system was informative and intuitive, Tørnvall was given the responsibility to oversee this part of the system, with emphasis on interface design and user experience. Additionally, he was responsible for expanding on the overall requirements given by the customer, refining them and exploring functional dependencies. This was done to ensure that the requirements were as precise, clear and complete as possible, reducing the risk of neglecting potential aspects of the system functionality. The role was given to him based on his wish to work with the design part of the system.

3.3 Resource management

This section describes plans for distribution of time and manpower. They are meant to provide a simple visualization of the work planned, and function as tools to track progress. The resource distribution work started in the introduction phase of the project, and the initial estimation and schedule were completed after deciding on the project methodology. The progress was tracked during the project by noting hours used on different tasks. Note that the figures contain the initial plans, and do not correctly reflect the end result.

3.3.1 Work breakdown

The work breakdown structure (hereby denoted WBS) in figure 3.2, provides an overview of the different work packages that the project is divided into. It is designed as a hierarchy, where each package may have several sub-packages. It also provides a plan for time usage on each major package. The WBS was used as a tool to create the Gantt diagram, and was used throughout the project to define tasks for each sprint.

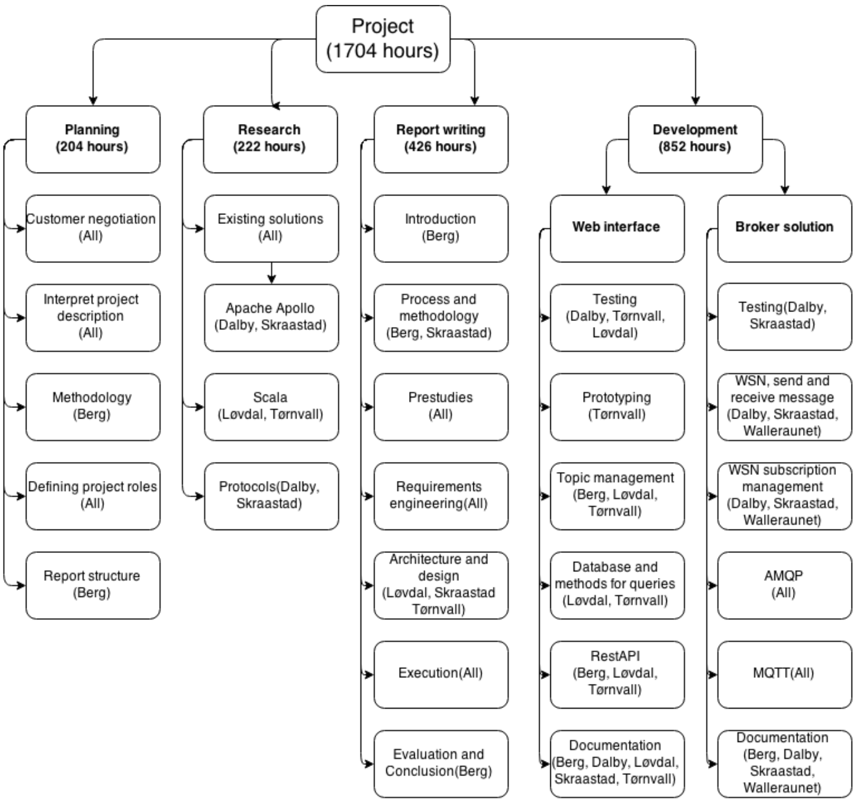

Project (1704 hours)
Planning (204 hours)
Customer negotiation (All)
Interpret project description (All)
Methodology (Berg)
Defining project roles (All)
Report structure (Berg)
Research (222 hours)
Existing solutions (All)
Apache Apollo (Dalby, Skraastad)
Scala (Løvdal, Tørnvall)
Protocols(Dalby, Skraastad)
Report writing (426 hours)
Introduction (Berg)
Process and methodology (Berg, Skraastad)
Prestudies (All)
Requirements engineering(All)
Architecture and design (Løvdal, Skraastad Tørnvall)
Execution(All)
Evaluation and Conclusion(Berg)
Development (852 hours)
Web interface
Testing (Dalby, Tørnvall, Løvdal)
Prototyping (Tørnvall)
Topic management (Berg, Løvdal, Tørnvall)
Database and methods for queries (Løvdal, Tørnvall)
RestAPI (Berg, Løvdal, Tørnvall)
Documentation (Berg, Dalby, Løvdal, Skraastad, Tørnvall)
Broker solution
Testing(Dalby, Skraastad)
WSN, send and receive message (Dalby, Skraastad, Walleraunet)
WSN subscription management (Dalby, Skraastad, Walleraunet)
AMQP (All)
MQTT(All)
Documentation (Berg, Dalby, Skraastad, Walleraunet)

Figure 3.2: Work breakdown structure

3.3.2 Gantt diagram

The Gantt diagram in figure 3.3 provides a simple overview of the planned time-distribution of each task in the project. The chart consists of the work packages from the WBS, which have been distributed over the different sprints. This was done in order to make a schedule. The Gantt diagram is concerned with major work packages and milestones, and may exclude lesser tasks.

Task	**Start**	**End**	**Duration**	**Hours**
Planning and research phase	21/01/2015	24/02/2015	14 days	336 hours
Iteration 1	25/02/2015	06/03/2015	8 days	192 hours
Iteration 2	09/03/2015	20/03/2015	10 days	240 hours
Iteration 3	23/03/2015	01/04/2015	8 days	96 hours [a]
Iteration 4	07/04/2015	17/04/2015	9 days	216 hours [b]
Iteration 5	20/04/2015	30/04/2015	9 days	216 hours [c]
Iteration 6	04/05/2015	15/05/2015	9 days	216 hours [d]
Final phase	22/05/2015	30/05/2015	8 days	192 hours
Total			**75 days**	**1704 hours**

[a]Easter holiday and adjusted for the leave of absence of three group members (USA)
[b]Monday of Orthodox Easter
[c]May 1
[d]Ascension Day

Table 3.1: Available time

3.3.3 Available time

Table 3.1 shows the time that was available. This was meant as a supplement to the Gantt diagram, providing a simple overview of the planned time usage. It accounts for Easter, as well as other holidays. Figure 3.4 shows the distribution among the different packages from the WBS.

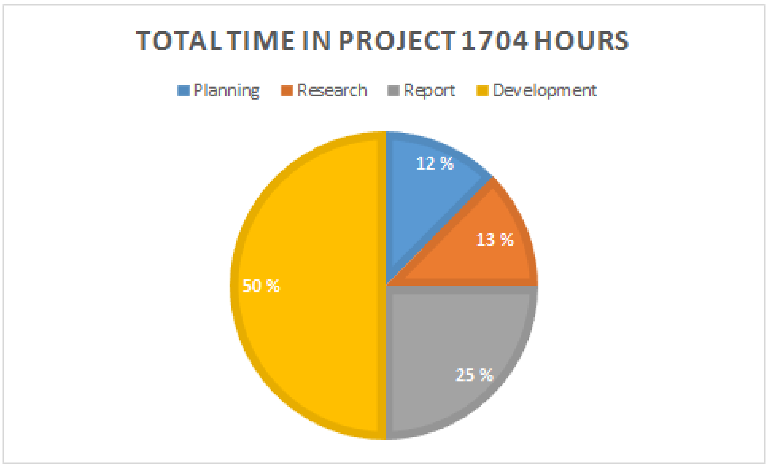
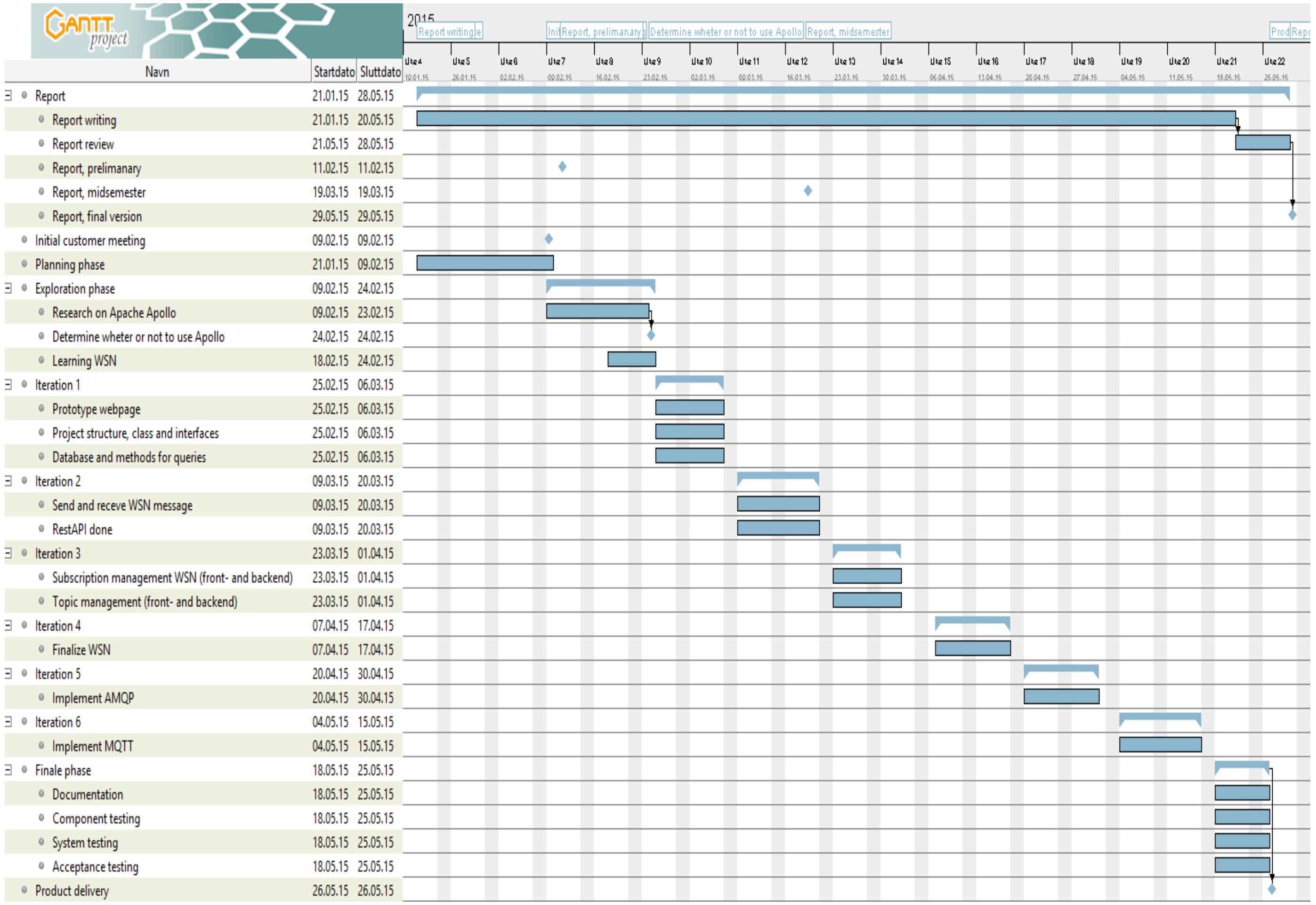


Figure 3.4: Time distribution of main WBS packages



Navn	Startdato	Sluttdato
Report	21.01.15	28.05.15
Report writing	21.01.15	20.05.15
Report review	21.05.15	28.05.15
Report, prelimanary	11.02.15	11.02.15
Report, midsemester	19.03.15	19.03.15
Report, final version	29.05.15	29.05.15
Initial customer meeting	09.02.15	09.02.15
Planning phase	21.01.15	09.02.15
Exploration phase	09.02.15	24.02.15
Research on Apache Apollo	09.02.15	23.02.15
Determine wheter or not to use Apollo	24.02.15	24.02.15
Learning WSN	18.02.15	24.02.15
Iteration 1	25.02.15	06.03.15
Prototype webpage	25.02.15	06.03.15
Project structure, class and interfaces	25.02.15	06.03.15
Database and methods for queries	25.02.15	06.03.15
Iteration 2	09.03.15	20.03.15
Send and receve WSN message	09.03.15	20.03.15
RestAPI done	09.03.15	20.03.15
Iteration 3	23.03.15	01.04.15
Subscription management WSN (front- and backend)	23.03.15	01.04.15
Topic management (front- and backend)	23.03.15	01.04.15
Iteration 4	07.04.15	17.04.15
Finalize WSN	07.04.15	17.04.15
Iteration 5	20.04.15	30.04.15
Implement AMQP	20.04.15	30.04.15
Iteration 6	04.05.15	15.05.15
Implement MQTT	04.05.15	15.05.15
Finale phase	18.05.15	25.05.15
Documentation	18.05.15	25.05.15
Component testing	18.05.15	25.05.15
System testing	18.05.15	25.05.15
Acceptance testing	18.05.15	25.05.15
Product delivery	26.05.15	26.05.15

Figure 3.3: Gantt diagram

Chapter 4

Requirements engineering

4.1 Requirements description

The requirements engineering process was continuously negotiated between the customer and the group. Although the customer had a general idea of what the system was supposed to do, this process went on throughout the duration of the project. This was done to make sure that the requirements were realistic, given the time available. The requirements lists reflect the final result of the process.

Requirements are separated into two categories. A functional requirement describes a function of the system or one of its components. A non-functional requirement can be viewed with regards to the operation of the system. Rather than listing specific functions of the system, it describes the performance or operation of the system as a whole. In this project, strict requirement and narrow requirement scope was not provided by the customer. The requirements given were general, and listed some features that the system would have to provide, but not an extensive list. This led to the group having to create a more comprehensible set of requirements as a basis for development.

For the remainder of this chapter, a client is defined as any application that implements the protocol in question as its means of communication. Implementing a general purpose client was not a part of the product description. To the group's knowledge, only one general purpose client exists for the protocol versions implemented. And it is a commercial client supporting AMQP 1.0. As such, libraries and scripts were used internally as client representations. These are covered in appendix C.8.

4.2 Requirements elicitation

The functional requirements were elicited as a result of discussion both within the group and with the customer. Additional requirements were taken into consideration from the results of the research phase, as well as from the standards of WSN, AMQP and MQTT. Use cases and user stories were intended as support tools for the group to define the exact requirements and formulations of them. This was due to the lack of strictly defined requirements from the customer. The use cases describe the steps that are needed to be performed in order to accomplish a certain task. The result of the process was a list, describing the requirements and the prioritization of them.

The non-functional requirements elicitation was purely a result of discussion with the customer. After identifying the properties of the system, they were classified and rated in order of importance.

Two issues arose when requirements for the different protocols were to be defined. First of all, the customer wanted as many protocols as possible to be implemented. After considering the time available, the three most important protocols were listed as requirements. The decision was subject to change however, depending on how much time it would take to implement each protocol.

The other issue was defining the protocols as requirements. On one hand, they could be seen as functional requirements. ”As a user, one must be able to send a message using protocol A”. This would be regarded as a functional requirement. On the other hand, protocol support could be viewed as a property of the system. The protocol would then be considered a set of standards that the system has to conform to, and fall under the compatibility category of non-functional requirements. The functional requirements would then only be described as ”a user being able to send a message”, regardless of protocol. The decision would not make any profound difference to the product. Thus, extensive research into this definition was not pursued.

4.3 Functional requirements

After a brief evaluation, the aspects around different protocols were defined as functional requirements, noted as FR in the requirements table. Mainly due to a consideration of how they would be tested. It seemed more reasonable to include them as functional requirements when considering the integration testing of components, as well as system and acceptance testing. The other functional requirements were concerned with what a user was supposed to do with the administration interface. The main goal of the interface was to expose configuration of mappings between supported protocols, publisher and subscriber database listings, as well as topics and content filters. The functional requirements are described in table 4.1, and were a result of the requirements elicitation process.

Functional requirements			
#	**Use Case**	**Description**	**Priority**
FR1	U5, U6	A user must be able to send and receive messages over the WSN protocol, with a client of his/her choice.	1
FR2	U7	A user must be able to subscribe to a topic using the WSN protocol, with a client of his/her choice.	1
FR3	U8	A user must be able to register as a publisher on a topic using the WSN protocol, with a client of his/her choice.	1
FR4	U9	A user must be able to unsubscribe from a topic using the WSN protocol, with a client of his/her choice.	1

FR5	U10	A user must be able to unregister as a publisher on a topic using the WSN protocol, with a client of his/her choice	1
FR6	U11	A user must be able to retrieve the latest message on a topic using the GetCurrentMessage function, using the WSN protocol, with a client of his/her choice.	1
FR7	U12	A user must be able to renew his/her subscription using the WSN protocol, with a client of his/her choice.	1
FR8	U13	A user must be able to pause his/her subscription using the WSN protocol, with a client of his/her choice.	1
FR9	U14	A user must be able to resume his/her subscription using the WSN protocol, with a client of his/her choice	1
FR10	U15	A user must be able to send multiple notification messages in a single Notify using the WSN protocol, with a client of his/her choice.	1
FR11	U5 & U6	A user must be able to send and receive messages using the AMQP protocol, with a client of his/her choice.	1
FR12	U16	A user must be able to subscribe to a topic using the AMQP protocol, with a client of his/her choice.	1
FR13	U17	A user must be able to unsubscribe using the AMQP protocol, with a client of his/her choice.	1
FR14	U1	An administrator must be able to map topics/dialects with other topics.	2
FR15	U2	An administrator must be able to edit the different subscriptions. That means delete one or all subscriptions on a given topic. This also includes deleting all topics.	3
FR16	U 3	An administrator must be able to get information in the administrator interface about the server.	4
FR17	U4	An administrator must be able to log in with a user name and password to get access to core functions.	5

Table 4.1: Functional requirements

4.4 Non-functional requirements

Following is a list of the non-functional requirements (hereby denoted NFR) created. The requirements are listed in descending order of importance.

4.4.1 NFR1 - Extendability

NFR1.1 - Modularization

The different components of the system should be decomposed in such a way that the respective components are loosely coupled, to allow easier addition of more protocols at a later point of time. It also allows upgrading and further development of different parts of the system individually.

NFR1.2 - Protocol Independence

The system must be able to translate between protocols independent of their type. More precisely, it must not only support to/from WSN, but also between other supported protocols, without having to go through WSN.

4.4.2 NFR2 - Documentation

All parts of the application must be documented according to the language specific standards, preferably in English.

4.4.3 NFR3 - Open Source

The software shall be open source. The software shall be available for use, change and distribution by anyone. It will be licenced under the MIT license [49]. The code will remain available on Github after completion.

4.5 User stories

The following table describes the user stories defined. The stories were used as tools to create use cases. They were defined based on conversations with the customer, and answers to what the customer wanted to do with the system. The different aspects were then structured in table 4.2.

User stories		
As a(n)	**I want to**	**So that (I can)**
Admin	Log in with username and password.	Gain access to the system.
Admin	Get information about server.	See number of subscribers and system status.
Admin	Get information about subscribers.	See number of connections to the broker.
Admin	Get server information.	See which IP the broker is running on.
Admin	Get IP and topics information.	See information about subscribers and topics.
Admin	Get resource overview.	See load and balance of the server.
Admin	Edit topics/dialects.	Map topics/dialects.
Admin	Edit subscriptions.	Unsubscribe users on topics.
Admin	Edit topics.	Delete a topic.
Client	Subscribe with AMQP or WSN	receive messages.
Client	Register as publisher with AMQP or WSN	send messages.
Client	Unsubscribe	No longer receive messages.
Client	Unregister	No longer send messages.

Table 4.2: User stories

4.6 Use cases

The use cases were mainly used for internal communication within the group, as a simple way to share information about the high-level workings of the system. They were also used as a tool to help create and visualize the requirements. Additionally, the group used the use cases to evolve and expand the administration interface by adding information that was useful during the creation of the use cases and the development phase. One use-case diagram is covered here as an example, the rest can be found in appendix A.1.

4.6.1 Use case 1 - Topic mapping

After (and/or before) a publisher has added a topic, the admin must be able to map this topic to another topic. The topic mapping interface must also provide a way of creating a relay from messages received on one topic, to a different topic.

Use case ID	U1
Use case Name	Topic mapping
Description	Admin should be able to map between topics.
Pre conditions	The admin is logged in.
Standard flow	1. Click the config pane. 2. Enter the topic you want to map from. 3. Enter the topic you want to map to. 4. Click the ”Add”-button.
Alternative flow	**2A:** Server will not answer request 1. Display error message. 2. Open config file for topic mapping. 3. Edit file with the mapping. 4. Save config file. 5. Restart/reload server.
Post conditions	Subscribers of different topics now receive a notify when a publisher publishes on one of the topics.

Table 4.3: Use case 1 - Topic mapping

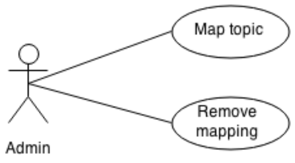


Figure 4.1: U1 - Topic mapping

Chapter 5

Architecture and implementation

5.1 Broker architecture

This section discusses the high-level aspects of the broker architecture. Some parts of the system are explained in greater detail to provide clear purpose or intent. Additionally, a developer manual can be found in appendix D.

5.1.1 Goals and strategy

Building on the results and findings in the research phase, a set of goals were created for the design of the system architecture. Several aspects of the Apollo broker were really well thought trough, and were on the priority list from the start. These included, but were not limited to: Complete protocol agnostic interface. A priority message queue supported by a multi-threaded execution service. Loose coupling of modules, as well as high extendability and low maintenance effort was important and in accordance with the non-functional requirements.

Expanding on these traits, the group wanted a system architecture where each component was a standalone service. A service would expose a public interface from which other components could request data access or services from. Another goal of the design was to have a core service responsible for registering other services, starting them, stopping them, and registering protocols. This would allow the system to be extended by simply registering the new component, without further modifications.

The customer had requested a graphical user interface, and building on experience and knowledge in the group, a web based interface was deemed most appropriate. The group wanted to have an administration interface built using the dynamic properties of the outlined system. Registering a new protocol should not require the administration interface component to be modified. Registering a new core service that a potential developer wanted to expose in the administration interface, should only require adding a new model, Application Programming Interface (herbeby denoted API) controller and template fragment.

Based on these thoughts and ideas, a list of goals were created:

- The system should not need to know about the details of registered protocols in order to perform as intended. In essence, it should be completely protocol agnostic.
- Extending the system with new functionality should be as easy as possible.
- Extending the system with new protocols should be as easy as possible.
- Extending the system should require as little alteration to existing code as possible.
- Core functionality should be adhering to the principle of ”Separation of concerns” [50].
- Core services should be standalone entities that other components can utilize.
- The system should have a core service responsible for the entire life cycle of other registered components.

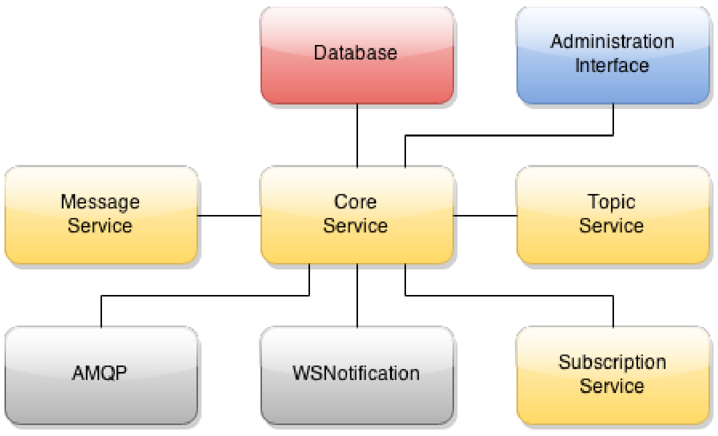


Figure 5.1: High-level system architecture

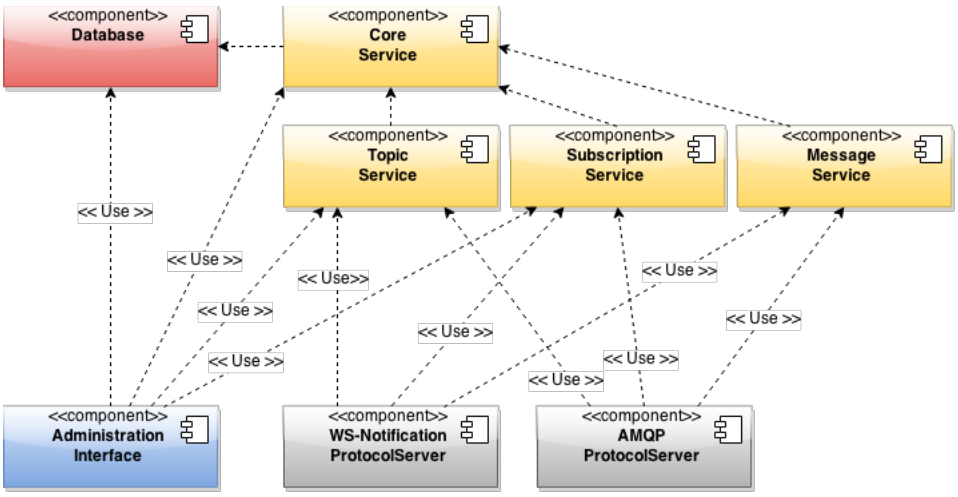
With these goals settled, it was apparent that the design outline had properties of a Service Oriented Architecture (hereby denoted SOA) [51], as well as the Module Pattern [52]. This prompted the actual implementation of the system to adhere to these principles and patterns as much as possible, for consistency and future familiarity.

Putting all this together, a high-level system architecture was created (see figure 5.1). It illustrates the main entities that were to become the brokering system. Lines between entities illustrate their connection to the core service, which is responsible for all other entity life cycles. Yellow entities are core services in the brokering system, while gray are protocol components.

5.1.2 Component dependency

The next step was to identify possible critical dependencies and usage dependencies required for a component to provide its complete set of features. This meant that a wide range of scenarios and situations, data flows and potential class structures had to be evaluated. By using the principles of SOA and Module Pattern, the amount of critical dependencies were reduced to a minimum.

Usage dependency could, however, not be reduced to a level lower than what was needed for the system to operate as intended. It was also planned to use event callbacks to handle situations that would have caused extra inter-service dependencies. This would allow services to run regardless of their real-world dependency on other services during certain tasks. An example of this is a situation where a topic is deleted in the administration interface, and subscribers have to be purged from the subscription service. The solution was to have the subscription service listen for topic-related events and update its subscriber set accordingly.



<<component>> Database
<<component>> Core Service
<<component>> Topic Service
<<component>> Subscription Service
<<component>> Message Service
<<component>> Administration Interface
<<component>> WS-Notification ProtocolServer
<<component>> AMQP ProtocolServer
<< Use >>

Figure 5.2: Component Dependencies

After much thought was put into these challenges, a component dependency diagram (see figure 5.2) was created to illustrate how the administration interface, services and protocols should depend on each other. As with the high-level architecture, core services are denoted with yellow and protocols with gray.

Dependencies between components that are annotated with $<< Use >>$ represent situations where one component depends on the other to offer its full set of features. In practice, a complete implementation of the brokering system would rely on all these dependencies to meet the specified requirements. But they should not be required for a component to perform its own internal responsibilities. Hence, a single instance of the WSN protocol should be able to run without connectivity to the rest of the system, and fully function in its local environment. This would, however, mean that the rest of the system would be unaware of the subscribers, topics and messages handled by the local instance.

5.1.3 Data flow

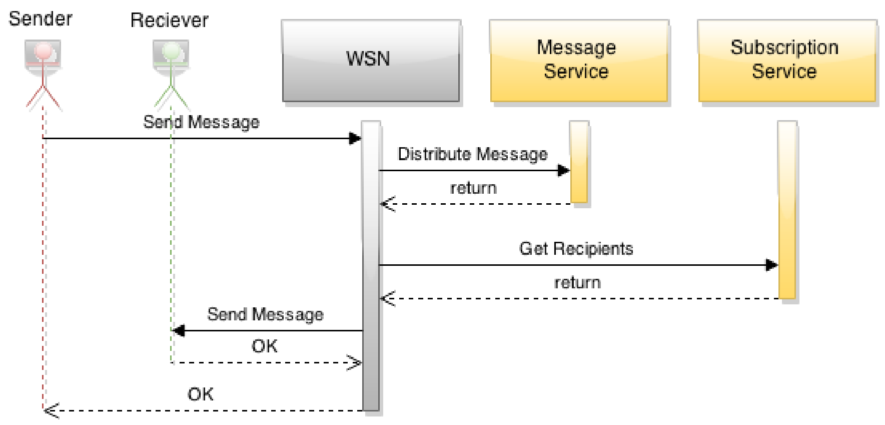


Figure 5.3: Simplified sequence diagram of possible data flow when sending a notification message using WS-Nu as the protocol library.

During the analysis of component dependencies, several potential data flows were analysed. They were used as references when evaluating which components would have critical dependencies and usage dependencies. This proved helpful, as the complexity and scope of what the group had to take into consideration began to dawn at that point. An example of one of these data flows is a possible notification message being sent. A simplified version of the data flow is illustrated in figure 5.3, and assesses requirement FR1. The figure illustrates which system calls are needed for a sender to send a message, process it, and deliver it. In practice, there might be several receivers and many additional calls, but this has been abstracted away for simplicity.

These simplified sequences were expanded later on, to provide a more realistic data flow in greater detail. During development, the detailed sequences were used as support when implementing proper start up sequences of the different components. Additionally, they were helpful for identifying possible computational bottlenecks, and in providing insight into different ways of structuring the code.

Figure 5.4 illustrates an expanded version of what the potential data flow might look like for a send notification message request using WSN. The expanded data flows show occurrences of loops, as well as some of the ideas for inner workings of the components. In this example, the send message call performed in the task runner would invoke a large branch of actions itself. To reduce complexity, the analysis of potential data flows were divided into different segments of the proposed system architecture.

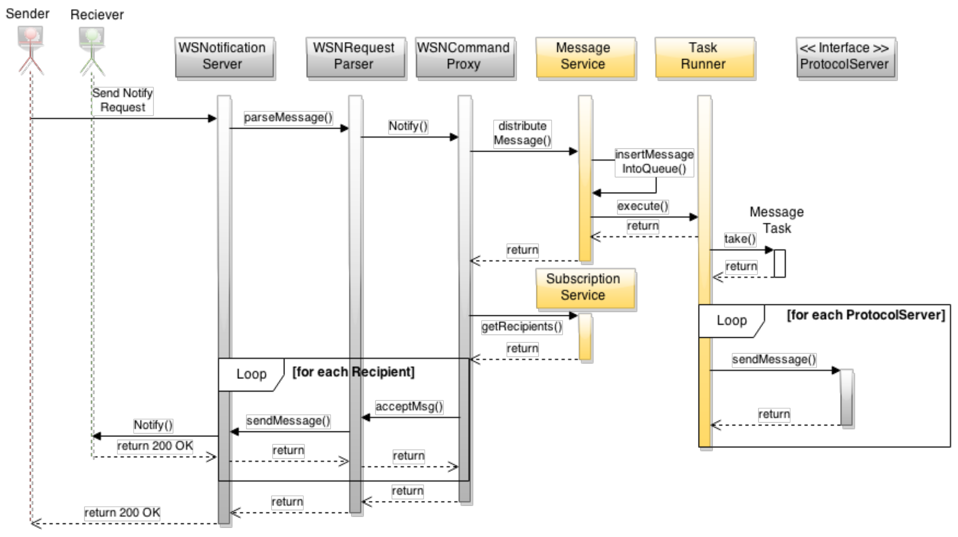


Figure 5.4: Detailed sequence diagram of possible data flow when sending a notification message using WS-Nu as the protocol library.

5.2 Broker implementation

This section provides a brief outline of implementation specific details in the OKSE message brokering system. It lists some common patterns used, choices that were made and the reasoning behind these. The main system components referred to in this section are illustrated in figure 5.1. The following sections assume a higher level of technical knowledge, and does not elaborate on every technical term used to describe the implementation. The scope of these terms fall into what can be described as general programming domain knowledge.

5.2.1 Core

As outlined in the architecture design goals, the core components of the system were implemented as standalone services, using the singleton pattern [53]. To provide a thread safe execution environment, all services were implemented with an internal task queue. This reactor thread pattern [54] acts as a demultiplexer, collecting requests for service methods from other threads and executing them synchronously by order of arrival.

CoreService

The CoreService is the main part of the OKSE message translation and brokering system. It is responsible for booting and stopping the registered core services, such as those described in subsequent sections of this chapter. Additionally, the CoreService boots up registered ProtocolServers, which are described in their own section below. The CoreService is also responsible for gracefully shutting down registered ProtocolServers.

As with all the OKSE core services, the main CoreService extends the AbstractCoreService class. This class holds some common attributes that are needed, as well as definitions of the abstract methods for `init()`, `boot()`, `run()` and `stop()` actions. All of these services are intended to follow the singleton pattern, uses static references in the implementation. Thus, the remaining needed methods and fields for initialization and instantiation are not provided by the abstract super class. They have to be implemented using a similar approach as described in the D.3 and D.6 chapters. An overview of the boot sequence of the main OKSE components is described in figure 5.5.

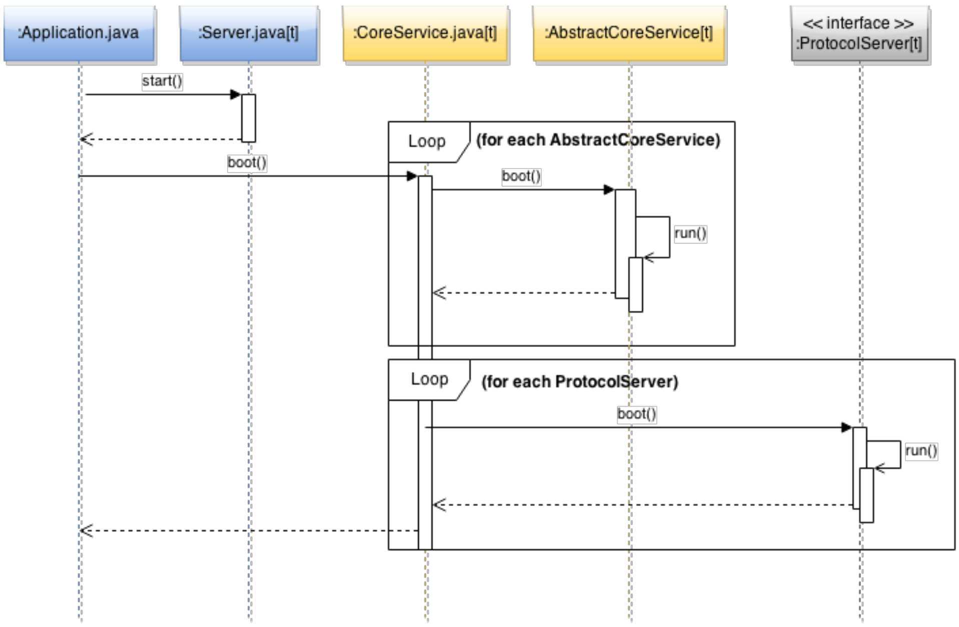


Figure 5.5: Boot sequence main components of OKSE

The details of the `init()`, `boot()` and `run()` methods will vary from service to service, and from protocol server to protocol server. After the boot sequence has completed, all of the instances of either core services or protocol servers have their own dedicated thread that awaits the next task or event to execute.

TopicService

The `TopicService` is responsible for creating, updating and deleting topics, as well as event propagation for topic related events. It runs as a single threaded reactor that consumes tasks from a blocking queue, securing thread safety. It is also supported by the

`TopicTools` class, which provides helper methods for topic tree traversal. It is worth mentioning that all topics that have subpaths will be linked together into a tree, regardless of the fact that none of the implemented protocols use this feature. They were implemented this way to account for future protocols that do support hierarchy based message distribution, and complements the non-functional requirement extendability (see section 4.4.1) in that way.

The `Topic` object itself has two ways of retrieving its name. A `getName()` which returns the name of that particular node, and a `getFullRawTopicString()` which traverses the tree until a root topic node is found. The latter will return a concatenation of the topic node names found on the path to the root node. A concatenated string of topic names is the default identifier of topics throughout the system. The `TopicService` implements listener support, which is used by other services that need to know when topics are created or deleted. This is done by implementing `TopicChangeListener` on the respective service's class, and registering it to the `TopicService` using the `addTopicChangeListener()` method. As topics may be deleted from the administration panel, or new topics could be created through the topic mapping feature, some protocol servers might need to update their local representations. Listener support was used as the preferred way of accomplishing these needs. The reason being that they will be performed as synchronized jobs that secure thread safety.

SubscriptionService

The `SubscriptionService` is responsible for the creating, updating and deleting subscribers and publishers, as well as event propagation for these types of events. It runs as a single threaded reactor that consumes tasks from a blocking queue, securing thread safety. The `SubscriptionService` implements listener support, which is used by other services that need to know when subscribers or publishers are created, updated or deleted. This is done by implementing `SubscriptionChangeListener` and `Publisher ChangeListener` respectively, and registering the class in question to the `SubscriptionService` using the `addSubscriptionChangeListener()` and `addPublisherChangeListener()` methods. As subscribers and publishers may be deleted from the administration panel, some services or protocol servers may need to update their local representations. Listener support was used as the preferred way of accomplishing these needs.

5.2.2 MessageService

The `MessageService` is responsible for distribution of messages, as well as duplicating messages if topic mappings are present. The `MessageService` delegates much of the actual work to the `CoreService`'s thread pool, which in turn schedules the messages to be sent. It runs as a single threaded reactor that consumes message distribution requests from a blocking queue, securing thread safety. When a message is received for distribution, it is added into the `MessageService`'s message queue, and consumed in order of arrival. In turn, the `MessageService` iterates through all registered protocol servers, and calls the `sendMessage()` method on each. Each of these calls are executed as a runnable job which is dispatched to the `CoreService`'s central thread pool.

It is important to note that the `MessageService` is oblivious to how protocol servers handle messages locally. This means that the `MessageService` will call the `sendMessage()` method on the same protocol server as the message was received from. This allows the protocol server implementations to choose how they handle message distribution. A requirement is that they check the `originProtocol` field of the message in their `sendMessage()` method, to prevent messages from entering an infinite loop.

Additionally, the `MessageService` keeps the latest messages sent on a topic in local cache, so that protocols like WSN that require this feature can retrieve them. In order to update the local latest message cache, the `MessageService` is registered as a listener to `TopicChangeEvents`, to respond appropriately if topics are deleted from the administration panel.

ProtocolServer

The `ProtocolServer` interface contains the common methods required in the general life cycle of a `ProtocolServer`. This interface is implemented in the `AbstractProtocolServer` class, which in turn provides some template functionality that is common to all protocol servers.

5.2.3 WSN

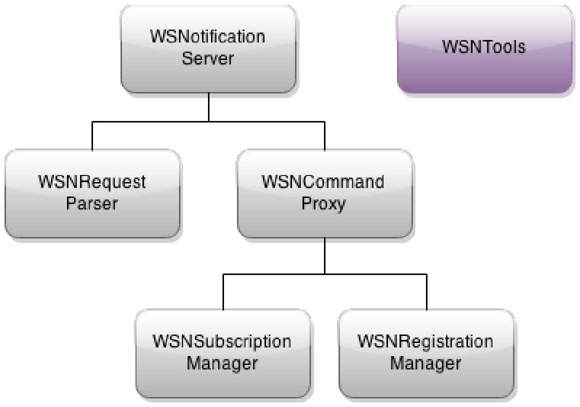


Figure 5.6: Component layout for WSN implementation

WS-Nu was chosen as the implementation library for the WSN protocol. In order to intercept messages and actions in the WS-Nu library, several of the main WS-Nu components had to be overridden and reimplemented. This led to a structure of components described in figure 5.6. The main constituents of the OKSE WSN implementation are the `WSNotification Server`, which runs the actual web server sending and receiving requests. The `WSN RequestParser`, which is responsible for parsing incoming messages and invoking corresponding methods on the appropriate component. The

WSNCommandProxy, which implements most of the methods defined in the WSN standards document. The command proxy is the parent component of the WSNSubscription Manager and WSNRegistrationManager. The two latter components are responsible for the local state and mapping between subscriber objects and publisher objects respectively.

It is important to note that there are shortcomings in the implementation of the WSN protocol. Some are based on fundamental challenges related to message translation to and from XML-based protocols, and others are errors in the WS-Nu library. These shortcomings are not listed in this chapter, but are described in detail in chapter 8.4.3.

WSNotificationServer

At the base of the WSN implementation lies the WSNotificationServer class. Like the core services, the server runs as a singleton. The component is responsible for managing an instance of the Jetty Web server, which is connected to a Handler class that processes incoming requests. The component also handles outgoing connections using a Jetty Web Client instance. When a request is received, it is processed through the Handler. If no errors have occurred during initial handling, it is passed on to the WSNRequestParser. Based on the results from the request parser call, an appropriate response message is generated and returned to the requesting client.

In the event of an outgoing message being sent, the sendMessage() method of the WSNotificationServer is invoked, and a connection is made by the Jetty Client instance to the subscribers destination address. Depending on whether the message originated from WSN or was translated from another protocol, additional steps are included. In the latter case, a WS-Nu message object is first created containing the relevant information, before dispatching it to the WSNRequestParser like any other WSN message.

WSNRequestParser

The WSNRequestParser is the OKSE implementation of what is considered a Hub in WS-Nu. All WSN requests to the OKSE message broker will pass through the request parser. The main tasks of the WSNRequestParser is to analyze requests and deliver them to the appropriate services, to generate valid outgoing message structures, and to accept WS-Nu internal messages for distribution.

It also acts as a service registry, which holds the WSNCommandProxy, WSNSubscri ptionManager and WSNRegistration Manager as potential recipient services. When accepting an internal message, the request parser will attempt to identify which of the registered services that are eligible to process the request message. If none are eligible, a standard ”not found” response message is delivered to the WSNotificationServer and relayed back to the requesting client.

WSNCommandProxy

The WSNCommandProxy is the main WSN brokering implementation. It contains most of the required methods defined in the WSNotification standard, and exposes methods for processing these different types of actions. It is implemented as a web service that is registered to the WSNRequestParser, and accepts requests for the service methods it exposes. Since the command proxy is an extension of the WS-Nu abstract notification broker class, it interacts seamlessly with the rest of the WS-Nu components. The command proxy also holds an instance of the WSNSubscriptionManager and WSNRegistrationManager classes, and delegates some of its responsibilities to these instances. This module assesses most of the requirements related to the structure of WSN, specifically FR1 to FR6 and FR10 (see section 4.3).

WSNSubscriptionManager

The WSNSubscriptionManager is responsible for keeping track of the local subscriber representations in WS-Nu and their relations to the OKSE subscriber representation. It is implemented as a web service that is registered to the WSNRequestParser, and accepts requests for the service methods it exposes. Since the subscription manager is an extension of the WS-Nu abstract subscription manager, it interacts seamlessly with the rest of the WS-Nu components.

As an example, requests to subscribe or unsubscribe is relayed through the WSNSubs criptionManager, which in turn calls the OKSE SubscriptionService. Since this is merely an intermediate manager to translate from WS-Nu to OKSE behaviour, the WSNSubscriptionManager is registered as a listener to the OKSE Subscription Service. In the event of a new subscriber, the local WS-Nu subscription representation is stored in the manager, and an addSubscriber() call is made to the OKSE SubscriptionService.

In the event of an unsubscribe request, a removeSubscriber() call is performed directly to the OKSE SubscriptionService. The manager then awaits a callback through its listener support, as a confirmation that it is okay to finally remove the local WS-Nu subscription representation. This kind of control flow allows the manager to automatically update itself. In the event of a subscriber being forcefully removed from the administration interface, or its subscription having expired in the OKSE SubscriptionServi ce, the manager will be updated through this event callback mechanism. The subscription manager assesses the functional requirements related to the subscribe mechanisms of WSN. Specifically FR7, 8 and 9 (see section 4.3).

WSNRegistrationManager

The WSNRegistrationManager is responsible for keeping track of the local publisher representations in WS-Nu and their relations to the OKSE publisher representation. It is implemented as a web service that is registered to the WSNRequestParser, and accepts requests for the service methods it exposes. Since the registration manager is an extension of the WS-Nu abstract publisher/registration manager, it interacts seamlessly with the rest of the WS-Nu components.

Similar to the WSNSubscriptionManager, the WSNRegistrationManager handles a local representation of registered publishers. It creates a local entry when a register-publisher request arrives, and delegates unregister requests directly to the OKSE SubscriptionService. This, in turn, allows the WSNRegistrationManager to automatically update itself based on the events it listens to, in the same manner as the `WSNSubscriptionManager`.

5.2.4 AMQP

The implementation of AMQP mainly consists of two parts. The first part is the handling of AMQP decoding and encoding of messages and the handling of the AMQP data in the network requests and responses. The second part is the handling of broker behaviour and the integration with the rest of the system.

As every protocol server is implemented to run on a single thread, there is a need to handle the incoming and outgoing requests/messages in an efficient way. After some consideration, the AMQP protocol server was implemented using the reactor event loop pattern [55]. The AMQP network socket was implemented using the Java non-blocking I/O library. The benefits of using a non-blocking socket is that instead of having to devote one thread per client, it will instead keep multiple channels open and perform multiplexing. This is ideal for the single thread approach, as no single client will halt the program when it has stopped sending data. After the input is received and has been demultiplexed, Qpid (see section 2.4.4) will convert the incoming data to an event, which is then added to the event queue.

Simultaneously with the event queue getting new events, the event loop is continuously dispatching the events to the correct methods on different handlers. A handler is a class that extends the `BaseHandler` class from Qpid. The handler can override event methods invoked by the event queue. The AMQP implementation has the following handlers; `AMQPServer`, `Driver`, `Handshaker` and `SubscriptionHandler`, with their own specific task.

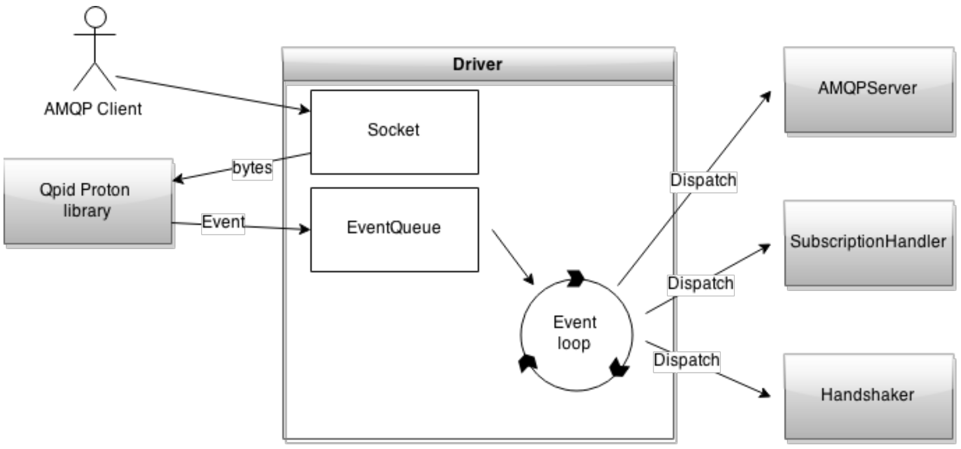


Figure 5.7: AMQP Receive implementation overview

AMQPServer

The AMQPServer handler is responsible for the incoming and outgoing messages. It handles the messages that come from AMQP producers and the internal messages from other protocols. It has an internal message queue where messages are stored until the corresponding event is triggered. It handles conversion of messages from AMQP format to the OKSE internal representation, and dispatching to the message service. It is implemented with optimizations to get AMQP messages out fast, and to prevent loops. When an AMQP message is received, it will automatically dispatch the AMQP message back to AMQP subscribers, and the OKSE message to the WSN subscribers. Figure 5.7 and 5.8 shows an architectural overview of AMQPServer.

Driver

The event loop and the socket logic is implemented in the Driver class. This class is responsible for accepting incoming connections, and handle the incoming requests. After a connection is opened with a client, every request received is passed to Qpid, which decodes the instruction and adds it back to the Driver in the form of an event. The event is added straight to a collector, which is used as the event queue. The event loop that is running continuously, will dispatch events as long as there are any in the queue. If there are no events, it will enter a wait state. The driver is responsible for the handling of requirement FR11 (see section 4.3).

SubscriptionHandler

In the `SubscriptionHandler` class, subscribers and their routes/topics are processed. Additionally, it handles the basic information about a sender/producer and the integration with the OKSE SubscriptionService. The functionality in this handler is based on addition and removal of subscribers. When a subscribe request occurs, the handler will add the user to the internal subscriber map, before adding the subscriber to the OKSE SubscriptionService. When a subscriber disconnects, it removes the disconnected subscriber in the same manner. This class assesses the the requirements FR12 and FR13 (see section 4.3), about subscribing using AMQP.

Handshaker

The Handshaker handles all of the open and close events of a connection. It makes sure that the initiation of the connection, session and link is done properly. It also ensures that it is closed properly when the link is broken or disconnected.

Queue vs Topic behavior

WSN and AMQP use different behaviour for handling topics/queues. As WSN is a pure publish/subscribe protocol, it is only concerned with topics. A description of the difference between topics and queues can be found in chapter 2.2.1.

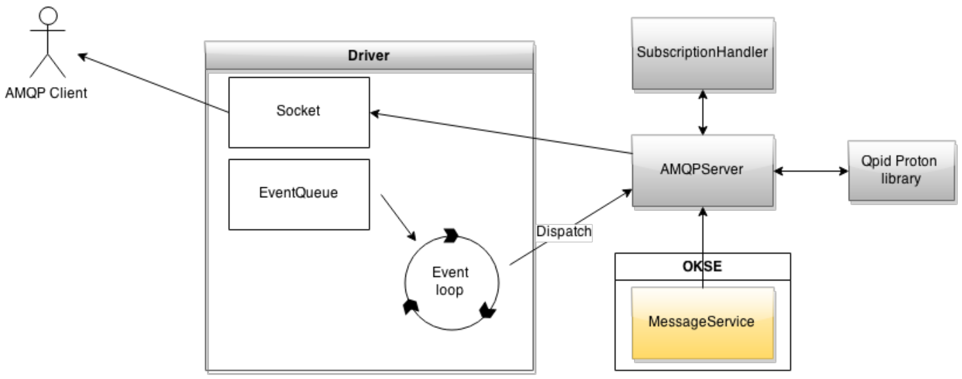


Figure 5.8: AMQP Send implementation overview

In the initial implementation, the system only complied with the given standards for each protocol. To cope with the differences, the group implemented logic to handle the conversion between the two behaviours. In addition, the group consulted the customer with the idea of implementing a non-standard topic implementation of AMQP. This meant that AMQP could be used with the same behaviour as WSN, and that all AMQP subscribers on one topic/queue would receive the message. The customer liked the idea, as it would fit nicely with their usage pattern.

A configuration switch was implemented to select which mode to use. The switch is available through the configuration file and the administration interface. The logic for this feature is implemented in AMQPServer which, as previously stated, is responsible for the outgoing message logic.

5.3 User interface

One of the initial requirements from the customer, was to have a graphical user interface. This was supposed to display relevant data and allow the administrator to moderate topics and subscriptions. After an initial consideration, the Spring Framework was chosen to be the most suitable framework (see section 2.6.4) for building the administration panel. The user interface was implemented using the MVC-pattern.

5.3.1 Design

As previously stated, a considerable amount of time was spent researching the Apollo broker during the research phase. This provided information about what an administration interface should provide. The customer had initially not stated any specific requirements for the administration panel. Therefore, an initial administration user interface wireframe was created and shown to the customer. The wireframe was heavily influenced by Apollo. This wireframe (see figure 5.9), along with the creation of use cases, set the terms for creating a prototype.

The group agreed upon creating a functional prototype for the customer. This prototype was made available, so that the customer could provide feedback. During most customer meetings, the current version of the administration interface was discussed. The customer gave feedback on how new elements in the panel met the customers expectations. This approach gave the group great advantage; rapid response from the customer, which made it easy to improve the interface and add additional features. Throughout the lifetime of the development phase, the administration panel was continuously deployed on the test server.

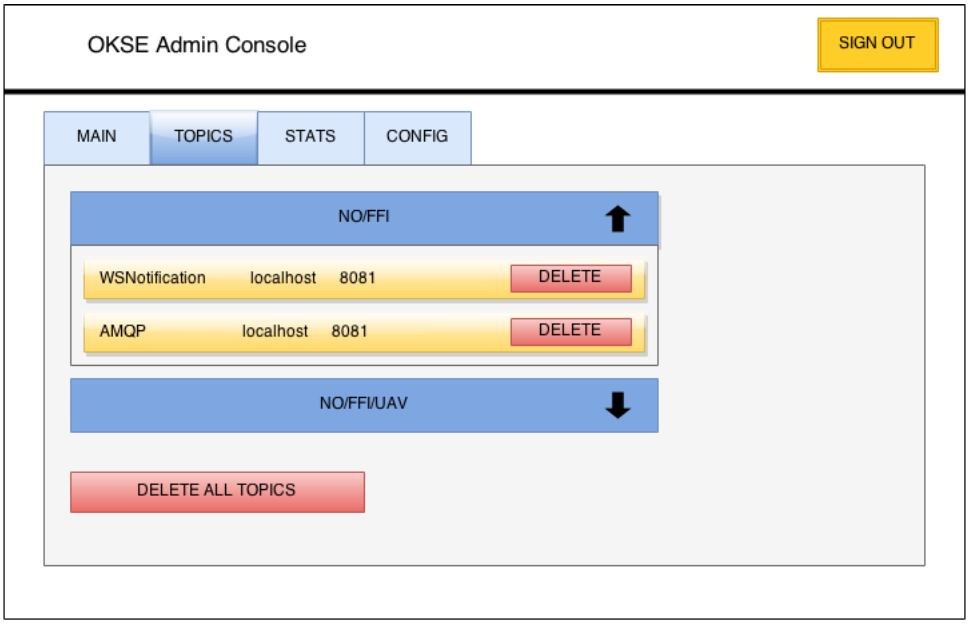


Figure 5.9: Initial website wireframe

The final interface contains six panes; ”Main”, ”Topics”, ”Subscribers”, ”Statistics”, ”Logs” and ”Configuration”. The ”Main”-pane contains information about the message broker. It is meant to give a general overview of the current operating environment, as well as the system status and memory usage. The ”Statistics”-pane contains useful statistics about requests, messages etc on the different protocols. The ”Logs”-pane contains runtime logs for debugging purposes. The ”Topics”-pane shows all registered topics in the message broker and information about the current topics, as well as the possibility to remove them. The ”Subscribers”-pane has the same functionality as the ”Topics”-pane, but for subscribers. Finally, in the ”Configuration”-pane, the user can change the brokers settings and map topics together. Also, a possibility to add relays between multiple OKSE brokers is located here. A user manual, including a visual guide to the interface, can be found in appendix C.

5.3.2 Implementation

The OKSE administration interface was designed in a modularized way, to achieve loose coupling as defined by the modularization requirement (see section 4.4.1). Each pane of the interface would have its own controller and a respective front-end JavaScript file. The advantage of this approach was having the administration panel designed as a single page web application. The page uses AJAX-requests to reach a representational state transfer (hereby denoted REST) API. By using the Spring Framework, the system exposes API-endpoints which can be used to access and/or modify data. This is the main data source for the front-end and is a critical part of the single page design. Translated to the MVC-pattern, the Spring Framework work as a controller through REST-controllers, and the different panes work as views. Each AJAX-request to the API-endpoints will reach the controllers and modify the models and/or message broker. Structuring the system this way gives some advantages:

- No need to refresh the entire administration panel.
- One controller class serves one pane; easy to debug.
- Loose coupling, no dependencies between panes.
- Easy to extend the application further.

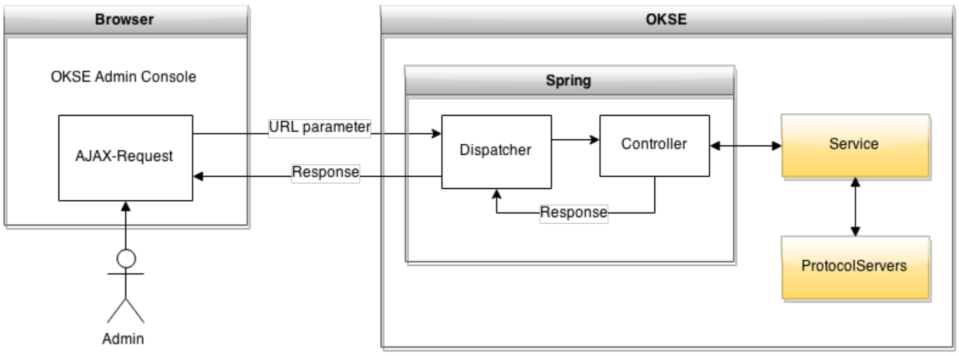


Figure 5.10: Request flow for the administration interface

Back-end

The back-end of OKSE works as a RESTful Web Service. It works as a link between the administration interface and all different registered services in OKSE. Figure 5.10 shows an overview of how a request is handled by the server. The key concepts to notice here are the AJAX-requests and the controllers. On an event, whether it is a page auto update or a button-click, the request URL is accessed by an AJAX-request. Springs dispatcher maps the URL and forwards it to the correct controller. This controller interacts with OKSE's services and returns a response.

Front-end

The front-end of OKSE is the only component of the system that is directly accessible for graphical user interaction. It is responsible for making HTTP-requests to the back-end for manipulation and presentation of the state that the system is currently in. To ensure and minimize the probability for cross-browser incompatibilities, the front-end is implemented using the jQuery framework. Each pane of the administration interface has its own module implemented with jQuery. These modules are responsible for event-handling, AJAX-requests and updating of information in their respective pane. For a more detailed description about the administration interface architecture, see appendix D.

Chapter 6

Testing

6.1 Test description

The sections below cover the different test phases. It explains the different tests, as well as discusses the reasoning and usage of the different test methodologies.

6.1.1 Unit Testing

A unit is a portion of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures. A unit test is a specific test written to test the validity of said component or components. In Java, a unit is commonly defined as an entire class, but can also be a single object method. A unit test is written to confirm that the methods will act as intended, even if other parts of the program code is changed. Unit tests were used on the parts of the system containing many logical operations and calculations. Unit tests do not, however, exclude the possibility of errors. It only reduces the amount of cases where an error might occur.

6.1.2 Integration testing

Integration testing is testing where individual modules of the software are grouped. It is a black-box test, in which major parts of the system are tested. The goal of the tests is to make sure that the major parts of the system are working as individual modules. It also ensures that modules are able to communicate with each other.

As each of the protocol implementations and core services of the system are standalone components that communicate with each other, integration testing is vital to the operation of the system.

6.1.3 Profiling

Profiling is a way to dynamically analyze the program while it is running. This is done to get an overview of where the resources are used. The method is useful to find bottlenecks and inefficient methods in the program. The software was run as a server instance, and had a possibility for high load on a wide variety of hardware. Profiling was useful to find the parts of the system that had performance issues. The system profiling was performed towards the end of the development period, as a part of the execution of system tests.

6.1.4 System testing

System testing is performed to ensure that the product meets its requirements. It is a black-box test, often performed upon completion of the product. The goal is to go through all the requirements, and ensure that they are met and working as intended. As a basis for the system testing, the group used several elements of the IEEE829 Test Plan Outline [56] as a template.

6.1.5 Acceptance Testing

Acceptance testing is generally concerned with whether or not the requirements of the system are met in a satisfiable way. Since the system has a lot of functionality that is not visible, the acceptance test focused on testing the administration interface and to send and receive messages with the implemented applications. As a precursor to the acceptance test, the group performed internal smoke tests [57] of the system after every medium to big component was implemented. General aspects that were tested were requirements compliance, look and feel of the related user interface components, and double checking the interpretation of the customers stated wishes. The reasoning for the smoke tests was to be able to find errors that could impact the result of the final acceptance test.

6.2 Test Execution

The following section discusses how tests were planned and executed for each part of the system. The goal is to give a general overview of how tests were conducted and written.

6.2.1 Unit testing

Initially, the plan was to use test-driven development as often as possible, on the components that had unit test coverage. This proved somewhat difficult. In a lot of cases it was hard to know exactly which units were needed, and as the system evolved, the units changed in scope and behaviour. Thus, tests that were initially written to verify units, were changed later on. This was something the group was well aware of, and intentionally opted to discard TDD where the scope and behaviour of a unit was developed with a trial and error approach. However, that did not mean that unit tests were discarded as a whole. After those kind of components were finished, unit tests were added to verify their validity, as well as provide a safeguard for errors caused by future alterations.

Units of the system responsible for inter-thread communication, multi-thread task processing and queuing were excluded from unit tests. It proved difficult to write tests for these scenarios, as the group lacked the necessary time and experience to write unit tests for these conditions. The testing of those types of units and components were performed during integration testing, which incorporated direct usage of these components.

The metric chosen to represent unit testing was test coverage. Test coverage is the percentage of code blocks covered by corresponding tests. All the unit tests passed validation, hence it was somewhat irrelevant to include a per-unit table of tests in this section. It was more descriptive to see how much of the code base was covered by unit tests. The actual execution of the unit tests were automated using Jenkins (see section 2.6.12), in combination with its TestNG plugin (see section 2.6.10). The following list provides an overview of the different packages in the product and their test coverage.

Name	**Status**
core	60%
core.event	85%
core.messaging	57%
core.subscription	40%
core.topic	74%
db	56%
protocol.wsn	37%
protocol.amqp	78%
web	0%

Table 6.1: Test coverage

The source code, including all the test methods, can be found in the ”test” package of the application.

6.2.2 Integration testing

The strategy used for integration testing was comprised of two parts. The first was to manually test the methods of the services and protocol modules. This first part was aimed at detecting problems with the internal dynamics of a service. A comprehensive list of test cases would not be appropriate to list in this document, as each component and service provides vastly different methods, yielding a large table. In short, all internal methods and components supporting a service’s public functionality were tested manually.

The second part was to test all the exposed methods of each service and protocol module during normal system operation. This was due to the services needing the methods of other services to perform some of the actions. In an effort to create appropriate test cases for the services and protocol modules, the best way to uncover any potential flaws was to test every service itself, as well as all other services it communicated with. In practice, this meant starting the application, and using either a pre-written test function, or the admin interface to invoke the service method in question.

Using this strategy, the group regularly performed integration testing as new features were added to the services, and new aspects of the protocol standards were implemented. This resulted in tables A.17 to A.20 of test cases and their pass criteria, listed in appendix A. All tests were performed under normal system operation.

6.2.3 System testing

The group assembled a system test plan based on the IEEE829 Test Plan outline, although excluding several sections. The below sections form the scope of the final system test plan as it was executed.

Introduction

This was the main system test plan. The plan covers the system test for all the main requirements, core services, protocol servers and support features of the OKSE Message Broker. It contains one level of testing, namely system testing.

Test items

The group assessed the functional requirements and the non-functional requirement of the system. A complete list of higher level test cases that were not directly covered by integration tests was devised. Additionally, an exhaustive list of test cases covering special cases, support features and profiling was created. The test cases for the defined requirements are found in tables A.22 through A.27 listed in appendix A. The additional system test cases are listed in table A.28 in appendix A.

Software risk issues

There are several parts of the OKSE Message Broker that were not directly in control of the group. It was therefore important that edge cases involving support features from the 3rd party libraries used were assessed as well.

The following risks affected the choice of test cases for the system test:

A. **WS-Nu and WSN standard compliance:** WSN support was the main concern of the customer, and as the protocol is the NATO standard, it holds the highest level of impact if defects are uncovered. Full standard support was desired.

B. **Qpid Proton:** The encoding of messages to be sent across the AMQP protocol needed preestimation of byte array size and enumeration until actual size is matched. This could cause a major performance hit on the AMQP protocol, and required profiling tests.

C. **Customer operational needs:** As the finished product will be used during military operational testing, support features like topic mapping and relay functionality is important for the product. Additionally, working initialization and setup of configuration directories and files are paramount to the effectiveness and features of the product.

D. **Core broker functionality.** The main task of the OKSE Message Broker is to act as a relay and translator for messages. Failure of the system to perform these core tasks would have devastating impact on the effectiveness of the product.

Components and features not to be tested

To reduce the amount of resources needed to perform the testing, some components and features were not tested during the system test. Partially because they are covered by unit or integration tests, or that they have too small related risk to the effectiveness and operation of the product. These are the components and features that were not covered by the system test:

A. The Log View pane in the administration interface, and its related controllers.

B. Support classes and components covered by unit tests and integration tests

C. Hierarchy based topic messaging. The OKSE Message Broker fully supports topic hierarchy aware messaging. Simply put, it means that subscribers to a root topic receive all messages from all children. However, none of the implemented protocols use this semantic. Additionally, the core methods of this functionality are well covered by unit tests, and was therefore deemed negligible for the system test.

Approach

The execution of the system test would be managed by the group member responsible for testing and configuration management, with support from the lead developer and involved team members. All the tests were to be performed during normal system operation. Deliverables after execution were the test results. They are included in the complete system test case table in appendix A.

Configuration management

The following configurations of the system were to be tested:

A. A clean installation with default configuration

B. An installation with custom hosts and ports set for the protocol servers, as well as the `WSN_USE_WAN` flag set to true.

C. An installation with the `AMQP_USE_QUEUE` flag set to true.

Pass criteria and results

The complete listings of pass criteria for each test case and their respective results are presented in tables A.22 through A.28 in appendix A.

6.2.4 Acceptance test

A final demonstration of the system was performed at a customer meeting on May 8 at NTNU. As previously stated, the customer did not have a strictly defined set of requirements for the product. This meant that the demonstration acted as our acceptance test. The group demonstrated the implementation of the customers requirements, as well as additional features and internal requirements from the requirements elicitation process. Due to time limitations, it was not possible to demonstrate all the different permutations of functionality acting together, as the customer had another meeting later that day. Nonetheless, the feedback given from the customer was very positive. There was some discussion regarding limitations of the WS-Nu library, as well as compatibility issues regarding translation of messages between WSN and other non-XML based protocols. Apart from one minor request for supporting topic based relaying between different instances of the product, the product as a whole was accepted by the customer. Documentation of the limitations of WS-Nu and the challenges regarding message translation were also requested. A summary of the feedback is presented below:

- Having a relay function that can forward messages of certain topics to other instances of the system.
- Support of WS-BrokeredNotification [58] and WS-BaseNotification, it was important that both were tested and documented.
- Document the research done on MQTT and MQTT libraries for further development.
- All the general requirements and expectations of the system were met.
- The interface was intuitive, and the functionality was sufficient and good.
- The name was approved, and kept for release.

These factors were taken into consideration, and these few issues were assessed prior or during the final phase of the project.

Chapter 7

Project life cycle

7.1 Planning and research

This section provides a general description of the planning and research phase of the project. It attempts to capture the main aspects of the process and work done.

7.1.1 Planning phase

The planning phase of this project consisted mainly of defining the outer scope of the project. That included negotiating core requirements with the customer, as well as research on relevant technologies and existing solutions. A report document structure was also created, and both the introduction and parts of the prestudies chapter were written. The group also had to define internal meeting and communication terms.

Goals

1. Discuss and define the project description.
2. Define core functional requirements.
3. Research previous and similar work.
4. Select development strategy and team organization.
5. Define development and communication environment.

Discussion

All of the goals were achieved in this phase. The work required during this phase was somewhat overestimated. This was mainly due to reaching agreement with the customer quickly, and the group working well together from the beginning. In total, 182 hours were spent on the planning phase, out of the 240 hours initially planned. 40 of these hours were dedicated to defining the report structure and start writing chapter 1 and 2. The remaining 58 hours could have been used more efficiently, and should have been allocated to the research phase.

7.1.2 Research phase

The research phase of this project was initially introduced due to the option of using Apollo (see section 2.4.1) as the core of the system. Thus, the only goal of this phase was to do extensive research, and find out whether to expand Apollo or develop a new system. The research work was split into different focal points, in order to assess all the major points of uncertainty (see table 2.1). At the end of this phase, a decision had to be made. This was anchored in the risk analysis, and could trigger the group to fall back to the alternative solution.

Discussion

The group was able to reach a decision regarding the system. The choice was a result of the repeated evaluation of the focal points in the associated risk analysis (see table 2.1). Two of the members were able to get a fairly good overview of the Scala programming language. However, problems arose when the code base was thoroughly inspected. It proved to be massive, and it required a lot more research to get a good enough understanding of. That also affected the time it would take to implement additional protocol modules. After hours of discussion, Apollo was ultimately discarded. The following arguments made the basis for the final decision:

- The group did not have sufficient understanding of the asynchronous workings of the embedded Scala classes
- Apollo is a fairly large and complex project, that requires a lot of effort to get a complete grasp on.
- Even though Apollo has a dynamic architecture, allowing extension by adding custom Java archived components, it requires thorough understanding of the core application.
- The group could not justify proceeding with Apollo, given the time available and the risk of not being able to gain a proper understanding in time to complete a working product.

A large amount of time had been invested in this research phase, and one might argue that it impacted the product development in a negative way. This is mainly due to lost development time. The positive aspect of it was that several good practices and ways to implement a brokering system were learned. Additionally, the group might not have been able to finish a working product in time, had the decision been to proceed with Apollo before understanding the amount of knowledge and effort required. In total, 218 hours were spent on the research phase, out of the 240 hours initially planed. 56 of these hours were spent on finalizing chapter 1 and 2. Furthermore the research done on Apollo were documented in appendix E.

7.2 Development

This section describes the iterations devoted to development of the software. The following section includes a general explanation of each sprint, as well as issues that were faced by the group. With the development phase of the project, burndown charts were introduced. The charts reflect the planned and actual time usage on development for each sprint. This was made in combination with an activity plan, which was a list of tasks to be executed for each sprint. Tasks that were not completed in a sprint were transferred to the next sprint. Each sprint documented below includes a simplified activity plan, and an example of a full activity plan is found in appendix B.4. They do not directly consider the requirements or user stories, but rather work packages from the development part of the WBS (see section 3.3.1). The burndown charts do not consider time used on non-development tasks like the report, peer evaluation, research and meetings.

7.2.1 Sprint 1

This was the first of the development cycles. It consisted mainly of setting up the development environment, and constructing the base project templates and structure. At the end of the sprint, the goal was to have a solid foundation both for the broker and for the administration interface. Additionally, the development methodology were to be documented in the report.

Task	**Status**
Initial JavaScript modules for admin console	Completed
Template for admin console	Completed
Web server instance	Completed
Decide on technology stack	Completed
Project structure	Completed
Create initial core service architecture	Completed
Initial database design and relational schema	Completed

Table 7.1: Goal achievement for sprint 1

Discussion

As expected, the first development sprint was fairly straightforward. The structure of the project was strongly influenced by Apollo. That made the work of defining the structure and base templates simpler, and less time consuming than first expected. However, due to the outcome of the research phase, time had to be used researching technologies and tools relevant to the new plan. There were also some minor deviations in the time spent on tasks versus the time that was estimated, but no specific issues arose at the time. 40 hours were spent on report writing and project management during this sprint, and 52 hours were used to research technologies.

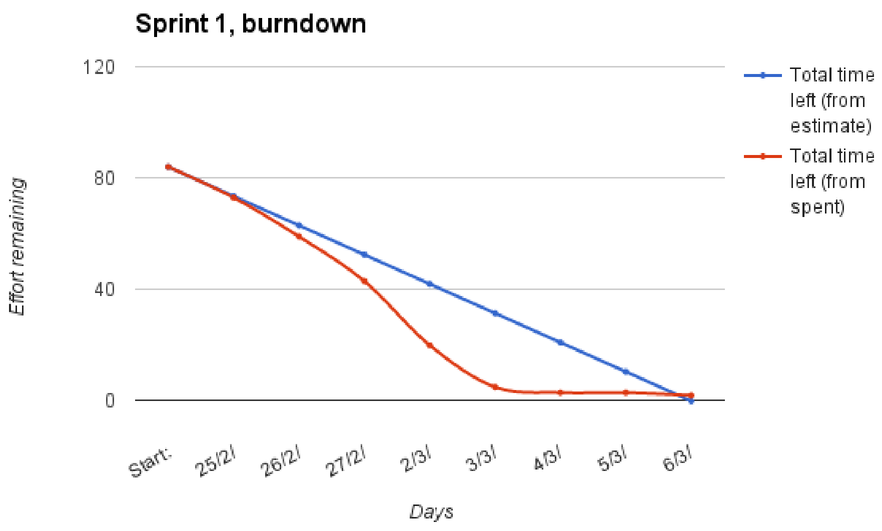


Figure 7.1: Burndown chart for sprint 1

7.2.2 Sprint 2

The major goals of sprint 2 were to initialize the first draft of the administration interface, as well as identify and implement support for processing incoming WSN messages. The midterm report was also due at the end of this sprint.

Task	**Status**
Structurize RESTapi	Not completed
Mockup for topics pane	Completed
JavaScript for creating tables for each topic the broker has a subscription on	Completed
Models for topics/stats	Completed
Stats functionality	Not completed
Identify WS-Nu components needed to intercept an incoming message	Completed
Identify WS-Nu components needed to create and deliver a valid WSN soap message	Not completed
Set up the needed components to receive and handle incoming message, and pass it to the CoreService	Not completed
Create initial core service architecture	Completed

Table 7.2: Goal achievement for sprint 2

Discussion

At the starting point of this sprint, quite a bit of content lacked on the report in order to deliver a satisfying mid term version. That prompted the group to dedicate a lot of time to the report, and 39 hours were spent on this task. Some of the work regarding WSN had to be pushed to the next sprint along with other, less important, tasks. Obviously, it would have been preferable to finish the work on time. However, quite a lot of time was left before a functional version of the WSN broker was to be completed. Thus, transferring the packages to the next sprint did not cause much concern time-wise. Additionally, the customer seemed happy with the progress, and was satisfied with the topic handling part of the user interface. One group member was ill throughout much of the sprint. This prompted a redistribution of tasks, according to the risk analysis (see section 2.7).

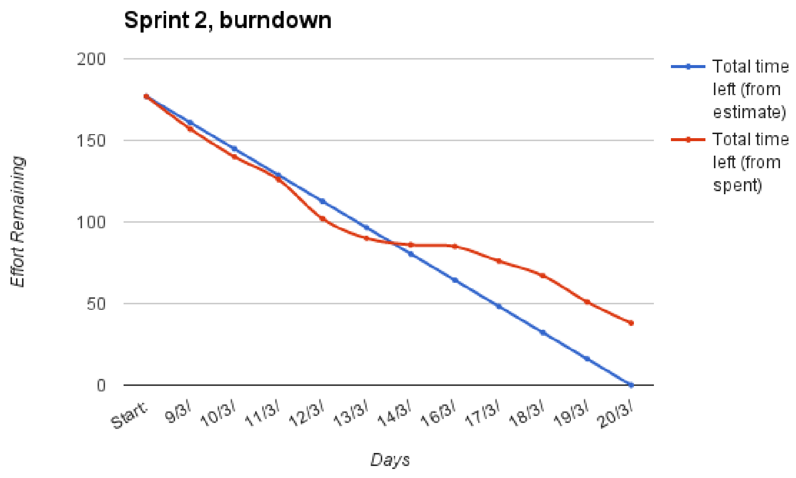


Figure 7.2: Burndown chart for sprint 2

7.2.3 Sprint 3

Only 60 hours of development were initially scheduled for sprint 3. This was due to Easter, and the fact that three of the group members were on a class trip. The main goals of this sprint were topic and subscription mapping. That included both the logic of it, and passing it to the administration interface. The incomplete packages from sprint 2 were also included.

Task	Status
Create mockup for config pane, including topic mapping	Not completed
Structurize RESTapi	Completed
Stats functionality	Not completed
Identify WS-Nu components needed to create and deliver a valid WSN soap message	Completed
Set up the needed components to receive and handle incoming message, and pass it to the CoreService	Not completed
Subscription	Not completed

Table 7.3: Goal achievement for sprint 3

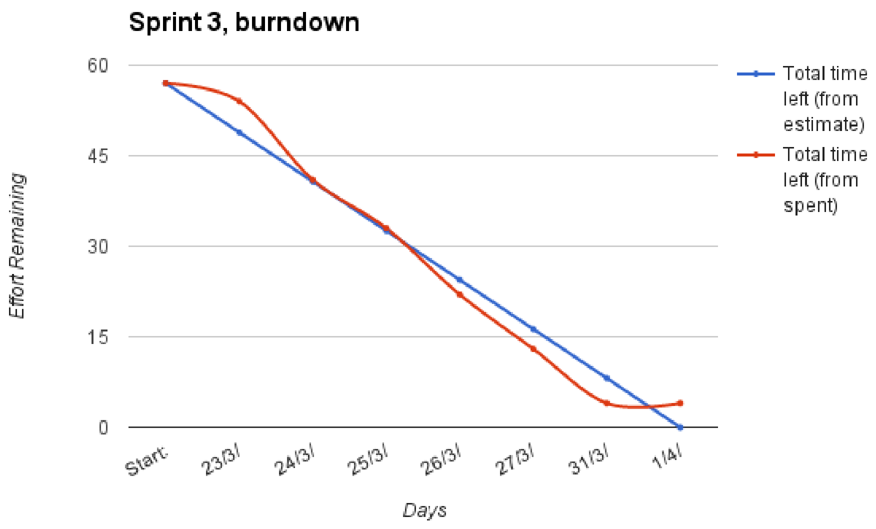


Figure 7.3: Burndown chart for sprint 3

Discussion

At this point, the progress and status of the project was not ideal. After the sprint, some of the smaller packages were still unfinished. They were left untouched due to the importance of the other parts of the system. As the system was shaping up as quite complex, more time was used to identify where and how to implement the different parts. This was obviously an issue, as it decreased the amount of code being added. 20 hours were also used on refining the first chapters of the report, which might have been a bad choice considering the shortcomings in development. However, the subscription and topic handling were almost completed at the end of the sprint. Thus, no remedial actions were taken at the time, but the issues were kept in mind.

7.2.4 Sprint 4

A lot of work had to be put into the report, the peer evaluation and general research on WSN. The group realized that finishing WSN would be difficult this sprint, so the main focus was to finalize the subscription and publish parts of it. Completing this would be a step in the right direction. One major issue was deliveries in other courses. This led to little time being scheduled for this sprint. This is something that could have been assessed earlier, and planned for better by following the risk analysis.

Task	**Status**
Create mockup for config pane, including topic mapping	Completed
Stats functionality	Completed
Set up the needed components to receive and handle incoming message, and pass it to the CoreService	Completed
Finish up custom web services and commandproxies that allow interception of requests in WS-Nu	Not completed
Create a first draft of OKSE SubscriptionManager	Completed
Create a first draft of OKSE Subscriber and Publisher objects	Completed
Create support methods for transformation to and from WS-Nu request and message representation	Not completed
Create a first draft for OKSE Topic management	Completed
Start looking into MQTT	Not completed

Table 7.4: Goal achievement for sprint 4

Discussion

The two main reasons for this sprint not containing the planned workload were major deliveries in other courses, as well as group members being in Oslo for summer job interviews. Additionally, 25 hours were spent on the report, which included time spent assessing the mid term feedback. Due to the challenges discussed in the previous sprint, the group had to discuss the priority of requirements with the customer. The solution was to re-prioritize the requirements of MQTT and AMQP support. The result was that AMQP now was the first-in-line protocol to be implemented, pushing MQTT down at a lower priority level. Previously, both AMQP and MQTT had been at the same priority level. That led to a rework of the schedule for the remaining sprints. A lot of time was used for research on the remaining parts of WSN this sprint, along with a lot of report work. WSN proved to be more comprehensive and complex than initially expected, and more code was needed for the implementation than first expected. Even with this in mind, the group still somewhat underperformed. Too little time was dedicated to actual development work. This was an issue, and the solution was to dedicate a lot of time to development the next sprint. A new work plan was made for the remaining sprints (see table 7.5).

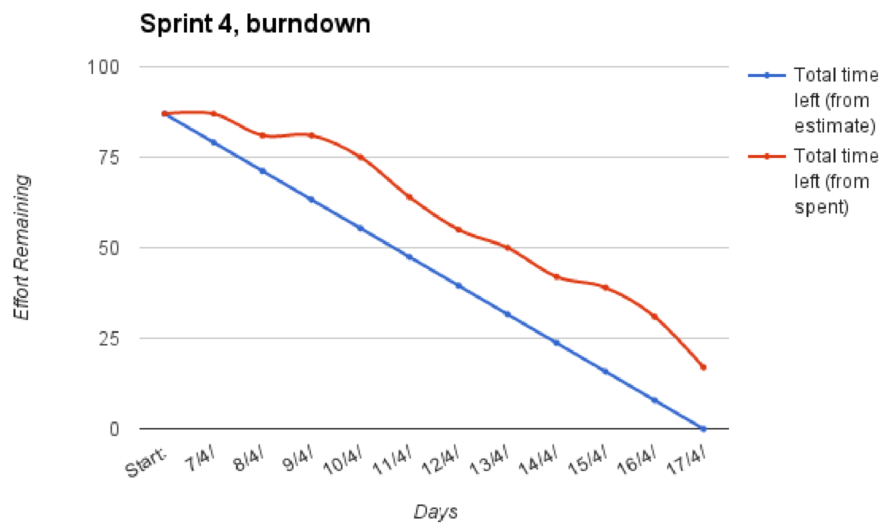


Figure 7.4: Burndown chart for sprint 4

ID	Task name	Days	Start	End	Notes
6	**Iteration 4** Stats functionality Subscribe and publish	**9**	**07.04**	**17.04**	**Monday of Orthodox Easter**
7	**Iteration 5** AMQP implementation Finalize WSN	**9**	**20.04**	**30.04**	**May 1**
8	**Iteration 6** Finalize AMQP Acceptance test	**9**	**04.05**	**15.05**	**Ascension day**
9	**Final phase** Documentation System test **Delivery**	**8**	**22.05** **30.05**	**30.05** **30.05**	 **Delivery day**

Table 7.5: Updated work plan

7.2.5 Sprint 5

At the start of sprint 5, lectures and tasks were completed in other courses. That meant that all resources could be devoted to the project. Considering the issues explained in the previous sprint, as well as the fact that the time left was getting short, a lot of work was scheduled for this sprint. Extra work hours were introduced in the evenings to assess the lack of time, in accordance with the risk analysis. The main task for this sprint was finalizing the most important aspects of WSN, as well as getting an overview of and start of implementing AMQP support.

Task	Status
Create support methods for transformation to and from WS-Nu request and message representation	Completed
Get an overview of AMQP standard	Completed
Create WSNRegistrationManager class and implement needed methods	Completed
Research, implement and test support and features needed for WSN to use different dialects	Not completed
Start on writing the developer manual	Not completed
Improve source code documentation	Not completed
Perform component testing of WSNotification	Not completed
Create subscribers-pane	Not completed
Refactor and improve UI responsiveness	Completed
Update GUI to new methods in CoreService	Completed
AMQP sendMessage	Completed
Refactorization of JS	Completed
AMQP subscribe	Not completed
AMQP Driver.stop()	Completed

Table 7.6: Goal achievement for sprint 5

Discussion

Almost all resources were assigned to implementation in this sprint, and a lot of work was done on the system. Although a fair amount of the packages were not fully completed at the end, WSN only lacked proper testing before completion. A lot of work had also been done on AMQP, and the protocol proved much simpler than WSN. Thus, the project was almost back on schedule, and it seemed likely that the implementation would be completed after the next sprint. It did, however, seem unlikely that there was sufficient time to implement MQTT. Thus, MQTT was completely abandoned during the group meeting at the end of the sprint, with compliance from the customer. The group noted that research done on the MQTT protocol, and libraries of interest should be documented in the report regardless.

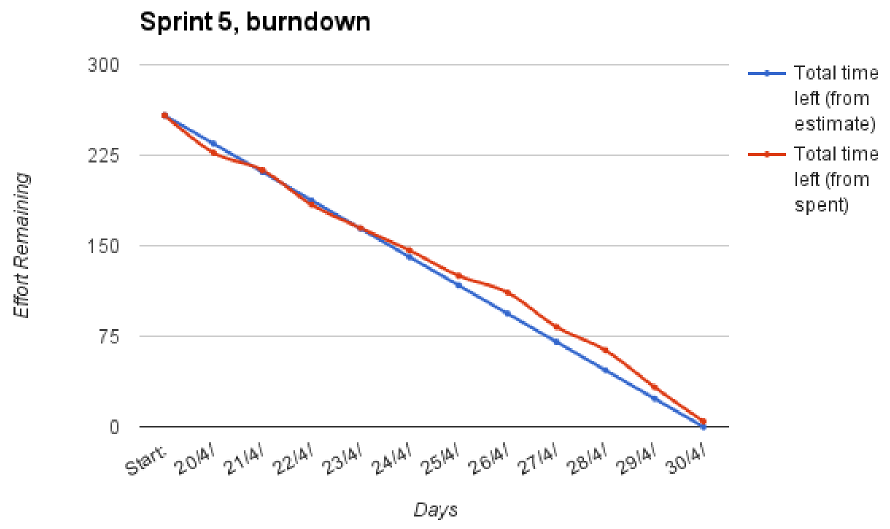


Figure 7.5: Burndown chart for sprint 5

7.2.6 Sprint 6

This was the final period of development. Only a small amount of development work remained before completion of the product. That, along with testing and a lot of report writing, were the most important aspects of the sprint. The plan was to finish up the development at the beginning of the sprint, and work on testing and report writing after that. The final days of this sprint had to be dedicated to studying for exams in other courses.

Discussion

Having exams coming up, most of the work was done at the beginning of this sprint. All tasks were completed by the end of this sprint, and both AMQP and WSN was implemented at this point. Only minor challenges arose, but were quickly overcome. The major challenges and headaches regarding WSN and AMQP were already resolved during the last sprint. With a clear remaining path, finalizing the last work packages progressed without noteworthy incidents. As the report had to start shaping up, 50 hours were also spent on report writing.

Task	**Status**
Create subscribers pane	Completed
Research, implement and test support and features needed for WSN to use different dialects	Completed
Create support for Topic Mapping in TopicService and MessageService	Completed
Hook Topic Mapping support up to admin panel	Completed
AMQP subscribe	Completed
AMQP get client port number	Completed
Database prepared statements	Completed
Graceful shutdown of AMQPProtocolServer	Completed
Create UI for relay functionality	Completed
Implement functionality for changing password	Completed
Improve source code documentation	Completed
Perform component testing of WSNotification	Completed
Perform component testing of AMQP	Completed
Test functionality for graceful shutdown and restart	Completed

Table 7.7: Goal achievement for sprint 6

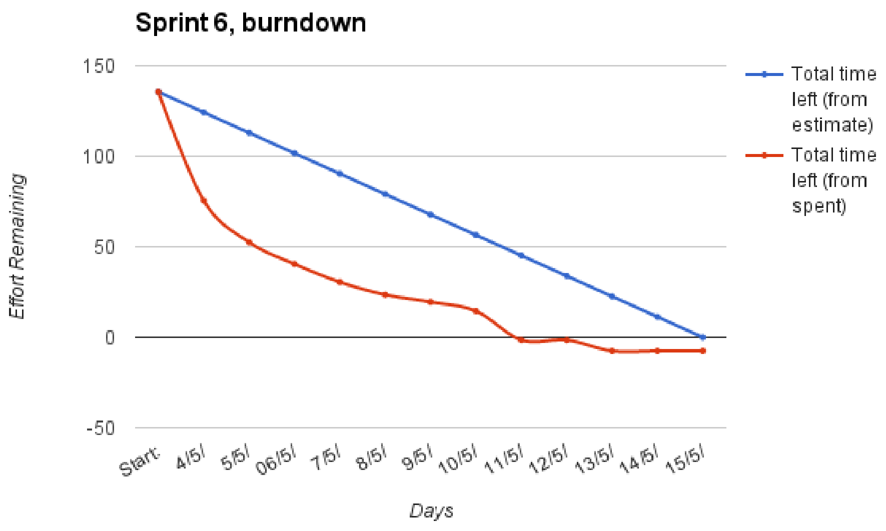


Figure 7.6: Burndown chart for sprint 6

7.3 Final phase

The final phase mostly consisted of report writing and documentation. A lot of final work and reviewing had to be done in order to deliver a satisfying report. That included writing the user and development manual, as well as an evaluation of the project. The plan was to finish the report on May 26. The report would then be handed to a third party for proofreading. The proofreading covered evaluation of structure, semantics and general language. The final delivery deadline was May 30, at which point the report would be delivered. The final phase consisted of the tasks listed in table 7.8, as well as the total amount of time spent on each task.

Task	**Time Used**
Final system testing and documentation of results	28
Project evaluation	30
Report writing	111
Creating the user manual	13
Creating the developer manual	16
Internal report proofreading and reference checking	22
Sum	220

Table 7.8: Tasks and resource usage for the final phase

Discussion

As the final deadline approached, the group had to put in large effort to finalize the formal parts of the project. The most prominent part was dedicated to the main report, focusing on the formulation of sentences, syntax, overall structure and readability. Additionally, the group put much effort into evaluating the project as a whole. This had to be done both to summarize lessons learned throughout the project, as well as documenting product related findings for the customer. An additional task during the final phase was to perform one final system test including all the stated requirements, as well as additional tests of the product. The main reason being that the group had to assure that the results provided were consistent with the behaviour of the system after development had ended. As some flaws were discovered during development, it was also important to check if they were still reproducible and, in any case, documented accordingly. This task was performed based on the plan outlined in section 6.2.3.

Chapter 8

Project evaluation

8.1 Research phase

After the initial evaluation of the product description and its requirements, it seemed like a difficult task. As none of the group members had any experience writing this kind of software, a lot of time had to be dedicated to research. Building software with this size and complexity, was unlike anything we had been taught in previous courses. Additionally, there were hundreds of pages of protocol documentation and standards, which required a lot of time to understand. Finally, there were no similar applications which implemented a message broker that included WSN. All of this required time, also during the initial development part of the project. In total, research accounted for a substantial portion of the time used in the project.

8.1.1 Apollo

Two full weeks were dedicated to research Apollo. As the total time of the project was less than one semester, that was a fairly large portion of it. The positive aspect of it, was that a lot was learned from this work, and some of the aspects and good practices were carried over to the implementation.

The decision to abandon Apollo was not made until the very end. It was only then that the components of the Apollo system proved to be so complex that it failed the gate criteria in the phase-gate model. After further discussion, additional issues which led to the final decision appeared. Had the research phase been shortened, it might have resulted in the decision to further develop Apollo, possibly resulting in a failure to deliver a working product. Thus, abandoning Apollo was probably the best choice.

8.1.2 Process Model

Although none of the group members had any experience with the phase-gate model, the choice worked out fairly well. Mainly because the first occurrence of an issue or problem that failed to meet the gate criteria would terminate further research. Subsequently, a deadline for a final decision was set, providing a firm point in time to be used in the overall project plan. This was key, as extending the research further could potentially have had a negative impact on the development part of the project. Additionally, splitting the research work into different areas, proved effective. It caused each member to have deep insight into one area, instead of all members having some insight into everything. The negative aspect of this is that it was challenging to have technical discussions with other group members.

The parallel research approach required frequent updates internally. Since each group member had a particular area of research, structured communication was key to keep the group sufficiently updated on current progress. Each member had to work in a structured manner, considering one task at a time, and continuously log progress and results.

8.1.3 Evaluation

The outcome of the research phase was not exactly as hoped. The ideal path would have been to be able to proceed with Apollo. However, the knowledge the group gained from Apollo was quite extensive, and will be useful for the customer in further research and development. The research also helped in structuring the implementation. Although a lot of time was used on Apollo, researching existing solutions is an important aspect of any development project. This was also the part of the project in which the members learned the most. Not just about Apollo and doing research work, but it was a new aspect of team work. All in all, the research work was important in order to develop a best possible solution.

Looking back at the effectiveness of the model, it proved applicable for this kind of work. In a theoretical situation, where problems might have arisen at an earlier point in time, the true effectiveness of the model would have been demonstrated. However, as previously stated, this did not occur until the end of the research phase.

In retrospect, the research period should have been compressed. By starting the research period earlier and by working more during the period, it might have been possible to cut one week off the research period. This is the most important aspect that could have been done differently. It would have led to more time available for development.

8.2 Development

Due to the research work, the development started in late February. This was a challenge, as less of the total time could be allocated to development. When constructing the development plan, the group realized that in order to complete the project, it was necessary to develop until late May. This was obviously a risk, as there would be no time to spare if something were to go wrong towards the end. Some of the less important protocols were also removed as requirements, in order to have the time to implement the most important parts.

8.2.1 Process model

Based on the arguments given in chapter 3.1.2, the group had chosen to use Scrum as a base development methodology, with elements of TDD to supplement it. However, due to the complexity of the system and the discovery of new problems and solutions to these problems, behaviour of methods often changed. This prompted frequent adjustments to the tests. The advantage should have been simple adaption to changes. Whenever new code is written, it is simpler and less time-consuming to find potential errors in existing code. Thus, greater confidence in the correctness of implemented features is achieved. This proved difficult, as many parts of the system were developed in a fashion of trial and error. Behaviour and scope of the units changed. In many situations, the group had little idea of how a component should behave, or what it should perform in contrast to delegating to other components. This was clearly a limitation of the group's use of the TDD model. In the end, only a small portion of the units' development followed the TDD approach. Perhaps with more experience with the TDD model, and writing tests under different conditions in general, greater reward would have been achieved. It simply did not seem justifiable to dedicate that much time to learn how to execute TDD properly, without jeopardizing the rest of the development. As stated in chapter 6.2.1, unit tests were mostly added after completion of components.

Scrum worked out well. The most important parts of it were the meetings and continuous releases, as they helped keeping track of progress at all times. Communication was key to assess the different development issues faced. Additionally, any problems regarding development were assessed by the group as a whole, and in collaboration with the customer.

An activity diagram and burndown chart were created each sprint. The templates were initialized at the start of each sprint, and the workload was decided and distributed amongst the group. This worked out well, as each member had clearly defined tasks for each sprint. There were some issues with assigning the different tasks, but this got easier with time. Additionally, it was helpful to keep track of the time usage when evaluating each sprint. The evaluation of the previous sprint was taken into consideration when defining the next. This was helpful to assign an appropriate workload for each sprint. During some sprints, work packages were left incomplete at the end. They were however, in most of the cases close to completion, or was not prioritized due to the importance of other tasks.

Having sprints lasting two weeks was appropriate. As the group had to consider other courses and tasks, the work load varied from week to week. Thus, it was nice to have two weeks to complete the tasks assigned. Additionally, shorter sprint intervals would have led to more time being used on management of tasks and resources, as it would have to be reevaluated each week. Having longer sprints could have affected the communication with the customer in a negative way. If something was implemented in a way that was not satisfiable, it would have led to more time being used before proper feedback was acquired. One example is the initial research made on MQTT, after it was decided that only two protocols would be implemented. At that time, it was important to know that AMQP was more important, to avoid too much time loss.

8.2.2 Resource management

Time-estimation was one of the most difficult tasks of the project. This was partially due to the uncertainty around how much time it would take to implement protocols. Some tasks proved way more time-consuming than the initial expectations. Additionally, the amount of research required to understand the protocols, and how to implement them was more than what was estimated.

During the development period, a little less time was used on development than what was planned. First of all, other courses took up a lot of time, and was not always properly planned for. The group did not properly structure the work days, and thus tasks in other courses hindered development at times. This could have been considered in a better way, by following the risk analysis and plan further ahead of time. Another factor was short term absence of group members. Although these factors were accounted for in the risk analysis, the issues that occurred were not always properly handled. The workload should have been distributed even better, and extra work hours introduced. The consequence of this was that a lot more time had to be used towards the end. The major lesson learned was to not neglect the risk analysis, and to follow up on time and resource issues that occur.

Combining report writing and development proved to be an issue. Although all of the group members contributed to both, it was difficult to keep track of both at the same time. This led to the person responsible for report and documentation having to do a lot of work on the report. While this worked out well in the process parts of the report, the responsible did not have as much insight into the development. Thus, it was difficult for him to write about the actual product. Although distributing the different parts of the report was done eventually, it should have been done earlier. This would have led to steadier progress on the report. Work distribution could have been planned better, this is one of the lessons learned.

8.2.3 Evaluation

Overall, the development period worked out well. The main issues concerned resource and time management. Not enough work was devoted to the project in the early stages of the process. This proved to be an issue later on, especially due to the task being more difficult than the initial assessment. Although this was sub-optimal, the issue was solved by adding more hours toward the end of the project. Emphasis put on test-driven development was drastically reduced as the project went on, but it did not affect the product in a considerable way. TDD aside, the Scrum process model and methodology worked well. Below is a diagram showing the work distribution among the major tasks (see figure 8.1), along with the final development burndown chart (see figure 8.2).

The final burndown chart only considers development work done in the sprints. Note that report writing was also executed during the sprints, but are not reflected in the burndown charts. A lot of emphasis was put on the report during the lifetime of the project. An important thing to note, however, is that a lot of the report work is related to project management, as this was documented throughout the project. This included tasks such as writing meeting agendas, documenting work hours and defining tasks for each sprint.

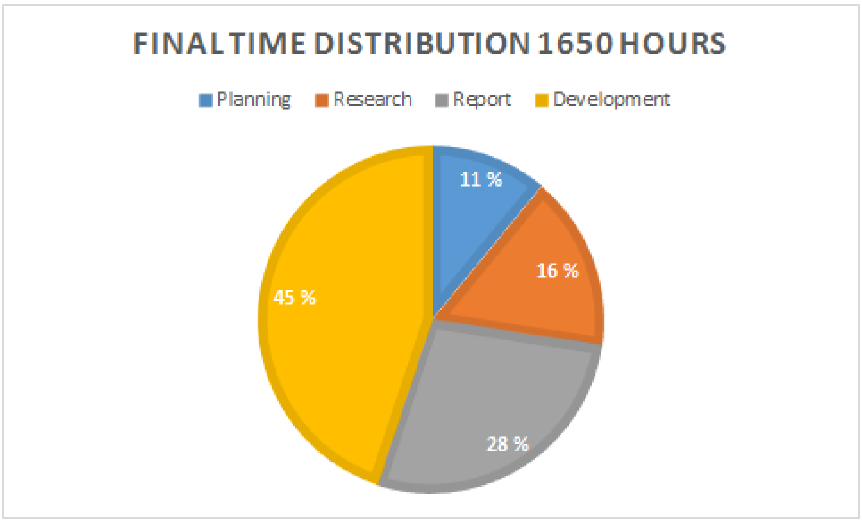


Figure 8.1: Final time distribution

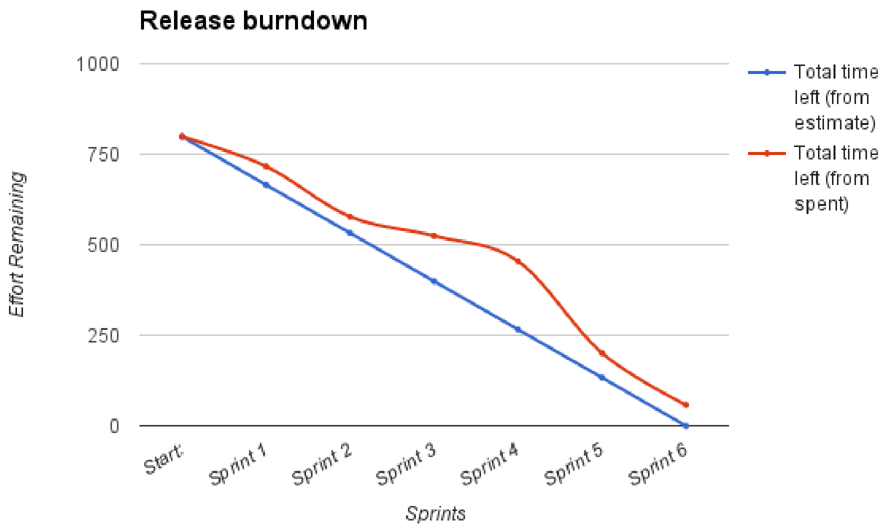


Figure 8.2: Overall burndown chart

8.3 Social and cooperational aspects

Group dynamic was rarely an issue. No social issues arose, and a nice work environment was upheld throughout the project. Eating lunch together, and taking breaks to talk about other things than the project made the days quite enjoyable. The social aspect helped throughout the project. It kept the morale of the group high, and promoted creativity. There were some minor arguments during the development phase. These were mainly due to different opinions on code principles, report structure and so forth. None of these arguments led to any anger or issues between anyone in the group, and were handled in a proper manner.

Cooperation worked out well throughout the project, both within the group, and with the customer and supervisor. It was easy to ask questions to other members, who were helpful if someone was stuck on a specific task or problem. Additionally, group meetings functioned as an arena where more complex tasks could be assessed by the whole group. All this made development more efficient, as it often prevented long periods of time being used on a specific problem. Meeting agendas were key when communicating with the customer and supervisor. The agendas provided a simple way of structuring each meeting, and they made sure nothing was forgotten or neglected.

The group also occasionally talked with other group's. The discussions were mainly about status on each group's project, and occasionally helping each other with report structure. The insight into other groups' work was helpful, as the group had someone to compare work with. This was sort of an additional safety net, to make sure no important parts of the project were neglected.

8.4 Implementation

Looking back at the group's initial expectations, compared to the challenges and complexities of the task known today, the group is very pleased with the result. What seemed like a somewhat straight forward implementation using existing libraries, turned into tough challenges and exploration.

It is worth mentioning that to the best of the group's knowledge, no other application exists that does what the group was asked to develop. The group did not manage to find a single multi-protocol brokering application supporting the WSN protocol. Additionally, general purpose clients were also very hard to find, thus resulting in the use of libraries to implement simple test clients during development. This created additional challenges, as best practices and documentation was hard to come across. Through much research, trial and error, the group eventually managed to overcome most of these challenges.

The group had to research, learn and implement several new patterns and best practices in order to develop a professional product. The steep learning curve served both as a source of frustration, as well as a source of motivation and reward. It is safe to say that the group members now have a much firmer grasp on what it takes to develop this kind of complex application. Although there are some areas of the implementation that have shortcomings, lack some unit tests, or other issues, the product as a whole works very well. The group is pleased with the degree of modularity, automation, and extendability it managed to implement.

The final system test showed that the group had met all the functional requirements. The shortcomings previously mentioned were uncovered or confirmed in the additional system tests. The main functional requirement results are listed in table A.22 through A.27. As such, the group argues that the final outcome is a good product, that delivers as expected or better. In depth explanations of the different challenges and shortcomings are described in subsequent sections below.

8.4.1 Web technologies

Bootstrap and jQuery generally made it simpler and less time-consuming to build the administration interface. Bootstrap also made it simple to give the site a clean and intuitive design. All the requirements (see section 4.3) related to the administration interface were tested and met.

In retrospect, using WebSockets [59] would have been a better approach for the administration interface. The system is now designed with AJAX for real-time update of content. This is not very scalable, and may result in high server load. Using WebSockets for this purpose would have proven much more efficient. Similar to the publish/subscribe pattern, WebSockets notify clients on a state change, instead of the clients constantly asking the server for the newest state. WebSockets would not make AJAX obsolete, but it would have been a better solution for serving true real-time update of content. AJAX would still have been used for making short-lived web service calls (e.g. HTTP DELETE requests).

8.4.2 Broker

Implementing the broker was a real challenge. Not only did the group have to figure out how to develop a working broker, but also to implement it in a way that followed the stated design and architecture goals. Questions like ”How should messages be represented?”, ”How should topics and messages be connected?” and ”How should messages be distributed?” were of paramount importance.

Much inspiration was gathered during the research phase, while investigating the Apollo broker. Several aspects of how it was designed became guidelines for how the group would implement the broker. There are vast differences of course, but some of the same patterns and techniques were used to achieve a robust implementation.

All in all the group is quite pleased with how the broker implementation turned out. It can consume messages very rapidly, and it instantly dispatches them as singular tasks in the core executor service. The executor in turn dynamically scales its number of concurrent threads to best suit the workload. This helps mitigate differences in how effective different protocols are, such that one resource heavy protocol does not cause starvation for another. In retrospect, there are ways it could have been improved of course. Allowing different priorities on message types, reuse of data structures for optimization, and a more elegant way of calling each protocol server’s `sendMessage()` method are some aspects.

Regardless, there is one high-level problem that still persists. The problem is related to translation between different protocols using vastly different representational structures. In the case of WSN, XML is used. A property in XML known as namespacing (discussed in the subsequent section) will cause translation problems with non-namespaced structures. In short, namespacing is used to allow multiple occurrences of elements with the same name. Without prior knowledge of what format the client is expecting, it seems (to the group) impossible to know which subscriber to send a non-namespaced message to. A message containing non-namespaced XML in an element called ”result” will have to either be delivered to all subscribers regardless, or by retrieving schema definitions from all the clients to test which one matches the actual data. The latter would cause a huge performance hit, and might be impossible if schemas are not available. The same goes for namespaced topic expressions used in WSN. This problem will have to be adressed for all protocols that support some form of namespacing, when translating from non-namespaced protocols.

To summarize the group’s thoughts on this particular challenge, it seems likely that this must be addressed by the operators of both the broker and the clients in use. In order to secure consistent behaviour which is predictable, some form of standard approach must be declared. Thus, the group leaves this question to system operators and their development teams, as they know their domain and usage pattern best.

8.4.3 WSN

The WSN protocol was one of the core requirements of the system, as it is the NATO standard chosen for publish/subscribe protocols. The group was provided with the WS-Nu library, an implementation of the WSN protocol previously developed by students at NTNU. In addition to this library, attempts to find similar implementations were made. With the exception of modules built into commercial enterprise systems, the group did not find a single library other than WS-Nu.

Implementing the WSN protocol from scratch was out of the question, as the amount of time it would take far exceeded the time available. The findings in the prestudy showed that the WS-Nu library seemed well written, followed some familiar patterns and was not overly complex. As such, there was really no other option than to use the WS-Nu library.

One of the first challenges of implementing the WSN protocol was to get an overview of the WS-Nu implementation itself. It is a large project, that was probably developed over the course of a semester. It is also built around the fact that the WSN protocol is designed for use as a web service. All the different components of WS-Nu therefore act as web services. This meant that the group had to re-implement many of these components to allow interception and modification where needed. This was a daunting task, and took a lot of time to get right.

Nonetheless, this period of research, trial and error also provided some inspiration on how the group could solve other problems in the OKSE system. After some time, and a series of eureka moments, development progressed smoothly and quickly, until some shortcomings, faults and design issues appeared.

Shortcomings and faults

The first of the WS-Nu shortcomings was that the act of distributing messages to subscribers was implemented in a procedural and synchronous way. This meant that any subscriber that had lost its internet connection, or had a long latency, would cause the remaining subscribers to wait. A theoretical worst-case scenario can be illustrated as 100 subscribers, where the first 99 have lost their connection. With the default connection time-out of 15 seconds, this would accumulate to a total wait time of just under 25 minutes. A totally unacceptable bottleneck. The group circumvented this problem by assigning each message a slot in the brokers internal executor service, allowing large-scale concurrency.

Shortly thereafter, issues with a property of XML called namespaces were discovered. Namespaces are used to allow multiple elements in a document to contain the same element name. To prevent two systems creating and reading documents causing trouble for each other, namespaces are used to identify which domain the element name belongs to. The WS-Nu namespace context resolver produces duplicate namespace declarations in some cases. This is not something that will likely break any applications, as most parsers will just issue a warning if duplicate namespaces are encountered.

During the implementation of a specific type of WSN topic called a `FullTopic`, issues with the expression validator were discovered. The groups additional system tests revealed that use of the $\star$ wildcard operator in the expression failed to return what should have been valid subscribers. This flaw was discovered too late, and was in a complex evaluation part of the WS-Nu library. Hence, this flaw still persists in the OKSE system, when using WSN `FullTopic` topics.

When a connection is made to send a message to a WSN subscriber, fixed content length is used until a certain message size is surpassed. This is defined internally in the Jetty HTTP client. The group discovered what seems to be a bug in the client, which causes a strange `IllegalStateException`. This was discovered after the product was considered to be finished, and emergency debugging was started. The reason seems to be that the client does not properly handle the input stream of data properly. When buffering the entire stream and sending it as a single chunk of data, no errors occur. The group tested replacing the input stream with static data converted to a generic Java `ByteArrayInputStream`. This was done to rule out the possibility that the fault lay in other parts of the system. Still, even after the change the problem persisted when the size threshold was surpassed. This reassured the group that it was indeed a bug in the Jetty HTTP client. A fix was quickly implemented to pre-buffer outgoing messages before they are sent, to prevent the failing Jetty chunked transfer encoding to occur. As it stands, the OKSE system fully supports chunked transfer-encoding on incoming requests, but does not use it on outgoing requests.

A final WSN specific issue was discovered, but after thoroughly reading the protocol documentation, the group agreed that this issue is a client standards compliance issue. Details on this peculiarity is found in additional system test STA.23 in table A.28.

Shortcomings aside, the group is pleased with how the re-implementation of WS-Nu and embedding into the OKSE message brokering system turned out. Worst-case performance bottleneck was reduced considerably, and some optimizations were performed along the way. An overview of which test cases that produced unexpected results are found in appendix A.3.

8.4.4 AMQP

During the first iteration of AMQP implementation, some challenges were uncovered. The largest one was the handling of topic and queues. As a queue is a different concept compared to a topic, the implemented solution gave the user an option to choose which solution to use. Implementing Simple Authentication and Security Layer (SASL) support for authentication was another issue that proved difficult. The solution was to let the user choose whether or not to use it. The solution was not optimal, but works as expected.

Another challenge regarding the construction of outgoing AMQP messages, was to convert a Java object to a fixed size byte array. As AMQP is a byte level protocol, this had to be resolved. The solution was to as accurately as possible calculate the objects byte size. This was done using qualified guessing by looking at the different parts of the object. After a lot of testing, the final guess only missed with between 6-12 bytes. The final size of an object was found by trial and error.

After the finalization of AMQP, the testing of the protocol implementation revealed to be more difficult than first thought. Finding applications that implements the 1.0 standard was difficult. FFI provided a version of NFFIPlayer (see section 2.4.5) with AMQP which implemented the 0.9.1 standard. However, due to the significant changes in the standard, it could not be used to any extent.

To be able to test the implementation properly, a functioning implementation was required. Two AMQP 1.0 Java implementations were found, and simple scripts were written to test both send and receive functionality. As there was a limited amount of free/open implementations, the group ended up using one open source solution from Apache Qpid (not proton) [60] and one commercial from SwiftMQ [61]. By using these implementations, the basis for stating the correctness of the implementation and the quality of the product was solid. The final test results are reflected in table A.23. All of the tests regarding AMQP passed. As the AMQP implementation was quite a lot smaller than WSN, it was easier to test.

Chapter 9

Further development

9.1 Protocols

A variety of proposals for other protocols that may be implemented are listed below. Most of them are based on the original wishes from FFI. This chapter is meant to explain the parts of the system that can or should be improved.

9.1.1 WSN

The implementation of the WSN protocol has some shortcomings, as defined in section 8.4.3. Research into improvements and workarounds for these shortcomings would be a first order of business for further development. An initial recommendation would be to replace the Jetty HTTP Client as the default client library for producing outgoing HTTP requests. This would allow removal of the pre-buffering used to prevent the chunked transfer-encoding issue with the client. This in turn would greatly improve outgoing request performance when sending large messages.

9.1.2 AMQP changes and more versions

The current implementation of AMQP will, while queue behaviour is configured, deliver the message to a random connected recipient connected to the current queue. This functionality could possibly be extended to supporting multiple algorithms for choosing. An example of this kind of algorithm can be round-robin.

It can also be useful, should the need arise, to implement an older version of the AMQP protocol. As mentioned in section 2.3.2, the older versions of the AMQP protocol are radically different, but still widely implemented.

Authentication

The system currently has support for SASL [62], but the implementation will only accept ANONYMOUS as the authentication criteria. Further development should implement a solution with actual user access.

9.1.3 MQTT

As MQTT was one of the original planned protocols, it makes sense to view this as the next step when it comes to extending OKSE with new functionality. MQTT is a pure publish/subscribe pattern and the integration with OKSE should be fairly compatible. During the research phase, when the group did research on different libraries for each protocol, the most promising Java implementation discovered was Moquette MQTT [63]. No deep research was performed, but the fact that it had a lot of traction, and that it was backed by the Eclipse foundation, gave a positive impression.

9.1.4 ZeroMQ

The next protocol that should be looked into is ZeroMQ. It was also on the wish list from FFI, but had a lower priority. ZeroMQ is by some considered to be the biggest competitor to AMQP. It was developed by iMatix Corporation [64], one of the original authors of AMQP. If ZeroMQ is considered for implementation, the AMQP implementation should be analyzed thoroughly and many of the same concepts should apply. Some research should be done on the benefits of using JeroMQ [65], which is a pure Java implementation. This implementation stands in contrast to the C implementation with Java bindings. Both versions are developed by the ZeroMQ team, and are considered ”drop-in” replacements of one another.

9.2 Web Technologies

Even though the administration interface proved to be stable during testing, some aspects of the architecture may be subject to change in the future. This section is meant to explain what parts of the administration interface that can or should be improved first.

9.2.1 WebSockets

As of now, the administration interface has not been tested with more than six simultaneous users. No issues were discovered, and the server load was minimal. But for future reference it would make sense to replace the auto update AJAX with WebSockets to reduce the server load. It would also guarantee for the users to see the most up to date information. See section 8.4.1 for more detailed information regarding the use of WebSockets contra AJAX.

References

[1] OASIS Web Services Notification (WSN) TC [Internet]. Organization for the Advancement of Structured Information Standards (OASIS); 2015 [cited 23 Jan 2015]. Available from: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.

[2] AMQP [Internet]. Organization for the Advancement of Structured Information Standards (OASIS); 2015 [cited 23 Jan 2015]. Available from: http://www.amqp.org.

[3] MQTT [Internet]. Organization for the Advancement of Structured Information Standards (OASIS); 2015 [cited 23 Jan 2015]. Available from: http://www.mqtt.org.

[4] Code Connected - zeromq [Internet]. iMatix; 2014 [cited 18 Feb 2015]. Available from: http://zeromq.org.

[5] Data Distribution Service — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 9 Jan 2015; cited 02 Feb 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Data_Distribution_Service&oldid=641720111.

[6] Subscription to topics controls the message types that reach each subscriber [Internet]. Microsoft Corporation; 2004 [cited 03 March 2015]. Available from: https://i-msdn.sec.s-msft.com/dynimg/IC141963.gif.

[7] Round-robin — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2014 [updated 10 Sep 2014; cited 20 April 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Round-robin&oldid=624937059.

[8] OASIS — Advancing open standards for the information society [Internet]. Organization for the Advancement of Structured Information Standards (OASIS); 2015 [cited 21 Jan 2015]. Available from: https://www.oasis-open.org.

[9] Hypertext Transfer Protocol — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 13 Feb 2015; cited 15 Feb 2015]. Available from: `http://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=646904728`.

[10] SOAP — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 9 Feb 2015; cited 15 Feb 2015]. Available from: `http://en.wikipedia.org/w/index.php?title=SOAP&oldid=646401386`.

[11] XML Tutorial [Internet]. W3C; 2015 [cited 15 Feb 2015]. Available from: `http://www.w3schools.com/xml/`.

[12] Web Services Base Notification 1.3 [Internet]. Organization for the Advancement of Structured Information Standards (OASIS); 2006 [updated 1 Oct 2006; cited 21 Jan 2015]. Available from: `http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.htm`.

[13] Bulding Facebook Messenger [Internet]. Lucy Zhang; 2011 [updated 12 Aug 2011; cited 10 Feb 2015]. Available from: `https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920`.

[14] Apollo [Internet]. The Apache Software Foundation; 2015 [cited 25 Feb 2015]. Available from: `http://activemq.apache.org/apollo/`.

[15] Welcome to The Apache Software Foundation! [Internet]. The Apache Software Foundation; 2015 [cited 27 Feb 2015]. Available from: `http://www.apache.org/`.

[16] Red Hat — The world's open source leader [Internet]. Red Hat, Inc; 2015 [cited 23 Feb 2015]. Available from: `http://www.redhat.com/`.

[17] OpenWire [Internet]. The Apache Software Foundation; 2011 [cited 25 Feb 2015]. Available from: `http://activemq.apache.org/openwire.html`.

[18] Stomp [Internet]; [cited 25 Feb 2015]. Available from: `http://stomp.github.io`.

[19] ActiveMQ [Internet]. The Apache Software Foundation; 2004-2011 [cited 25 Feb 2015]. Available from: `http://activemq.apache.org/`.

[20] Scala [Internet]. École Polytechnique Fédérale de Lausanne (EPFL); 2015 [cited 19 Feb 2015]. Available from: `http://www.scala-lang.org/`.

[21] RabbitMQ - Messaging that just works [Internet]. Pivotal Software, Inc; 2015 [cited 25 Feb 2015]. Available from: `http://www.rabbitmq.com/`.

[22] Home — Pivotal [Internet]. Pivotal Software, Inc; 2015 [cited 25 Feb 2015]. Available from: `http://www.pivotal.io/`.

[23] Erlang Programming Language [Internet]. Erlang Solutions; 2015 [cited 25 Feb 2015]. Available from: `http://www.erlang.org/`.

[24] WS-Nu [Internet]. Tormod Haugland and Inge Edward Halsaunet; 2014 [updated 2014; cited 30 Jan 2015]. Available from: `http://ws-nu.net/doc/wsnu_03.pdf`.

[25] Qpid Proton [Internet]. The Apache Software Foundation; 2013 [cited 20 April 2015]. Available from: `http://qpid.apache.org/proton/`.

[26] Skype — Free calls to friends and family [Internet]. Skype and/or Microsoft; 2015 [cited 21 Jan 2015]. Available from: `http://www.skype.com/`.

[27] java.com: Java + You [Internet]. Oracle Corporation; 2015 [cited 25 Feb 2015]. Available from: `https://www.java.com/en/`.

[28] Spring Boot [Internet]. Pivotal Software, Inc; 2015 [cited 25 Feb 2015]. Available from: `http://projects.spring.io/spring-boot/`.

[29] Model View Controller [Internet]; 2015 [cited 25 Feb 2015]. Available from: `http://en.wikipedia.org/wiki/Model\OT1\textendashview\OT1\textendashcontroller`.

[30] Bootstrap [Internet]; 2015 [cited 25 Feb 2015]. Available from: `http://getbootstrap.com/`.

[31] jQuery [Internet]. The jQuery Foundation; 2015 [cited 25 Feb 2015]. Available from: `http://jquery.com/`.

[32] IntelliJ IDEA [Internet]. JetBrains s.r.o; 2015 [cited 25 Feb 2015]. Available from: `www.jetbrains.com/idea/`.

[33] Apache Maven Project [Internet]. The Apache Software Foundation; 2015 [cited 25 Feb 2015]. Available from: `http://maven.apache.org/`.

[34] Gradle: Build Automation for the JVM, Android, and C/C++ [Internet]. Gradle, Inc; 2015 [cited 25 Feb 2015]. Available from: `https://gradle.org/`.

[35] Eclipse [Internet]. The Eclipse Foundation; 2015 [cited 25 Feb 2015]. Available from: `http://www.eclipse.org/`.

[36] Welcome to NetBeans [Internet]. Oracle Corporation; 2015 [cited 25 Feb 2015]. Available from: `http://www.netbeans.org/`.

[37] TestNG - Welcome [Internet]; 2015 [cited 25 Feb 2015]. Available from: `http://www.testnq.org/`.

[38] JUnit - About [Internet]; 2014 [cited 25 Feb 2015]. Available from: `http://junit.org/`.

[39] Git [Internet]; 2015 [cited 25 Feb 2015]. Available from: `http://git-scm.com/`.

[40] GitHub [Internet]. GitHub, Inc; 2015 [cited 25 Feb 2015]. Available from: https://www.github.com/.

[41] Welcome to Jenkins CI! — Jenkins CI [Internet]; 2015 [cited 25 Feb 2015]. Available from: http://jenkins-ci.org/.

[42] GanttProject: free desktop project management app [Internet]. GanttProject Team; 2015 [cited 05 March 2015]. Available from: http://www.ganttproject.biz/.

[43] Google Drive - Cloud Storage & File Backup for Photos, Docs & More [Internet]. Google; 2015 [cited 25 Feb 2015]. Available from: https://drive.google.com/.

[44] ShareLaTeX, Online LaTeX Editor [Internet]. ShareLaTeX; 2015 [cited 02 Feb 2015]. Available from: http://sharelatex.com/.

[45] Phase–gate model — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2014 [updated 5 Dec 2014; cited 28 Jan 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Phase%E2%80%93gate_model&oldid=636770760.

[46] Scrum (software development) — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 10 Feb 2015; cited 10 Feb 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Scrum_(software_development)&oldid=646437369.

[47] Breaking Down Software Development Roles [Internet]. Jupitermedia Corp.; 2006 [cited 25 Jan 2015]. Available from: http://www.hs-kl.de/~amueller/vorlesungen/se/breaking%20down%20software%20development%20roles.pdf.

[48] Systems development life cycle — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 14 Feb 2015; cited 15 Feb 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Systems_development_life_cycle&oldid=647059126.

[49] The MIT License (MIT) — Open Source Initiative [Internet]. The Open Source Initiative; 2015 [cited 09 Feb 2015]. Available from: http://opensource.org/licenses/MIT.

[50] Separation of Concerns — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2014 [updated 30 September 2014; cited 2 March 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Separation_of_concerns&oldid=627731535.

[51] Service-oriented architecture — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 13 Feb 2015; cited 15 Feb 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Service-oriented_architecture&oldid=646956163.

[52] Module pattern — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 02 March 2015; cited 04 March 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Module_pattern&oldid=649466863.

[53] Singleton pattern — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2014 [updated 11 December 2014; cited 2 March 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Singleton_pattern&oldid=63761750.

[54] Reactor pattern — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 03 Feb 2015; cited 04 March 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Reactor_pattern&oldid=645507904.

[55] Event loop — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 04 April 2015; cited 10 April 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Event_loop&oldid=654851862.

[56] Test Plan Outline (IEEE 829 Format) [Internet]; 2015 [cited 10 April 2015]. Available from: http://www.computing.dcu.ie/~davids/courses/CA267/ieee829mtp.pdf.

[57] Smoke testing (software) — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 7 May 2015; cited 14 May 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Smoke_testing_(software)&oldid=661237168.

[58] Web Services Brokered Notification 1.3 [Internet]. Organization for the Advancement of Structured Information Standards (OASIS); 2006 [updated 1 Oct 2006; cited 1 Feb 2015]. Available from: http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf.

[59] The Benefits of WebSocket [Internet]. Kaazing Corporation; [cited 04 May 2015]. Available from: https://www.websocket.org/quantum.html.

[60] Home - Apache Qpid [Internet]. The Apache Software Foundation; 2013 [cited 20 April 2015]. Available from: https://qpid.apache.org.

[61] SwiftMQ.COM [Internet]. IIT Software GmbH; 2015 [cited 05 May 2015]. Available from: http://www.swiftmq.com.

[62] Simple Authentication and Security Layer — Wikipedia, The Free Encyclopedia [Internet]. Wikipedia; 2015 [updated 19 Jan 2015; cited 06 May 2015]. Available from: http://en.wikipedia.org/w/index.php?title=Simple_Authentication_and_Security_Layer&oldid=643202396.

[63] Moquette MQTT [Internet]. The Eclipse Foundation; [cited 25 Feb 2015]. Available from: `https://projects.eclipse.org/proposals/moquette-mqtt`.

[64] Distributed systems are in our blood [Internet]. iMatix; [cited 25 Feb 2015]. Available from: `http://www.imatix.com`.

[65] zeromq/jeromq [Internet]. iMatix; [cited 25 Feb 2015]. Available from: `https://github.com/zeromq/jeromq`.

Appendix A

Tables and graphs

A.1 Use cases

Below are the different use cases created during the requirements elicitation process.

A.1.1 Use case 2 - Topic handling

This use case describes how an administrator can delete topics. The administrator must be able to delete one or all topics.

Use case ID	U2
Use case Name	Topic handling
Description	Admin should be able to delete topics.
Pre conditions	The admin is logged in and there exists one or more topics.
Standard flow	1. Click the topics pane. 2. Click delete button on the topic. 3. Alternatives: (a) Delete one topic. (b) Delete all topics.
Alternative flow	
Post conditions	Topics are deleted from the broker.

Table A.1: Use case 2 - Topic handling

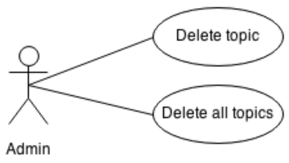


Figure A.1: U2 - Topic handling

A.1.2 Use case 3 - Server information

This use case describes how an administrator can view information about the server. This includes server statistics and message brokering statistics.

Use case ID	U3
Use case Name	Server information.
Description	As an admin I would like to see information and status about the server.
Pre conditions	The admin is logged in.
Standard flow	1. Click the pane of interest. 2. Read information on the pane of interest.
Alternative flow	
Post conditions	The admin has seen the information.

Table A.2: Use case 3 - Server information

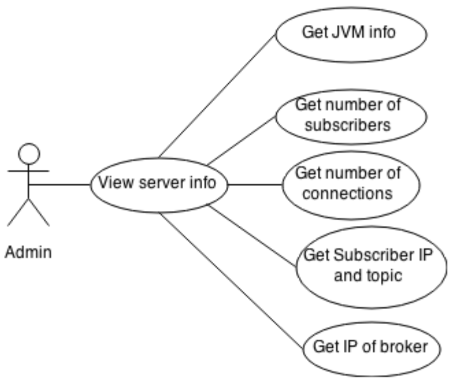

Get JVM info
Get number of subscribers
Admin
View server info
Get number of connections
Get Subscriber IP and topic
Get IP of broker

Figure A.2: U3 - Server information

A.1.3 Use case 4 - Log in

The following describes the log in procedure. The group concluded that no other functions were needed than a simple username and password log in function. Recover password were not included due to there being little to no persistent data in the database. Thus, a new instance could be initiated instead.

Use case ID	U2
Use case Name	Log in
Description	As an Admin I would like to log in to the admin console.
Pre conditions	The admin is not logged in.
Standard flow	1. Inserts username and password. 2. Clicks ”Log in”. 3. Enter the admin console.
Alternative flow	**2A:** The username or password is wrong. 1. Redirect user to front page. 2. Display error message.
Post conditions	Admin logged in.

Table A.3: Use case 4 - Log in

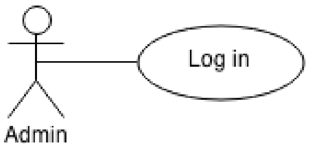


Figure A.3: U4 - Log in

A.1.4 Use case 5 - Message sending

This use case attempts to capture the process of sending messages over the system. This only captures a simplified version of the message sending. Some variations occur from protocol to protocol.

Use case ID	U5
Use case Name	Message sending
Description	User should be able to send a message over a protocol.
Pre conditions	
Standard flow	1. Register as a publisher on a topic. 2. Send the message.
Alternative flow	1A: Topic is protected. 1. User registers as publisher with username and password.
Post conditions	Message is sent.

Table A.4: Use case 5 - Message sending

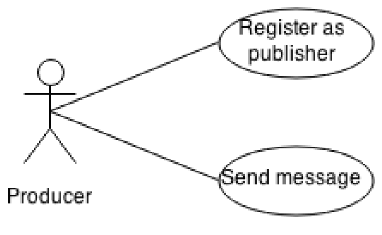


Figure A.4: U5 - Message sending

A.1.5 Use case 6 - Message Receiving

This use case attempts to capture the process of receiving messages over the system. This only captures a simplified version of the message retrieval. Some variations occur from protocol to protocol.

Use case ID	U6
Use case Name	Receiving messages.
Description	User should be able to receive a message over a protocol.
Pre conditions	The message is sent.
Standard flow	1. Register as a subscriber on a topic. 2. Receive the message.
Alternative flow	
Post conditions	Message has been received.

Table A.5: Use case 6 - Message retrieval

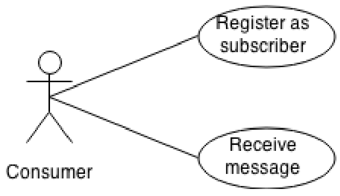


Figure A.5: U6 - Message retrieval

A.1.6 Use case 7 - Subscribe with WSN

This use case attempts to capture the process of subscribing to a topic using the WSN protocol.

Use case ID	U7
Use case Name	Subscribe with WSN.
Description	A consumer should be able to subscribe to a topic using the WSN protocol.
Pre conditions	OKSE broker is up and running.
Standard flow	1. Send subscription request to OKSE.
Alternative flow	
Post conditions	The consumer is now subscribed to the requested topic.

Table A.6: Use case 7 - Subscribe with WSN

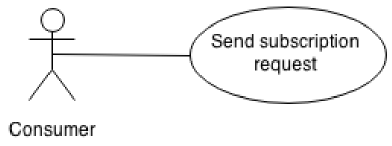


Figure A.6: U7 - Subscribe with WSN

A.1.7 Use case 8 - Register as a publisher on a topic with WSN

Use case showing how to register as a publisher on a topic using the WSN protocol

Use case ID	U8
Use case Name	Register producer WSN.
Description	A client should be able to register as publisher on a topic with the WSN protocol.
Pre conditions	OKSE broker is up and running.
Standard flow	1. Send request to OKSE. 2. Receive registration confirmation.
Alternative flow	
Post conditions	Client is now registered as a publisher on the requested topic.

Table A.7: Use case 8 - Register producer WSN

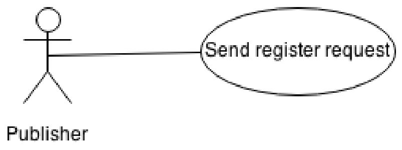


Figure A.7: U8 - Register producer WSN

A.1.8 Use case 9 - Unsubscribe with WSN

Use case showing how to unsubscribe from a topic using the WSN protocol.

Use case ID	U9
Use case Name	Unsubscribe with WSN.
Description	A user must be able to unsubscribe for a topic using the WSN protocol.
Pre conditions	OKSE broker is up and running. The user has an active subscription on the topic which the user wants to unsubscribe from.
Standard flow	1. Send unsubscribe request to OKSE. 2. Receive unsubscribe confirmation.
Alternative flow	
Post conditions	The user is now unsubscribed from requested topic and will no longer receive messages sent on that topic.

Table A.8: Use case 9 - Unsubscribe from topic WSN

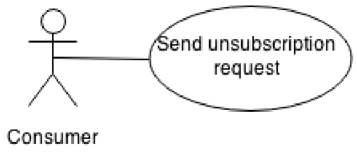


Figure A.8: U9 - Unsubscribe from topic WSN

A.1.9 Use case 10 - Unregister as a publisher on a topic wih WSN.

Use case showing how to unregister as a publisher using the WSN protocol.

Use case ID	U10
Use case Name	Unregister as publisher with WSN.
Description	A user must be able to unregister as a publisher on a topic using the WSN protocol.
Pre conditions	OKSE broker is up and running. The user is registered as a publisher on the topic which the user wants to unsubscribe from.
Standard flow	1. Send unregister request to OKSE. 2. Receive unregister confirmation.
Alternative flow	
Post conditions	The user is now unregistered as a publisher on the requested topic.

Table A.9: Use case 10 - Unregister as publisher using WSN

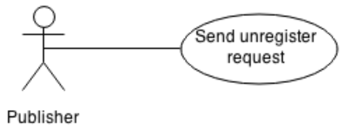


Figure A.9: U10 - Unregister as publisher using WSN

A.1.10 Use case 11 - Retrieve the latest message on a topic using the GetCurrentMessage function with WSN.

Use case showing how to retrieve the latest message on a topic using the GetCurrentMessage function.

Use case ID	U11
Use case Name	Get latest message on topic using WSN.
Description	A user must be able to get the last message sent on a topic using the WSN protocol
Pre conditions	OKSE broker is up and running. There is a message to retrieve.
Standard flow	1. Send request using GetCurrentMessage to OKSE using WSN. 2. Receive last message sent on requested topic.
Alternative flow	
Post conditions	User now has local copy of last message sent on requested topic

Table A.10: Use case 11 - GetCurrentMessage function with WSN

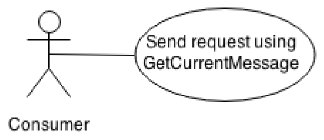


Figure A.10: U11 - GetCurrentMessage function with WSN

A.1.11 Use case 12 - Renew subscription using the WSN protocol

Use case showing the steps involved in renewing subscription on topic with WSN.

Use case ID	U12
Use case Name	Renew subscription using WSN protocol.
Description	A user should be able to renew subscription on a requested topic using WSN protocol.
Pre conditions	OKSE is up and running. The user has an active subscription on the topic which he/she wants to renew.
Standard flow	1. Send request to OKSE with new timeout date. 2. Receive confirmation from OKSE with updated subscription timeout date.
Alternative flow	
Post conditions	The user will have a updated timeout date for the subscription on the requested topic.

Table A.11: Use case 12 - Renew subscription using WSN

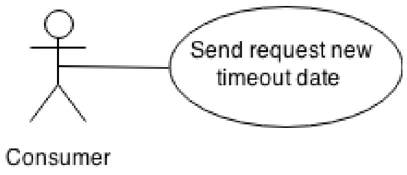


Figure A.11: U12 - Renew subscription using WSN

A.1.12 Use case 13 - Pause subscription using the WSN protocol

Use case showing the steps involved in pausing subscription on topic with WSN.

Use case ID	U13
Use case Name	Pause subscription using WSN protocol.
Description	The user should be able to pause a subscription on a requested topic using the WSN protocol.
Pre conditions	OKSE is up and running. User must have active subscription on the topic which he/she wants to pause
Standard flow	1. Send request to OKSE. 2. Receive confirmation from OKSE.
Alternative flow	
Post conditions	Subscription is now on pause. User will no longer receive messages sent on that topic. Subscription is not removed from OKSE

Table A.12: Use case 13 - Pause subscription using WSN

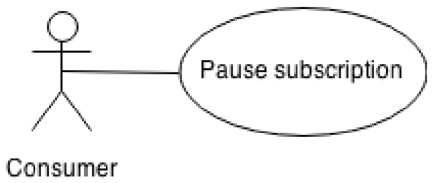


Figure A.12: U13 - Pause subscription using WSN

A.1.13 Use case 14 - Resume subscription using the WSN

Use case showing the steps involved in resuming a paused subscription on topic with WSN.

Use case ID	U14
Use case Name	Resume subscription with WSN protocol
Description	The user should be able to resume a subscription on a topic which previously was paused.
Pre conditions	OKSE is up and running. User has a paused subscription on the topic which he/she wants to resume.
Standard flow	1. Send request to OKSE 2. Receive confirmation from OKSE
Alternative flow	
Post conditions	The subscription on the requested topic is resumed. The user will now receive messages sent on the topic which the user resumed subscription on.

Table A.13: Use case 14 - Resume subscription with WSN

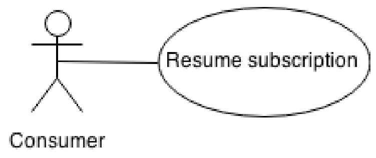


Figure A.13: U14 - Resume subscription with WSN

A.1.14 Use case 15 - Multiple notifications, single Notify using WSN

Use case showing the steps involved in sending a Notify to subscribers containing more than one notification using WSN.

Use case ID	U15
Use case Name	Multiple notification single Notify
Description	The user must be able to send multiple notification messages in a single Notify using WSN
Pre conditions	OKSE is up and running. Subscribers on the topic which the Notify is sent to.
Standard flow	1. Send one Notify containing multiple notifications
Alternative flow	
Post conditions	Subscribers receive more than one Notify on the topic which the notification was sent on

Table A.14: Use case 15 - Multiple notifications

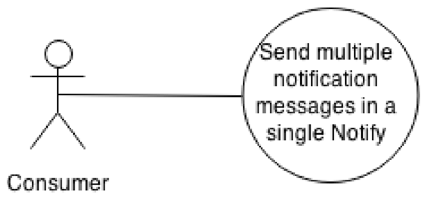


Figure A.14: U15 - Multiple notifications

A.1.15 Use case 16 - Subscribe with AMQP protocol

Use case showing the steps involved in subscribing to a topic using the AMQP protocol.

Use case ID	U16
Use case Name	AMQP subscribe
Description	The user must be able to subscribe to a topic using the AMQP protocol.
Pre conditions	OKSE must be up and running.
Standard flow	1. Connect to the server.
Alternative flow	
Post conditions	The user is now connected and subscribed to the topic of choice. Can now receive messages on that topic.

Table A.15: Use case 16 - AMQP subscribe

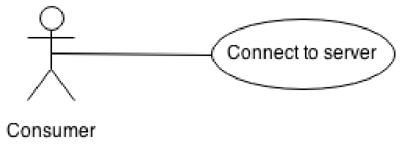


Figure A.15: U16 - AMQP subscribe

A.1.16 Use case 17 - Unsubscribe with AMQP protocol

Use case showing the steps involved in unsubscribing from at topic using the amqp protocol.

Use case ID	U17
Use case Name	Unsubscribe AMQP
Description	The user must be able to unsubscribe from a topic using the AMQP protocol.
Pre conditions	OKSE is up and running. Client is connected and subscribed to a topic.
Standard flow	1. Close connection on the desired topic.
Alternative flow	
Post conditions	Connection on the topic is closed and the subscriber no longer receives messages on that topic.

Table A.16: Use case 17 - Unsubscribe AMQP

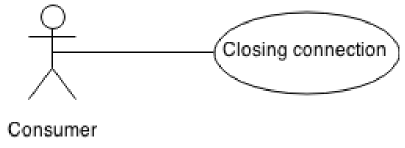


Figure A.16: U17 - Unsubscribe AMQP

A.2 Testing

A.2.1 Integration testing test cases

Core Service					
ID	**Service/Protocol**	**Method**	**Procedure**	**Expected**	**Result**
IT1.1	CoreService	boot	Start the Application	Thread running and all registered services alive	Success
IT1.2	CoreService	stop	Stop the Application	All registered services shut down and threads killed. Invoked flags set to false. CoreService thread removed and invoked flag set to false.	Success
IT1.3	CoreService	execute	Create and insert a runnable job to the executor service using the execute method	Job successfully performed	Success
IT1.4	CoreService	registerService	Initialize a stub abstractcoreservice and register it to the CoreService	Stub service is correctly registered	Success
IT1.5	CoreService	removeService	Perform IT1.4 and then remove the service again	Service correctly removed	Success
IT1.6	CoreService	addProtocolServer	Register the DummyProtocolServer	DummyProtocolServer correctly added to list of protocol servers	Success
IT1.7	CoreService	removeProtocolServer	Perform IT1.6 and then remove the protocol server again	Protocol server correctly removed	Success
IT1.8	CoreService	getTotalRequests FromProtocolServers	Perform a request using DummuProtocolServer and verify counter increment in admin panel	Request counter incremented by 1 per request	Success
IT1.9	CoreService	getTotalMessages ReceivedFromProtocolServers	Increment the DummyProtocol's received messages field	Messages received counter incremented by 1 per message	Success
IT1.10	CoreService	getTotalMessages SentFromProtocolServers	Increment the DummyProtocol's sent messages field	Messages sent counter incremented by 1 per message	Success
IT1.11	CoreService	getTotalBadRequests FromProtocolServers	Write jibberish using the DummyProtocol telnet command interface	Bad requests counter incremented by 1 per bad request sent	Success
IT1.12	CoreService	getTotalErrors FromProtocolServers	Increment the DummyProtocol's total errors field	Total errors counter incremented by 1 per error	Success
IT1.13	CoreService	getAllProtocolServers	Perform IT6.1 and verify that it is returned from the method	Set of registered protocol servers increased by 1 during registry, and DummyProtocolServer included in the shallow copy hash set returned.	Success
IT1.14	CoreService	getProtocolServer	Perform IT6.1 and then provide the Class type of the DummyProtocolServer as an argument.	The instance of DummyProtocolServer successfully returned	Success
IT1.15	CoreService	stopAllProtocol Servers	Call the method from test class	All protocol servers gracefully shut down, their threads removed and invoked flag set to false.	Success
IT1.16	CoreService	bootCoreServices	Perform 1.4 twice using 2 different stubs. Then call the method.	Both registered stub services booted successfully in their own thread, with run and invoked flags set to true.	Success
IT1.17	CoreService	bootProtocolServers	Perform IT1.6 then call the method	DummyProtocoLServer successfully booted in its own thread, with run and invoked flags set to true.	Success
IT1.18	CoreService	registerListenerSupport ForAllCoreServices	Perform IT1.4 with debug output in its registerListenerSupport method. Then invoke the test method	Debug output successfully displayed when method is invoked.	Success

Table A.17: Integration Testing - Test Cases - CoreService

Subscription service					
ID	**Service/Protocol**	**Method**	**Procedure**	**Expected**	**Result**
IT2.1	SubscriptionService	boot	Start the SubscriptionService	Thread running and awaiting new tasks. Listenersupport registered	Success
IT2.2	SubscriptionService	stop	Stop the SubscriptionService	Listener support removed, subscribers purged, thread shut down and invoked flag set to false	Success
IT2.3	SubscriptionService	registerListenerSupport	Start the SubscriptionService	Test an unsubscribe and verify that registered listenres receive call	Success
IT2.4	SubscriptionService	startScheduledRemoval OfSubscribersAnd Publishers	Start the SubscriptionService and add a subscriber with 1 min timeout	Verify that the subscriber is purged within 2 minutes	Success
IT2.5	SubscriptionService	insertTask	Start the SubscriptionService and inject a SubscriptionTask	Verify that the job is executed	Success
IT2.6	SubscriptionService	addSubscriber	Start the SubscriptionService and add a subscriber	Total subscribers increased by 1, subscriber present in local registry	Success
IT2.7	SubscriptionService	removeSubscriber	Start the SubscriptionService and add a subscriber, then attempt to remove it	Subscriber successfully removed and not present in local registry	Success
IT2.8	SubscriptionService	renewSubscriber	Start the SubscriptionService and add a subscriber. Perform a renew request with a new termination time	Subscriber is successfully updated with a the same termination time as provided	Success
IT2.9	SubscriptionService	pauseSubscriber	Start the SubscriptionService and add a subscriber, then perform the pause request	Subscriber has paused flag set to true and does not receive messages	Success
IT2.10	SubscriptionService	resumeSubscriber	Start the SubscriptionService and add a subscriber. Pause the subscriber. Perform a resume request	Subscriber pause flag is false and starts recieving messages again after the resume request has been processed.	Success
IT2.11	SubscriptionService	addPublisher	Start the SubscriptionService and add a publisher	Publisher is added and present in the local registry	Success
IT2.12	SubscriptionService	removePublisher	Start the SubscriptionService and add a publisher, then remove it	Publisher that was present is removed and no longer present in local registry	Success
IT2.13	SubscriptionService	getSubscriberByID	Start the SubscriptionService, add a subscriber. Then get it by providing its subscriber ID	The same object is returned	Success
IT2.14	SubscriptionService	getAllSubscribers ForTopic	Start the SubscriptionService and add three subscribers to a topic. Call the method specifying the same topic	All three subscribers are present in the return set	Success
IT2.15	SubscriptionService	getAllPublishers ForTopic	Start the SubscriptionService ad add three publishers to a topic. Call the method specifying the same topic	All three publishers are present in the return set	Success
IT2.16	SubscriptionService	getAllSubscribers	Start the SubscriptionService and add three subscribers to different topics, call the method	All three subscribers present in return set	Success
IT2.17	SubscriptionService	getAllPublishers	Start the SubscriptionService and add three publishers to different topics	All three publishers present in return set	Success
IT2.18	SubscriptionService	fireSubscription ChangeEvent	Start the SubscriptionService, register a subscriptionlistener, fire the event	Listener successfully invokes subscriptionChanged method	Success
IT2.19	SubscriptionService	firePublisher ChangeEvent	Start the SubscriptionService, register a publisherlistener, fire the event	Listener successfully invokes publisherChanged	Success

Table A.18: Integration Testing - Test Cases - SubscriptionService

Topic service					
ID	**Service/Protocol**	**Method**	**Procedure**	**Expected**	**Result**
IT3.1	TopicService	boot	Start the TopicService	Thread running and awaiting new tasks. Listenersupport registered	Success
IT3.2	TopicService	stop	Start the TopicService, then stop it.	Topics purged, thread stopped, invoked flag set to false	Success
IT3.3	TopicService	getAllRootTopics	Start the TopicService and add two root topics and a child topic to one of them. Call method	Only the two root topics are returned.	Success
IT3.4	TopicService	getAllTopics	Start the TopicService and add two root topics and a child topic to one of them. Call method	All three topics are returned	Success
IT3.5	TopicService	getAllLeafTopics	Start the TopicService and add two root topics and a child topic to one of them. Call method	Only the child topic is returned.	Success
IT3.6	TopicService	getTopic	Start the TopicService and add one root topic and a child topic to it. Call method with the full path to the child.	Child topic is returned.	Success
IT3.7	TopicService	getTopicByID	Start the TopicService and add two root topics and a child topic to one of them. Call method with ID to the sole root topic	The sole root topic is returned	Success
IT3.8	TopicService	getAllMappings	Start the TopicService and add two root topics and a child topic to one of them. Create a mapping from the sole root to the child topic. Call method	The sole root to child topic mapping is returned	Success
IT3.9	TopicService	getAllMappings AgainstTopic	Start the TopicService and add two root topics and a child topic to one of them. Create mapping from sole root to the child topic. Call method with child topic as argument.	The sole root topic to child topic mapping is returned	Success
IT3.10	TopicService	topicExists	Start the TopicService and add two root topics and a child topic to one of them. Call method with full string path to child topic as argument	Method returns true	Success
IT3.11	TopicService	addMapping	Start the TopicService and add two root topics and a child topic to one of them. Create mapping between sole root and the child topic	The mapping is successfully created and messages sent to sole root are received on the child topic subscribers	Success
IT3.12	TopicService	deleteMapping	Perform IT3.11 and remove the mapping again.	Mapping successfully removed	Success
IT3.13	TopicService	fireTopic ChangeEvent	Start the TopicService and register a listener to topic change events. Trigger the fireTopicChangeEvent method	Registered listener successfully invokes topicChange method.	Success

Table A.19: Integration Testing - Test Cases - TopicService

Message Service					
ID	**Service/Protocol**	**Method**	**Procedure**	**Expected**	**Result**
IT4.1	MessageService	boot	Start the MessageService	Thread running and awaiting new tasks.	Success
IT4.2	MessageService	stop	Start the MessageService and then stop it. Start it again, set broadcast system messages to true and stop it again.	First run, thread stopped and removed, invoked flag set to false, no messages broadcasted. Second run, same state, but a shutdown message is broadcasted to all subscribers before shutdown.	Success
IT4.3	MessageService	distributeMessage	Start the MessageService and CoreService. Register a dummy protocol server to the core service. Create a message and distribute it.	sendMessage is successfully invoked on the dummy protocol server	Success
IT4.4	MessageService	getLatestMessage	Perform IT4.3. Call the method	Message returned is identical to the one sent during IT4.3	Success
IT4.5	MessageService	generateMessages ForAGiven TopicSet	Start the MessageService, create 3 topics and a single message. Call the method with the message and topic set	Returned set contains 3 messages, each one containing the exact same information except they have the three provided topics that differentiate them	Success
IT4.6	MessageService	generateMessage ToAllTopics	Start the MessageService and TopicService. Add 3 different topics to the topic service. Create a message and call the method using that message	The returned set of messages contains 3 messages that are equal, except in their topic field, which corresponds to the topics registered in the topic service.	Success
IT4.7	MessageService	distributeMessage (with topic mapping)	Start the MessageService and TopicService. Create two topics. Create a mapping between the topics. Create a message bound for topic 1 and distribute it	When message is consumed, a duplicate message bound for topic 2 is created and injected into the send queue. The second, duplicate message is then consumed and dispatched to topic 2	Success

Table A.20: Integration Testing - Test Cases - MessageService

Web controllers					
ID	**Controller**	**Method**	**Procedure**	**Expected**	**Result**
IT5.1	IndexViewController	index	Start the Spring server and access route "/", then log in	Front page loaded successfully with correct information	Success
IT5.2	TopicController	getAllTopics	Start the Spring server, then add some subscriptions.	Check that the API returns all the unique topics created.	Success
IT5.3	TopicController	deleteAllTopcis	Start the Spring server, then add some subscriptions. Then click 'Delete all topics' in the administration panel.	Check that the API returns no topics.	Success
IT5.4	TopicController	deleteSingleTopic	Start the Spring server, then add one subscription. Then click 'Delete' on that topic in the administration panel.	Check that the API returns no topics.	Success
IT5.5	SubscriberController	getAllSubscribers	Start the Spring server, then add some subscriptions.	Check that the API returns all the unique subscriptions created.	Success
IT5.6	SubscriberController	deleteAllSubscribers	Start the Spring server, then add some subscriptions. Then click 'Delete all subscribers' in the administration panel.	Check that the API returns no subscribers.	Success
IT5.7	SubscriberController	deleteSingleSubscriber	Start the Spring server and add a subscription. Then click 'Delete' on that subscription in the administration panel.	Check that the API returns no subscribers.	Success
IT5.8	MainController	main	Start the Spring server, then access the Main-pane.	The main pain contains of all the information it should, like basic stats, protocolservers etc.	Success
IT5.9	MainController	powerProtocolServers	Start the Spring server, then use the administration panel to power on and off the protocol servers. Try to access one of the protocols before and after power on/off	The protocol servers should not respons after power off, and vice versa.	Success
IT5.10	LogController	log	Start the Spring server and access the Logs-pane. Access the okse.log file.	Check if the okse.log file displayed in the Logs-pane is the same as the one in the `config`-folder	Success
IT5.11	LogController	logFilesAvailable	Start the Spring server and access the API-url to get log files.	The returned JSON-string should contain the same files that is located in the `config`-folder	Success
IT5.12	LogController	logLevelsAvailable	Start the Spring server and access the API-url to get log levels.	The returned JSON-string should contain the same levels as the logLevels HashMap in the LogController.	Success
IT5.13	PasswordChange Controller	changePasswordGet	Start the Spring server, log in, and click the 'Change password'-button.	If logged in, the Controller should return the changePassword-view, else it should redirect to the indexNotLoggedIn-view	Success
IT5.14	PasswordChange Controller	changePasswordPost	Start the Spring server, log in, and click the 'Change password'-button. Change the password to `testPassword`	The user should now be able to be log in with the new password	Success
IT5.15	StatsController	getAllStats	Start the Spring server, log in. Add a WSN-subscriber. Send a message to the subscribed topic. Subscribe to the same topic with AMQP. Send a message to subscribed topic	Check the Stats-pane, and make sure the values are correctly updated. Stats should be; total request = 4, total sent = 2, topics = 1, subscribers = 2	Success
IT5.16	ConfigController	getAllMappins	Enter a mapping the the topicmapping.properties file and boot OKSE.	Check the Configuration-pane, and make sure the mappings are equal to the ones entered in the config-file	Success
IT5.17	ConfigController	addMapping	Start OKSE and enter a mapping from no/ffi to nato/hq the Configuration-pane.	Check the API-url, and make sure the mappings are equal to the one entered in the Configuration-file	Success
IT5.18	ConfigController	deleteMapping	Enter a mapping the the topicmapping.properties file and boot OKSE. Click 'Delete' on the mapping in the Configuration-pane.	Check the API-url, and make sure the returned list is empty	Success
IT5.19	ConfigController	deleteAllMapping	Enter two mappings the the topicmapping.properties file and boot OKSE. Click 'Delete all mappings' in the Configuration-pane.	Check the API-url, and make sure the returned list is empty	Success
IT5.20	ConfigController	changeAMQPaueue	Check the flag set in the okse.properties file. Then boot OKSE and change the value in the Configuration-pane	Check the log-file and make sure the value is opposite to the flag set in the okse.properties file	Success
IT5.21	ConfigController	addRelay	Boot two OKSE-brokers. Log in to one and add a relay to the other one.	Log into the other OKSE-broker and check that the other broker is a registered subscriber.	Success
IT5.22	ConfigController	deleteRelay	Boot two OKSE-brokers. Log in to one and add a relay to the other one. Then delete the relay by clicking the 'Delete'-button	Log into the other OKSE-broker and check the log file. Make sure there is one subscribe and one unsubcribe request.	Success
IT5.23	ConfigController	deleteAllRelays	Boot two OKSE-brokers. Log in to one and add a relay to the other one. Then delete the relay by clicking the 'Delete-all-relays'-button	Log into the other OKSE-broker and check the log file. Make sure there is one subscribe and one unsubcribe request.	Success
IT5.24	ConfigController	getAllInfo	Perform the first step of IT5.17 and IT5.21.	Check the API-url and make sure that the relay and topic mapping are returned.	Success

Table A.21: Integration Testing - Test Cases - Web controllers

A.3 System testing test cases

FR1-10, WSN				
Test general functionality regarding the WSN protocol implementation				
Id	**Test case**	**Procedure**	**Expected result**	**Result**
ST1.1	Send and receive message (FR1)	Create a subscriber on a topic, send valid notify to that topic	Message sent and received correctly	Success
ST1.2	Subscribe	Send a valid subscription request (FR2)	Subscriber successfully added, topic created and admin panel showing subscriber with correct host, port and topic	Success
ST1.3	Register Publisher (FR3)	Send a valid publisher registration request	The publisher is successfully registered.	Success
ST1.4	Unsubscribe (FR4)	Perform ST1.2, then perform an unsubscribe request.	Subscriber successfully removed. Topic still exists.	Success
ST1.5	Unregister (FR5)	Perform ST1.3, then perform a destroy publisher registration request using the returned publisherregistration endpoint reference.	Publisher successfully removed, total count decreased by ONE in admin panel	Success
ST1.6	GetCurrent Message (FR6)	Send a message to a topic. Send a getCurrentMessage request with the specified topic.	Last message sent correctly returned.	Success
ST1.7	Renew (FR7)	Perform ST1.2. Then send a valid Renew request.	Subscription correctly renewed.	Success
ST1.8	Pause (FR8)	Perform ST1.2. Then send a valid Pause request.	Subscription correctly paused.	Success
ST1.9	Resume (FR9)	Perform ST1.2. Then perform ST1.8. Then send a valid Resume request.	Subscription correctly resumed.	Success
ST1.10	Multiple Messages (FR10)	Perform ST1.2. Then send a Notify with TWO NotificationMessages.	System accepts the Notify and delivers to subscriber.	Success

Table A.22: System Testing - Test Cases - FR1-10

FR11-13, AMQP				
Test to see if an AMQP message can be sent, and received on all protocols				
Id	**Test case**	**Procedure**	**Expected result**	**Result**
ST2.1	Send message	Create a subscriber on a topic, send valid message to that topic	Message sent and received correctly	Success
ST2.2	Receive message	Receive AMQP message from AMQP	Message received	Success
ST2.3	Receive message	Receive AMQP message from other protocol	Message received	Success
ST2.4	Mapped topic	Send ”Hello World” via a mapped topic	Message sent on a topic will be received on mapped topic	Success
ST2.5	Subscribe AMQP	Subscribe to AMQP and update admin interface	Client is subscribed to topic and admin interface is updated	Success
ST2.6	Unsubscribe	Remove AMQP subscriber from the admin interface	When subscriber is removed from admin interface client is unsubscribed and will no longer receive messages on that topic	Success
ST2.7	AMQP protocol stats	Check AMQP statistics	Stats for AMQP is updated according to activity	Success

Table A.23: System Testing - Test Cases - FR11-13

FR14, Topic mapping				
Test to see if an topics can be mapped to each other properly				
Id	**Test case**	**Procedure**	**Expected result**	**Result**
ST3.1	Map topic	Set up a topic mapping from topic A to topic B. Send a notification message to topic A.	The system correctly duplicates the message and distributes on topic	Success
ST3.2	Two-way mapping of topics	Set up a topic mapping from topic A to topic B. Set up another topic mapping from topic B to topic A. Send a notification message to topic A.	The system correctly duplicates the message and distributes on topic B. The system cor- rectly identifies the originating topic mapping and does NOT relay back to topic A from B	Success

Table A.24: System Testing - Test Cases - FR14

FR15, Edit subscription				
Test to see if subscribers and topics can be deleted				
Id	**Test case**	**Procedure**	**Expected result**	**Result**
ST4.1	Delete subscription on a topic	Create a topic ”test”, and subscribe on the topic. Delete the subscription.	The subscription no longer exists on the topic.	Success
ST4.2	Delete all subscribers	Create topic ”test” and ”test2”, then subscribe on both topics from two clients. Click ”Delete all subscribers”.	No subscribers exists on any of the two topics.	Success
ST4.3	Delete all topics	Create two topics ”test” and ”test2”. Click ”Delete all topics”.	Both topics are deleted	Success

Table A.25: System Testing - Test Cases - FR15

FR16, Information				
Test whether or not correct information is displayed				
Id	**Test case**	**Procedure**	**Expected result**	**Result**
ST5.1	Debian linux	Launch the application on a public server using Linux Debian. Verify validity of information	Server information corresponds to the actual server specification	Success
ST5.2	Windows 8.1	Launch a local version of the application on Windows 8.1. Verify validity of information	Server information corresponds to the actual information	Success
ST5.3	Mac OSX 10.9	Launch a local version of the application on Mac OSX 10.9	Server information corresponds to the actual information	Success
ST5.4	Mac OSX 10.10	Launch a local version of the application on Mac OSX 10.10	Server information corresponds to the actual information	Success

Table A.26: System Testing - Test Cases - FR16

FR17, Log in				
Test the log in functionality				
Id	**Test case**	**Procedure**	**Expected result**	**Result**
ST6.1	Log in with correct information.	Start an instance of the system. Attempt to log in with username=admin, password=password	Access granted, and redirected to ”Main” pane.	Success
ST6.2	Log in with wrong information	Attempt to log in with username=something, password=something	Access denied, error message should appear	Success

Table A.27: System Testing - Test Cases - FR17

ID	Area	Category	Procedure	Expected	Result
STA.1	WSN	Subscribe	Send a valid subscription using SimpleTopic, and an incorrect subscription using child topics	The system accepts the valid SimpleTopic, and rejects the child topic request	Success
STA.2	WSN	Subscribe	Send a valid subscription using ConcreteTopic, and an incorrect subscription using XPATH values	The system accepts the valid ConcreteTopic and rejects the one containing XPATH	Success
STA.3	WSN	Subscribe	Send a valid subscription using FullTopic, and an incorrect subscription using invalid format	The system accepts the valid FullTopic, and rejects the invalid one	Success
STA.4	WSN	Subscribe	Send a valid subscription using FullTopic with wildcards, and an invalid subscription using FullTopic with wildcards	The system accepts the valid FullTopic, and rejects the invalid one	Success
STA.5	WSN	Subscribe	Send a valid subscription using XPATH message content filter, and an invalid subscription with an invalid XPATH expression	The system accepts the valid subscription, and rejects the invalid one	Success
STA.6	WSN	Subscribe	Send a valid subscription using XPATH topic expression, and an invalid subscription using incorrect XPATH expression	The system accepts the valid XPATH subscription, and rejects the invalid one	Success
STA.7	WSN	Subscribe	Send a valid subscription using the UseRaw element	The system accepts the subscription and messages sent to that Subscriber are not wrapped in a Notify element	Success
STA.7	WSN	Subscribe	Send a valid subscription using namespace prefixed SimpleTopic, and an invalid subscription using namespace prefixed SimpleTopic	The system accepts the valid subscription and rejects the invalid one	Success, but with duplicate namespace declarations
STA.8	WSN	Subscribe	Send a valid subscription using namespace prefixed ConcreteTopic, and an invalid subscription using namespace prefixed ConcreteTopic	The system accepts the valid subscription and rejects the invalid one	Success, but with duplicate namespace declarations
STA.9	WSN	Subscribe	Send a valid subscription using namespace prefixed XPATH topic, and an invalid subscription using namespace prefixed XPATH topic	The system accepts the valid subscription and rejects the invalid one	Success, but with duplicate namespace declarations
STA.10	WSN	Register	Send a valid RegisterPublisher using SimpleTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success
STA.11	WSN	Register	Send a valid RegisterPublisher using ConcreteTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success
STA.12	WSN	Register	Send a valid RegisterPublisher using FullTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success
STA.13	WSN	Register	Send a valid RegisterPublisher using namespace prefixed SimpleTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success, but with duplicate namespace declarations
STA.14	WSN	Register	Send a valid RegisterPublisher using namespace prefixed ConcreteTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success, but with duplicate namespace declarations
STA.15	WSN	Register	Send a valid RegisterPublisher using namespace prefixed FullTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success, but with duplicate namespace declarations
STA.16	WSN	Notify	Send a valid Notify using SimpleTopic and an invalid one	The system accepts the valid request and rejects the invalid one.	Success
STA.17	WSN	Notify	Send a valid Notify using ConcreteTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success, but with duplicate namespace declaration
STA.18	WSN	Notify	Send a valid Notify using FullTopic and an invalid one	The system accepts the valid request and rejects the invalid one	Success, but with duplicate namespace declaration
STA.19	WSN	Notify	Send a valid Notify using namespace prefixed SimpleTopic and an invalid one	The system accepts the valid request and rejects the invalid one. WS-Nu filters message correctly based on namespace prefix	Success, but with duplicate namespace declaration
STA.20	WSN	Notify	Send a valid Notify using namespace prefixed ConcreteTopic and an invalid one	The system accepts the valid request and rejects the invalid one. WS-Nu filters message correctly based on namespace prefix	Success, but with duplicate namespace declaration
STA.21	WSN	Notify	Send a valid Notify using namespace prefixed FullTopic and an invalid one	The system accepts the valid request and rejects the invalid one. WS-Nu filters message correctly based on namespace prefix	Unstable, wildcards do not seem to work. WS-Nu filtering malfunctioning.
STA.22	WSN	Notify	Send a valid Notify to a subscriber with the UseRaw flag	The system correctly extracts the message content and sends the notify without Notify wrapper	Success
STA.23	WSN	Notify	Send one Notify using the WSNotification BaseNotification namespace, and another using the WSNotification BrokeredNotification namespace.	The system correctly accepts and distributes both notifications	Success on BaseNotification. Fail on BrokeredNotification. Standard declares that BrokeredNotification imports interface from predefined Notify in BaseNotification, thus it is implied that BaseNotification should be used as namespace regardless.

Table A.28: System Testing - Additional Test Cases

STA.24	WSN	Notify	Send a valid Notify containing two bundled NotificationMessage elements	The system correctly delivers the Notify to WSN subscibers, and distributes TWO Okse internal Message objects to other protocol subscribers	Success
STA.25	WSN	Notify	Send a valid Notify containing two bundled NotificationMessage elements to a subscriber with the UseRaw flag	The system correctly extracts the message contents and sends as TWO separate messages, each containing their respective message counterparts.	Success internally in OKSE, but the MicroWSN test client cant seem to receive the two concurrent messages simultaniously.
STA.26	OKSE	Relay	Set up two OKSE instances, and set up a relay WITHOUT explicit topic declaration from one instance to the other. Send a message to the PRODUCING instance.	The relay RECEIVING instance correctly consumes the message and starts internal distribution.	Success
STA.26	OKSE	Relay	Set up two OKSE instances, and set up a relay WITH explicit topic declaration from one instance to the other. Send a message to the PRODUCING instance.	The relay RECEIVING instance correctly consumes the message and starts internal distribution.	Success
STA.27	OKSE	Profiling	Set up 3 WSN subscribers on a topic. Set up 10 AMQP subscribers on the same topic. Use a non-sleeping AMQP send script to produce 1000 messages to the topic.	All messages are successfully distributed without errors.	Success
STA.28	OKSE	Profiling	Set up 3 WSN subscribers on a topic. Set up 10 AMQP subscribers on the same topic. Use a non-sleeping AMQP send script to produce 15000 messages to the topic.	All messages are successfully distributed without errors.	Success
STA.29	OKSE	Core system	Delete the config folder. Start the application.	Config directory successfully created, all config files created with defaults from internal classpath.	Success
STA.30	OKSE	Core system	Delete the logs folder. Start the application.	Log directory successfully created, Log files created and written to.	Success
STA.31	OKSE	Core system	Use the administration interface to stop the protocol servers. Then start them again.	The protocol servers are properly shut down. Subscribers removed. Topics persist. Protocol servers are successfully started again, accepting connections.	Success
STA.32	OKSE	Core system	Use the administration interface to stop the protocol servers. Edit the host and port of one of the protocol servers. Then start them again.	The protocol servers are properly shut down. Subscribers removed. Topics persist. Protocol servers are successfully started again, accepting connections. The modified server now listens on a different host and port.	Success
STA.33	OKSE	Core system	Use the administration interface to check the ”Use AMQP queues” checkbox. Subscribe two AMQP clients on the same topic/queue. Send a message to that queue. Then uncheck the checkbox and send another message to the queue.	Only one of the subscribers receives the message when checkbox is ON, both subscribers receives message when checkbox is OFF.	Success

Appendix B

Meeting summary and activity plan examples

B.1 Introduction

This chapter shows examples of meeting summaries produced during the lifetime of the project. Status meetings with the customer was held once a week. Every second week we had a meeting with the supervisor.

B.2 Group status meeting

2015-04-17 Group meeting

Room: Skype meeting (NB!)
Time: 11.00 - 12.00

Agenda:

- Standup
- Meeting with FFI on Monday
- Testing of WS-Nu
  - What's the default behaviour when a user subscribes twice on the same topic?
  - What happens in microWSN when a user subscribes twice on the same topic?
  - If each subscription request is unique, will WSN return two messages when subscribed twice? Or will it only send one reply per endpoint.
  - What happens if we set up a WS-Nu broker implementation where demand == true (publishers must register to publish on a topic)? The last should be tested

with one WS-Nu broker running, and a publisher on a different machine (WS-Nu). It's possible to test this with microWSN, but we don't trust microWSN completely. Also we must work out how to change the returned server address to not be 0.0.0.0:8080

- Architecture
- Example data workflow for WSNotification Subscribe/publish request

Summary:

- We have progress, everybody works on their respective sprint task.
- We need to show the log-files in the admin console. Håkon, Fredrik and Kristoffer will look into this. This will ease debugging.
- Starting to connect subscribers and topic in the administration console to the different services.
- Testing of WS-Nu as stated in the agenda. Trond and Kristoffer will look into this.
- FFI meeting:
  - We must report the status on WSNotification implementation
  - Due to short time and under estimated difficulty of WSNotification, we decrease to two protocols (WSN and MQTT?)
  - Date for product delivery (ask for 15th of may?)

B.3 Supervisor meeting

2015-02-18 Meeting with supervisor

Room: ITV-160
Time: 13.00 - 14.00

Agenda:

- Group status
- Feedback on report

Summary:

- Write about quality assurance in the report
- Write about quality improvement in the report (strategy)
- Check out your requirements, and deliberate on how they interact with your process.
- How do we use the user stories? Why do we have them?
- Present the process model with figures (with boxes and arrows)

- Represent your time management with figures. Also add a Gantt diagram to visualize time available.
- Move the architecture after the project definition.
  - The same with prestudies
- Risk analysis:
  - Write about the cost of evaluating different alternatives in the Phase Gate Model
  - What's the risk of exploring two alternatives?
  - Keep in mind the cost of prevent a risk, contra the risk of handling the risk when it arises.
- Check out Learned Hand and the Calculus of negligence
- Improve the meeting agenda, so that the supervisor is more prepared

B.4 Customer meeting

2015-02-19 Customer meeting using Skype

Room: B1-120
Time: 09.00 - 10.00

Agenda:

- Project status:
- We are still in the research phase, looking deep into Apache Apollo.
- Successfully built it from the repository. Was quite a challenge.
- Currently two people are working on breaking down Apollo, while two others are looking into Scala.
- WS-Nu will be used as a library for WSNotification implementation
- Started sketching up the web interface (two people are working on this)
- Show the wireframe of the web interface
- Ask about the report, and possibility for read through.
- Information about ZeroMQ

Summary:

It should be possible to map between different topic dialects. Also mapping between concrete-topic $\rightarrow$ simple-topic and vice versa. This is because some receivers are not capable of receiving all topic dialects. WSNotification does not support that a broker sends what kind of dialect it supports. We need to keep this problem in mind.

We have successfully built Apollo in IDEA, with all dependencies. Still in the research phase, so it's quite a lot of work to do here. Two people are currently looking into Scala. By Wednesday, we will come to a conclusion on whether to use Apollo or not.

FFI would like detailed information about how WS-Nu is built and how we are going to use it. FFI would also prefer to get a copy of the report every time significant changes has been done.

WS-Nu does also have an non open-source implementation of WS-Cache (an internal NATO specification), but we do not need to address this, keep it in mind though.

B.5 Activity plan, sprint 2

Activity plan sprint 2, 9/3 - 20/3

15 Studiepoeng means every resource (team member) works approximately 20 hours a week.

			Planning				Follow-up		
Nr	**Work Package**	**Activity**	**Resource**	**Planned work (hrs)**	**Start**	**Finish**	**Actual work (hrs)**	**Status (% complete)**	**Comment**
1	Development	Structurize RESTApi	Tørnvall, Løvdal	20	9/3/	20/3/	20	90	Just lacks some bug fixing
2	Development	Create mockup for topics pane	Tørnvall, Løvdal	15	9/3/	17/3/	13	100	
3	Development	Create javascript for creating tables for each topic the broker has a subscription on	Løvdal	10	9/3/	12/3/	11	100	
4	Development	Create models for topics/stats	Tørnvall	5	9/3/	10/3/	4	100	
5	Development	Stats functionality	Tørnvall	10	13/3/	18/3/	11	40	Needs more data in order to complete, going to
6	Development	Identify WS-Nu components needed to intercept an incoming message	All	50	10/3/	20/3/	51	100	We have a fairly well understanding at this point
7	Development	Identify WS-Nu components needed to create and deliver a valid WSN soap message	All	40	11/3/	20/3/	16	60	Shouldn't require as much work as initially thought. Should be able to complete this in the next sprint
8	Development	Set up the needed components to receive and handle an incoming message, and pass it to core service	Skraastad, Walleraunet, Dalby	24	11/3/	20/3/	11	40	Requires some more research.
9	Development	Create initial core service architecture	Skraastad	3	10/3/	10/3/	2	100	Initial sketch, might need update at a later point
10	Report	Finish satisfying mid-term version	Berg, all for review	40	10/3/	18/3/	39	100	Finished, could had more content regarding implementation

Appendix C

User Manual

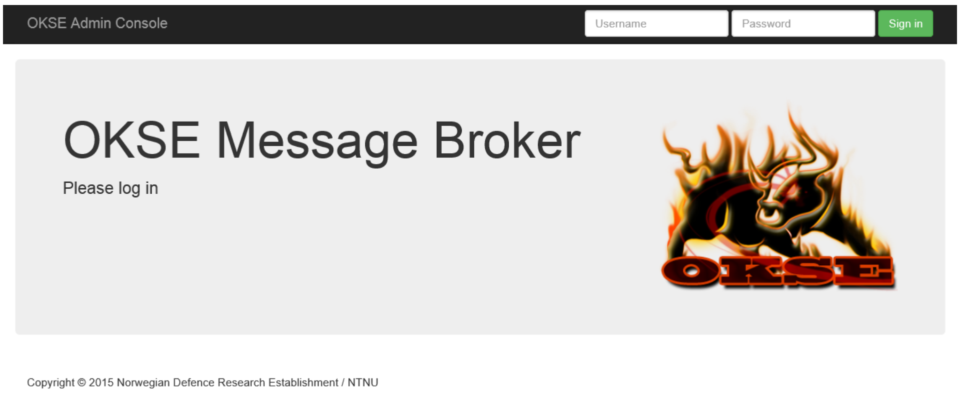


Figure C.1: The OKSE Admin Console - Login page

C.1 About

This chapter contains instructions for installing and using the OKSE Message Broker. The intended users are primarily researchers working with collaborative networking. OKSE is open source, and licensed under the MIT license.

C.2 Requirements

OKSE Message Broker requires Java Runtime Environment (hereby denoted JRE) version 8 update 31 or newer. The broker is developed and tested exclusively with this version.

The software is available for free from `http://java.com`[1].

For the JRE to function properly, it must be added to the path variable. On UNIX-based systems, like Linux or OS X, it is done by the Java installer. On Windows, open up a command line window (cmd.exe) and type in:

```
path %path%;<PATH-TO-JRE>
```

Usually this can by done on Windows by entering:

```
path %path%;C:\ProgramData\Oracle\Java\javapath
```

The variable can also be accessed and modified through the ”Environmental variables” tab in the advanced system settings. The system should be possible to run on all platforms with JRE installed. In addition to JRE, OKSE requires a network connection to be able to send and receive messages. It does not however, require Internet on the connection.

C.3 Installation

The following sections describes how to download and run OKSE. Apart from JRE, no other additional configuration should be necessary.

C.3.1 Download software

Download the latest version of OKSE Message Broker from `http://okse.fap.no/` or use the version provided with this document. The files can be placed at any location within the filesystem, as long as the user has write permission in the target folder. It is recommended that all files are placed in the same folder.

C.3.2 Running the software

Using the start script

Within the main folder of the software package, there are two scripts, `start.bat` and `start.sh`. The scripts will automatically try to detect where Java is installed and start the application.

With Windows, the application can be started by double clicking `start.bat`. This will bring up a console window with the application. To close the application, simply close the window.

With Unix/Linux/OS X, the script should be run with a terminal. As there is a lot of different versions of Linux/Unix, consult the documentation of the distribution on how to do this.

With OS X, the script can be started by following these steps:

[1] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

1. Right click `start.sh`
2. Select Open With → Other...
3. Go to Utilities
4. Select Terminal.app (Select Enable: All Applications)

Using command line

On UNIX-based systems and using cmd.exe on Windows, the broker is started by running these commands from the command line or terminal:

```
cd <PATH-TO-OKSE>
java -jar okse.jar
```

If the broker is going to be used for bigger files or a very high amount of messages, a higher memory limit should be set for Java. Note that it is not possible to set the memory limit higher than the amount of memory that is available: The following command changes the memory limit:

```
java -jar -Xms2048m -Xmx4096m okse.jar
```

See section C.5 for information about accessing the administration panel.

C.4 Configuration

After the initial start up of OKSE Message Broker, a folder named `config` is created in the same location as the .jar-file. This folder contains three configuration files with default configuration:

- okse.properties
- log4j.properties
- topicmapping.properties

See section C.7 for detailed information about all the possible settings.

C.5 Inital login

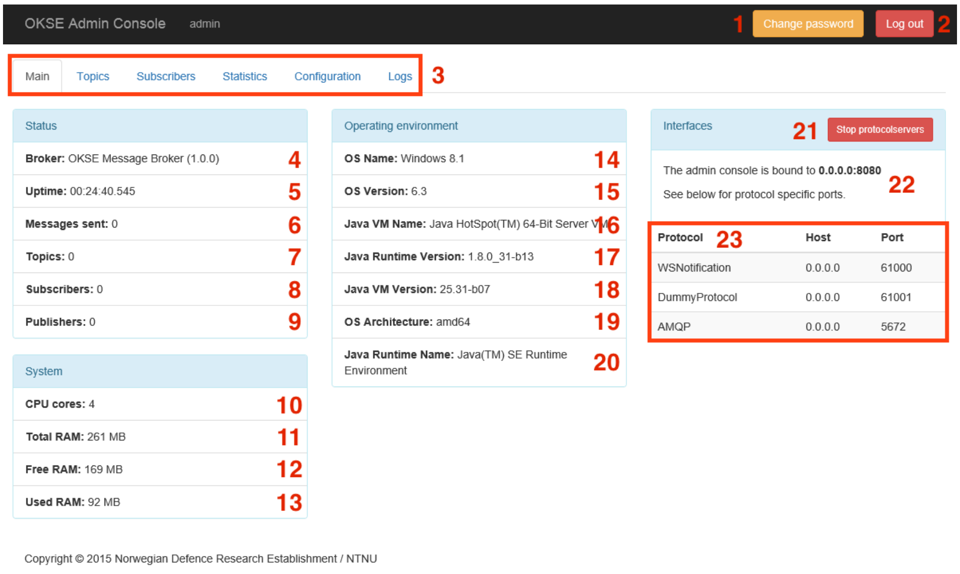
After launching the application, the administrator interface will be reachable from the web browser. The default hostname, port and login information can be found below. If OKSE is hosted on a server, not that the host will be the domain/IP of the server.

- Default host and port for admin console: `localhost:8080`
- Default username and password: `admin` and `password`

C.6 Admin Console

The OKSE Admin Console provides information about OKSE and it is operation environment. It also gives access to configure OKSE. The interface have information sorted in six panes, ”Main”, ”Topics”, ”Subscribers”, ”Statistics”, ”Configuration” and ”Logs”.

C.6.1 The main pane



Status

Broker: OKSE Message Broker (1.0.0)

Uptime: 00:24:40.545

Messages sent: 0

Topics: 0

Subscribers: 0

Publishers: 0

System

CPU cores: 4

Total RAM: 261 MB

Free RAM: 169 MB

Used RAM: 92 MB

Operating environment

OS Name: Windows 8.1

OS Version: 6.3

Java VM Name: Java HotSpot(TM) 64-Bit Server VM

Java Runtime Version: 1.8.0_31-b13

Java VM Version: 25.31-b07

OS Architecture: amd64

Java Runtime Name: Java(TM) SE Runtime Environment

Interfaces

The admin console is bound to **0.0.0.0:8080**

See below for protocol specific ports.

Protocol	Host	Port
WSNotification	0.0.0.0	61000
DummyProtocol	0.0.0.0	61001
AMQP	0.0.0.0	5672

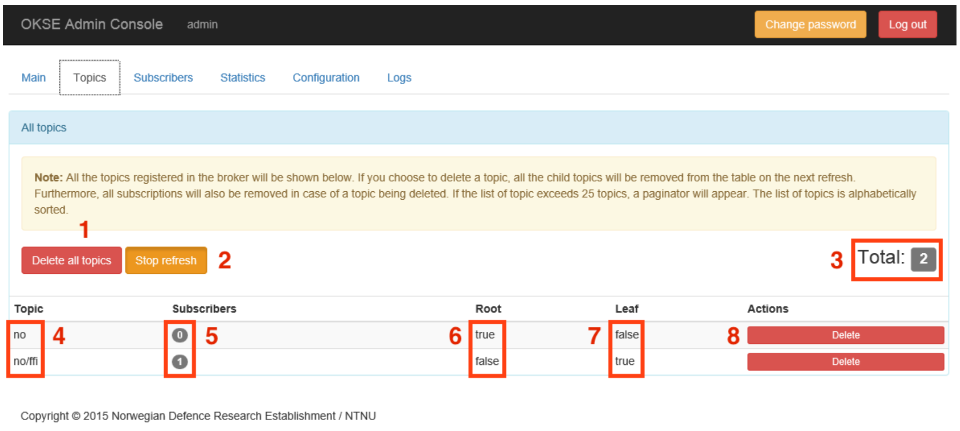
Copyright © 2015 Norwegian Defence Research Establishment / NTNU

Figure C.2: The OKSE Admin Console - Main pane

1. Change the admin password
2. Log out from OKSE
3. Navigation bar with available pages
4. Version and name of broker
5. Time since the broker was started
6. Total number of sent messages
7. Total number of topics
8. Total number of subscribers
9. Total number of publishers
10. Amount of CPU cores in the host system
11. Total amount of RAM allocated to the Java Virtual Machine
12. Total amount of free RAM
13. Total amount of used RAM

14. Name of the operating system
15. Version of the operating system
16. Name of the Java VM
17. Version of the Java VM Runtime
18. Version of the Java VM
19. Architecture of the host system
20. Name of the Java VM Runtime
21. Stop/Start all the protocol servers
22. Information about the address and port for the admin UI
23. Address and port information for each protocol

C.6.2 The topics pane



Note: All the topics registered in the broker will be shown below. If you choose to delete a topic, all the child topics will be removed from the table on the next refresh. Furthermore, all subscriptions will also be removed in case of a topic being deleted. If the list of topic exceeds 25 topics, a paginator will appear. The list of topics is alphabetically sorted.

Total: 2

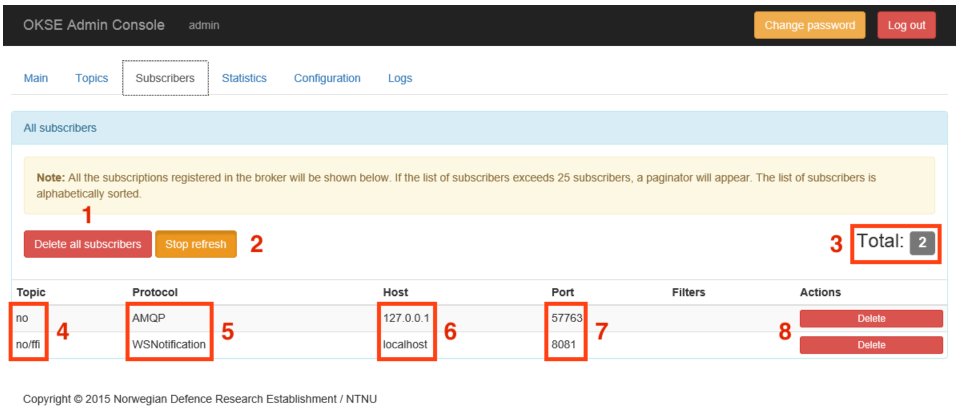
Topic	Subscribers	Root	Leaf	Actions
no	0	true	false	Delete
no/ffi	1	false	true	Delete

Copyright © 2015 Norwegian Defence Research Establishment / NTNU

Figure C.3: The OKSE Admin Console - Topics pane

1. Delete existing topics, cannot be recovered
2. Stop automatic refresh of the webpage
3. Total amount of topics
4. Name of the topic
5. Amount of subscribers
6. Is the topic a root node?
7. Is the topic a leaf node?
8. Delete the topic on this line

C.6.3 The subscribers pane



Note: All the subscriptions registered in the broker will be shown below. If the list of subscribers exceeds 25 subscribers, a paginator will appear. The list of subscribers is alphabetically sorted.

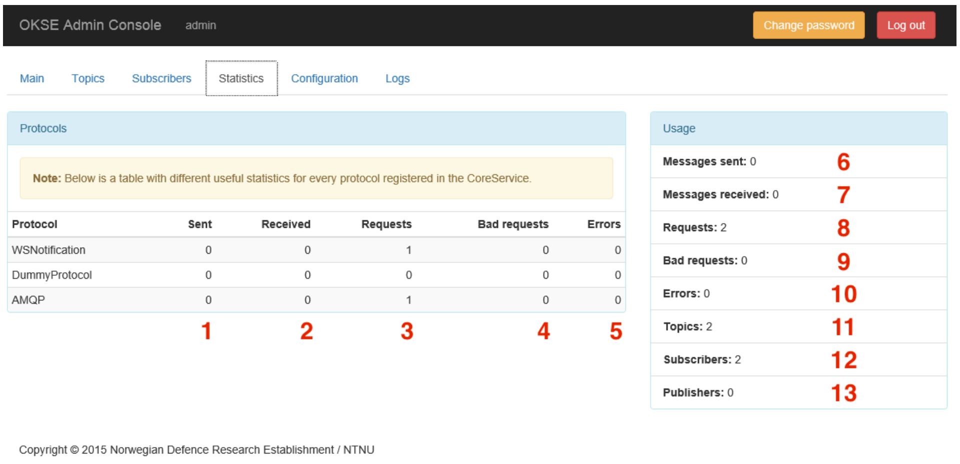
Topic	Protocol	Host	Port	Filters	Actions
no	AMQP	127.0.0.1	57763		Delete
no/ffi	WSNotification	localhost	8081		Delete

Copyright © 2015 Norwegian Defence Research Establishment / NTNU

Figure C.4: The OKSE Admin Console - Subscriptions pane

1. Delete all existing subscribers
2. Stop automatic refresh of the webpage
3. Total amount of subscribers
4. The topic that the subscriber is subscribed to
5. The protocol in use by the subscriber
6. The port in use by the subscriber
7. Delete the subscriber on this line

C.6.4 The Statistics pane



Note: Below is a table with different useful statistics for every protocol registered in the CoreService.

Protocol	Sent	Received	Requests	Bad requests	Errors
WSNotification	0	0	1	0	0
DummyProtocol	0	0	0	0	0
AMQP	0	0	1	0	0

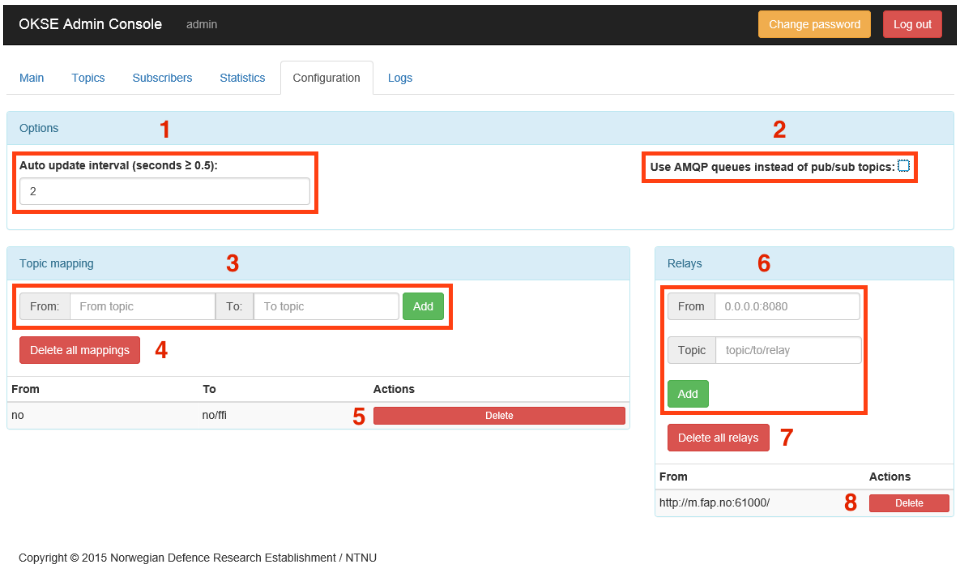
Messages sent: 0	6
Messages received: 0	7
Requests: 2	8
Bad requests: 0	9
Errors: 0	10
Topics: 2	11
Subscribers: 2	12
Publishers: 0	13

Copyright © 2015 Norwegian Defence Research Establishment / NTNU

Figure C.5: The OKSE Admin Console - Statistics pane

1. Amount of messages sent by each protocol
2. Amount of messages received by each protocol
3. Amount of request to each protocol
4. Amount of bad request to each protocol
5. Amount of errors received on each protocol
6. Total amount of messages sent through the system
7. Total amount of messages received by the system
8. Total amount of requests handled by the system
9. Total amount of bad requests received by the system
10. Total amount of errors received by the system
11. Total amount of topics currently handled by the system
12. Total amount of subscribers currently handled by the system
13. Total amount of publishers currently handled by the system

C.6.5 The Configuration pane



Auto update interval (seconds ≥ 0.5): 2

Use AMQP queues instead of pub/sub topics: ☐

From	To	Actions
no	no/ffi	Delete

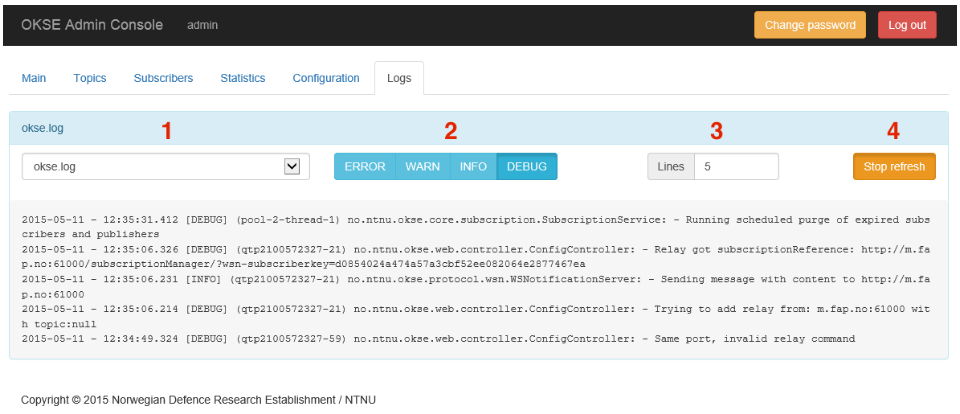
From	Actions
http://m.fap.no:61000/	Delete

Copyright © 2015 Norwegian Defence Research Establishment / NTNU

Figure C.6: The OKSE Admin Console - Configurations pane

1. The admin panel update interval
2. Toggle the use of queue/topic behaviour in AMQP
3. Map one topic to another.
4. Delete all existing topic mappings
5. Delete the topic mapping on this line
6. Set up a relay to another WSN compliant broker
7. Delete all existing relays
8. Delete the relay on this line

C.6.6 The Logs pane



```
2015-05-11 - 12:35:31.412 [DEBUG] (pool-2-thread-1) no.ntnu.okse.core.subscription.SubscriptionService: - Running scheduled purge of expired subs
cribers and publishers
2015-05-11 - 12:35:06.326 [DEBUG] (qtp2100572327-21) no.ntnu.okse.web.controller.ConfigController: - Relay got subscriptionReference: http://m.fa
p.no:61000/subscriptionManager/?wsn-subscriberkey=d0854024a474a57a3cbf52ee082064e2877467ea
2015-05-11 - 12:35:06.231 [INFO] (qtp2100572327-21) no.ntnu.okse.protocol.wsn.WSNotificationServer: - Sending message with content to http://m.fa
p.no:61000
2015-05-11 - 12:35:06.214 [DEBUG] (qtp2100572327-21) no.ntnu.okse.web.controller.ConfigController: - Trying to add relay from: m.fap.no:61000 wit
h topic:null
2015-05-11 - 12:34:49.324 [DEBUG] (qtp2100572327-59) no.ntnu.okse.web.controller.ConfigController: - Same port, invalid relay command
```

Copyright © 2015 Norwegian Defence Research Establishment / NTNU

Figure C.7: The OKSE Admin Console - Logs pane

1. Choose the desired logfile
2. Select the loglevel
3. Enter the amount of lines of log
4. Stop automatic refresh of the webpage

C.7 Configuration files

OKSE has three configurations files. The main configuration file is the okse.properties file. The two other files are log4j.properties, and topicmapping.properties for log- and topic mapping configuration. The following sections describe the default behaviour, defined by each of the files.

C.7.1 okse.properties

This is the main configuration file of the system. Below is a list of the possible settings, and a description.

sprint.application.name
Application name to be displayed in the Admin Console
Default: `OKSE Message Broker`

server.port
Tells Jetty which port the Admin Console should listen to
Default: `8080`

ADMIN_PANEL_HOST
The host the Admin Console should listen to
Default: `0.0.0.0`

CACHE_MESSAGES
Tells OKSE if it should cache messages
Default: `true`

BROADCAST_SYSTEM_MESSAGES_TO_SUBSCRIBERS
Tells OKSE if system messages should be broadcasted to all subscribers
Default: `false`

ENABLE_WSNU_DEBUG_OUTPUT
Tells OKSE if it should log WSNU
Default: `false`

DEFAULT_SUBSCRIPTION_TERMINATION_TIME
Tells OKSE what the subscription termination time should be
Default: `15552000000`

DEFAULT_PUBLISHER_TERMINATION_TIME
Tells OKSE what the publisher termination time should be
Default: `15552000000`

TOPIC_MAPPING
Tells OKSE the path to the topic mapping configuration file
Default: `config/topicmapping.properties`

WSN_HOST
Tells OKSE what host WSNotificanServer should listen to
Default: `0.0.0.0`

WSN_PORT
Tells OKSE what port WSNotificationServer should listen to
Default: `61000`

WSN_CONNECTION_TIMEOUT
Tells OKSE what connection timeout to use with WS-Notification
Default: `5`

WSN_POOL_SIZE
WSNotification http client thread pool used to queue outbound requests
Default: `50`

WSN_MESSAGE_CONTENT_ELEMENT_NAME
Tells OKSE what name non-XML content should be wrapped in
Default: `Content`

WSN_USES_NAT
Tells OKSE if it is hosted behind NAT/Port forwarded network
Default: `false`

WSN_WAN_HOST
Tells OKSE what host it is behind when using NAT
Default: `test.doman.com`

WSN_WAN_PORT
Tells OKSE what port it is behind when using NAT
Default: `61000`

DUMMYPROTOCOL_HOST
Tells OKSE what host DummyProtocolServer should listen to
Default: `0.0.0.0`

DUMMYPROTOCOL_PORT
Tells OKSE what port DummyProtocolServer should listen to
Default: `61001`

AMQP_HOST
Tells OKSE what host AMQPProtocolServer should listen to
Default: `0.0.0.0`

AMQP_PORT
Tells OKSE what port AMQPProtocolServer should listen to
Default: `5672`

AMQP_USE_QUEUE
Tells OKSE if AMQP should use the standard queue or the non-standard topic implementation
Default: `false`

AMQP_USE_SASL
Tells OKSE if AMQP should use SASL
Default: `true`

spring.resources.cache-period
: Tells Spring what cache-period to set on HTTP-requests
Default: `1`

spring.thymeleaf.suffix
: Tells Spring what all Thymeleaf templates are suffixed with
Default: `.html`

spring.thymeleaf.mode
: Tells Spring what type all Thymeleaf templates are
Default: `HTML5`

spring.thymeleaf.encoding
: Tells Spring what encoding to use on all Thymeleaf templates
Default: `UTF-8`

spring.thymeleaf.content-type
: Tells Spring what content-type all Thymeleaf templates should be returned with
Default: `text/html`

C.7.2 log4j.properties

This is the configuration file for all the log files available for the system. See below for a detailed list of all possible settings, and their description.

log
: Default folder location for log files, relative to .jar file
Default: `logs`

pattern
: Default pattern to print log output in
Default: `%d{yyyy-MM-dd - HH:mm:ss.SSS} [%p] (%t) %c: - %m%n`

maxLogFileSize
: Max log file size, before log rotate
Default: `5MB`

numberOfBackups
: Max number of log files, before it purges old files
Default: `10`

log4j.logger.no.ntnu.okse
: Default log level for okse log messages
Default: `DEBUG, OKSE`

log4j.logger.org.apache.qpid
: Default log level for qpid log messages
Default: `DEBUG, QPID`

log4j.logger.org.eclipse.jetty
: Default log level for Jetty log messages
Default: `INFO, JETTY`

log4j.logger.org.springframework
Default log level for Spring log messages
Default: `INFO, SPRING`

log4j.appender.OKSE
Default appender to use for OKSE logs
Default: `org.apache.log4j.RollingFileAppender`

log4j.appender.OKSE.File
Default log file to use for OKSE logs
Default: `${log}/okse.log`

log4j.appender.OKSE.MaxFileSize
Default log file size to use for OKSE logs
Default: `${maxLogFileSize}`

log4j.appender.OKSE.MaxBackupIndex
Number of backups for OKSE logs
Default: `${numberOfBackups}`

log4j.appender.OKSE.layout
Pattern engine for OKSE logs
Default: `org.apache.log4j.PatternLayout`

log4j.appender.OKSE.layout.conversionPattern
Default pattern to use for OKSE logs
Default: `${Pattern}`

log4j.appender.SPRING
Default appender to use for SPRING logs
Default: `org.apache.log4j.RollingFileAppender`

log4j.appender.SPRING.File
Default log file to use for SPRING logs
Default: `${log}/spring.log`

log4j.appender.SPRING.MaxFileSize
Default log file size to use for SPRING logs
Default: `${maxLogFileSize}`

log4j.appender.SPRING.MaxBackupIndex
Number of backups for SPRING logs
Default: `${numberOfBackups}`

log4j.appender.SPRING.layout
Pattern engine for SPRING logs
Default: `org.apache.log4j.PatternLayout`

log4j.appender.SPRING.layout.conversionPattern
Default pattern to use for SPRING logs
Default: `${Pattern}`

log4j.appender.JETTY
Default appender to use for JETTY logs
Default: `org.apache.log4j.RollingFileAppender`

log4j.appender.JETTY.File
Default log file to use for JETTY logs
Default: `${log}/okse.log`

log4j.appender.JETTY.MaxFileSize
Default log file size to use for JETTY logs
Default: `${maxLogFileSize}`

log4j.appender.JETTY.MaxBackupIndex
Number of backups for JETTY logs
Default: `${numberOfBackups}`

log4j.appender.JETTY.layout
Pattern engine for JETTY logs
Default: `org.apache.log4j.PatternLayout`

log4j.appender.JETTY.layout.conversionPattern
Default pattern to use for JETTY logs
Default: `${Pattern}`

log4j.appender.QPID
Default appender to use for QPID logs
Default: `org.apache.log4j.RollingFileAppender`

log4j.appender.QPID.File
Default log file to use for QPID logs
Default: `${log}/qpid.log`

log4j.appender.QPID.MaxFileSize
Default log file size to use for QPID logs
Default: `${maxLogFileSize}`

log4j.appender.QPID.MaxBackupIndex
Number of backups for QPID logs
Default: `${numberOfBackups}`

log4j.appender.QPID.layout
Pattern engine for QPID logs
Default: `org.apache.log4j.PatternLayout`

log4j.appender.QPID.layout.conversionPattern
Default pattern to use for QPID logs
Default: `${Pattern}`

log4j.appender.stdout
Default appender to use for console output
Default: `org.apache.log4j.ConsoleAppender`

log4j.appender.stdout.Target
Default target to use for console output
Default: `System.out`

log4j.appender.stdout.layout
Pattern engine for console logs
Default: `org.apache.log4j.PatternLayout`

log4j.appender.stdout.layout.conversionPattern
Default pattern to use for console logs
Default: `${Pattern}`

C.7.3 topicmapping.properties

This is a configuration file for adding predefined topic mappings upon system initialization. To add predefined topic mappings, use the format from the example below. The example shows a simplex mapping and a duplex mapping. In the simplex mapping the first topic, no/ffi, is forwarded to nato/hq/info, but not the other way. In the duplex mapping, all messages received on each topic will be forwarded to the other.

```
# Simplex mapping
no/ffi=nato/hq/info

# Duplex mapping
no/test=com/test
com/test=no/test
```

C.8 Test clients

As none of the protocols had any useful clients for testing, a JAR file is included for AMQP send and one for AMQP receive. Keep in mind that little time has been spent on these programs, and that they are only suitable for minor testing. The files can only be run from the command line with the arguments given below. Other tools where used in the testing of WSN, but the tools are owned by FFI and cannot be distributed any further, more information about this in appendix F. Note that these test clients do not reflect proper usage of the software, but are meant for test purposes.

C.8.1 AMQP send

Provided in the project delivery is a folder named test-clients, within this folder, the file amqpsend.jar is located. The file takes three arguments.

-a amqp://localhost:5672/topic
Address argument, needed to tell what server to send the message to(AMQP is hosted at port 5672 by default).
Default: `amqp://localhost:5672/example`

-s ”subject”
Subject argument, not needed.
Default: `example`

message
The last argument will be sent as the message
Default: `example`

Below an example of a valid command is shown, the message in the example is sent to the localhost. The results should be reflected in the administration console under the ”Statistics” tab.

```
java -jar amqpsend.jar -a amqp://localhost/test -s "test" test
```

C.8.2 AMQP receive

Provided in the project delivery there is a folder named test-clients, within this folder, the file amqprecv.jar is located. The file takes one argument.

-a amqp://localhost:5672/topic
Address argument, tells the receiver where to subscribe
Default: `amqp://localhost:5672/example`

Below an example of a valid command is shown, the receiver will subscribe to localhost:

```
java -jar amqprecv.jar -a amqp://localhost/test
```

C.8.3 WSN Test script

A script is included that can be used to send different messages over WSN, but *cannot* receive messages. The script is called bullrider.py and is located in the test-clients folder. The script is written in Python 2.7 and uses a third party plugin called Requests[2]. This script should run on any platform as long as both Python and Requests are available. Below, an installation guide for Windows, Linux and OS X is included.

Windows

To install Python and Requests on a Windows machine, the following steps are required:

1. Download and install the latest version of Python 2.7 from `https://www.python.org/downloads/`.
2. Open cmd.exe,
3. Change directory to the location of the python installation.
4. Install Requests by running the command:

   ```
   python -m pip install requests
   ```

After the installation, change directory of the cmd session to the location of bullrider.py. To run python in the shell in any directory, the `Python2.7` directory should be added to the path. This can be done for the current session by entering the command:

```
path %path%;C:\Python27
```

Note that this will have to be done for every new cmd.exe window you open. Now that Python and Requests are installed, move on to the Usage section.

[2] `http://docs.python-requests.org/en/latest/`

Linux/OS X

Mac OS X and most Linux distributions comes with Python 2.7 installed. To verify that you have Python 2.7 installed, open a terminal and write ”python”. You should see something like:

```
$ python
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
[GCC 4.9.2] on linux2
```

If ”command not found” appears, check the package manager of the distribution for instructions on installing Python 2.7, or go to python.org and download the installer from there. When Python 2.7 is installed, Requests needs to be installed. In the terminal, install the software by typing the following command.

```
pip install requests
```

If pip is not available, use easy_install in the following way:

```
easy_install requests
```

Note that sudo access might be required to run pip or easy_install. In that case, run

```
sudo pip install requests
or
sudo easy_install requests
```

.
After the installation, change directory in the terminal to the location of bullrider.py. Now that Python and requests are installed, move on to the Usage section.

Usage

To use the bullrider.py script, open a terminal/cmd.exe window and change the directory to where bullrider.py is located. If Windows is being used, remember to repeat the path step if Python is not permanently added to your path.

Below is a description on what the different arguments do, and the available options that can be used. Note that the arguments need to be given in the listed order. The format of the arguments is as follows:

```
python bullrider.py [request type] [hostname/ip] [port] [topic]
```

request type
Type of request to send, available: all, notify, massnotify, multinotify, subscribe, unsubscribe

hostname/ip
Hostname or IP address of the OKSE instance

port
: Port of the OKSE instance

topic
: The topic to send the given request to.

Below an example of a valid command is shown, this will send a notify to a OKSE hosted on localhost at port 61000:

```
python bullrider.py notify localhost 61000 test
```

Click enter as a response to the message asking which IP:PORT should be used.

C.8.4 Example test case

An example test case for the application, can be performed by running the AMQP receiver and the bullrider.py script to send a WSN message. By doing this, the tester can verify that the message is converted and forwarded correctly.

Steps

1. First, setup an OKSE instance as explained in section C.3.
2. Launch AMQP receive (see section C.8.2) and subscribe on topic ”test”.
3. Launch bullrider.py (see section C.8.3) and send a message with ”notify” to the ”test” topic.
4. Check the AMQP receive instance.

Appendix D

Developer Manual

D.1 Introduction

This manual provides an overview of the initial steps needed to extend and further develop OKSE. The first sections are concerned with the set up of the development environment. The rest of explains the different parts of the system and how they may be extended. The system is open source, and licenced under the MIT licence[1].

D.2 Requirements and Setup

The recommended setup, as noted below, is using IntelliJ IDEA[2] as the Java IDE. However, other IDEs that meet the requirements listed below, or a combination should suffice.

D.2.1 Java

The minimum required major version of Java is 8, and it is recommend to use the latest minor release. Currently, Java 8 version 31 is the version used to compile the packaged version of OKSE.

D.2.2 Apache Maven

The project source code, tests and dependency management is structured with Apache Maven[3]. Maven is used to fetch dependencies, run tests and build the final jar files. The minimum major version of Maven is 3, and any minor version from 3.0.5 and up can be used. The version used for the current version of the system is 3.3.3.

It is worth mentioning that many of the popular Java IDE's has Maven support. Thus, its not required to install it separately.

[1] `http://opensource.org/licenses/MIT`
[2] `https://www.jetbrains.com/idea/`
[3] `https://maven.apache.org`

D.2.3 Recommended IDE

The development team strongly recommends IntelliJ IDEA Ultimate[4] as the Integrated Development Environment. Note that the free Community edition should suffice, although the Ultimate edition has better support for frameworks like Spring and TestNG (test framework of choice), which is used for the web administration interface. Both Community and Ultimate edition comes bundled with Maven.

IntelliJ IDEA setup

As the user stands free to choose the IDE, the setup documentation will be fairly basic and based on IntelliJ IDEA.

Prerequisites: IntelliJ IDEA is installed, an Internet connection and the source code is stored locally.

1. Open IntelliJ IDEA
2. Click ”Import Project” or File → ”Import Project”
3. Browse the file system and select the source code folder
4. Choose ”Import project from external model”
5. Choose ”Maven”
6. Check ”Import Maven projects automatically”
7. Click Next
8. no.ntnu.okse... should be selected, click Next
9. Select 1.8 (if it is not available, click ”+” and add it)
10. Click Next
11. Click Finish

After the IntelliJ IDEA window is opened with the project, all the needed dependencies will be downloaded. After this is complete, the project setup is complete.

D.2.4 Test Framework: TestNG

The system has a lot of unit tests, written with the test framework TestNG[5]. Any new features and expansions of the system should be accompanied by unit tests. Please refer to the TestNG documentation[6] for information and documentation.

[4]`https://www.jetbrains.com/idea/`
[5]`http://testng.org/doc/index.html`
[6]`http://testng.org/doc/documentation-main.html`

D.3 Common Patterns Used

D.3.1 Service oriented architecture

Although there exists no formal definition or agreement upon what constitutes a service oriented architecture, some main principles are common. The OKSE message brokering system is implemented with some of these principles in mind. They include, but are not limited to:

- Service loose coupling: Minimize dependencies
- Service abstraction: Hide logic from the outside world
- Service re-usability: The DRY (Don't repeat yourself) principle
- Service autonomy: Services have complete control over their own logic
- Service encapsulation: Services incorporate other sub-components that were not service oriented, but serve as a part of the larger entity.

D.3.2 Module pattern

Software module pattern is a design pattern that emphasizes on organizing software components into modules. Some languages have this built into the language, but similar results can be accomplished through patterns. One way of creating modules is through the singleton pattern, described in the next section. OKSE implements several ”tools” or ”library” classes, which are considered software modules through their static references, and their manipulation of data from other sources.

D.3.3 Singleton pattern

The singleton pattern is a specialization of the module pattern. It is used to guarantee that an object cannot be instantiated more than once. Any further invocation would return the already instantiated object. This secures a single state for that particular component or module. In the case of the OKSE message brokering system, all services and protocol servers adhere to this pattern. This ensures that all references to a service point to the same instance.

This also makes it easier to reference the different services, as they are all referenced statically from their class. This means that it is not necessary to keep a local reference to the service in another class, or pass it as an argument to a certain method.

A practical example of how the singleton pattern is implemented in the OKSE core services and protocol servers, see sections D.5 and D.6

D.3.4 Reactor pattern

In the OKSE system, all services and protocol servers are run in their own personal thread. To ensure a thread safe execution environment, inspiration was gathered from the reactor pattern. The reactor pattern generally receives events concurrently, and demultiplexes them synchronously, before dispatching to event handlers.

Although the OKSE system does not strictly implement separate dispatchers and event handlers, the basic idea stays the same. All singletons block on an internal task queue, awaiting new tasks (events) to be performed. The exact structure of these events and how they are consumed vary from service to service.

This approach allows different threads running client requests to concurrently perform tasks on the services without having to worry about concurrent modification errors.

D.3.5 Observer pattern

Due to the asynchronous nature of the services, listener support is implemented in most of them. This allows the service in question to invoke callback methods synchronously after a task has been completed. The more general use of the observer pattern is that all changes in internal state at the core service level, will also update registered listener components. This allows components such as the WSNSubscriptionManager to automatically synchronize its internal state when a change occurs in the OKSE SubscriptionService.

D.4 Components

This section provides an overview of which steps are needed to extend the system with new services and protocols. For general system reference and the interaction of existing components, see chapter 5.2.

D.4.1 Application

The `Application.java` file that resides in the root of the project is the main invocation point when starting the OKSE application. The operations performed in this class are initializing the web admin console and the CoreService instance, which in turn boots up more services.

The most important thing to note regarding this main class, is that it is also the class in which further extensions should be registered. Extending the application with additional core services is done through registering them to the CoreService via the `registerCoreService()` method.

Extending the application with additional protocol support is done through use of the `addProtocolServers()` method of the CoreService instance.

The `Application` class also exposes a static method for reading in configuration files. This method will also verify that the directory structure and necessary files exist. The `readConfigurationFiles()` method does all this, and returns a `Properties` object, which contains all the key-value pairs of the OKSE configuration file.

D.4.2 Administration interface

The OKSE administration console is built on the Spring framework, more exactly the spring-boot package[7] with Thymeleaf-templates[8]. The main aspects of the Web Admin component are its models, controllers, templates and front-end JavaScript. The administration interface is built in a modular way, to achieve loose coupling of modules. Each pane of the interface has its own controller and respective front-end JavaScript. With this approach it is easy to extend the web interface.

Back-end

The back-end of OKSE works as a RESTful Web Service. As the back-end uses the Spring framework, the main architectural approach for the back-end is based on Spring's approach for building RESTful web services. All HTTP-requests are handled by a dedicated controller. These controllers are easily identified by the `@RestController` annotation. Each controller is responsible for handling all HTTP-requests on given routes defined by `@RequestMapping` annotations.

Front-end

The front-end of OKSE is the only component of the system that is directly accessible for graphical user interaction. It is responsible for making HTTP-requests to the back-end for manipulation and presentation of the state OKSE is currently in. See chapter 5.3.2 for a more detailed description of the front-end architecture.

D.4.3 Interacting with the MessageService

The MessageService is responsible for distributing messages to all the registered protocol servers. It provides a simple interface to put messages up for processing. The following code snippet shows how messages are created and distributed.

```
// Define a topic
String topic = "no/ffi";
// Create som content
String msg = "some data";
// Fetch a related publisher, null if none is relevant
Publisher p = null;
// Set the originating protocol, must be the same as
// the protocolServerType field in the ProtocolServer class.
String originProtocol = "WSNotification";

// Create a Message object
Message m = new Message(msg, topic, p, originProtocol);
// Distribute the message
MessageService.getInstance().distributeMessage(m);
```

Listing D.1: Distributing a message

[7]`https://spring.io/guides/gs/rest-service/`
[8]`http://www.thymeleaf.org`

The MessageService also allows requestors to retrieve the last message sent on a topic, regardless of originating protocol. This is done in the following fashion.

```java
// Initialize the message variable
Message m;
// Fetch the latest message
m = MessageService.getInstance().getLatestMessage("topic/subtopic");

// Check if there was any message
if (m != null) {
    /* Do something with the message */
}
```

Listing D.2: Retrieving latest message on a topic

Another handy helper method is the ability to take a single message, and duplicate it to all the topics in a provided set of topics. Most of the time, strings are used to represent topics, for ease of use. Still, the actual topics are stored in Topic objects. This helper method uses Topic objects.

```java
// Create a message
Message m = new Message("<data>1</data>", "hq/test", null, "
WSNotification");

// For example, retrieve all Topics that are root nodes
HashSet<Topic> topicSet = TopicService.getAllRootTopics();
// Fetch an additional topic node
Topic extraTopic = TopicService.getTopic("no/ffi");
// Add it to the set
topicSet.add(extraTopic);

// Generate a list of messages destined to each of the
// provided topics
List<Message> generated = MessageService.getInstance()
    .generateMessageForAGivenTopicSet(m, topicSet);

/* Do some action on the messages, for example... */
generated.forEach(m -> MessageService.getInstance().distribute(m));
```

Listing D.3: Duplicating message to multiple topics

D.4.4 Interacting with the TopicService

The TopicService is responsible for maintaining an overview of all topics that have been referenced at some time. It exposes a general API for adding and removing topics, as well as several methods for retrieving topics.

A Topic in itself is a part of a tree of nodes. It may have a parent, and it may have children. The TopicService keeps track of all these trees. For the implemented protocols in the OKSE system, none use this hierarchical nature. Still, they are represented in this way for future use.

In order to retrieve the canonical name of a topic, one uses the `getFullTopicString()` method on a Topic object. This representation is what is expected in all parts of the OKSE system that accepts topics represented by strings.

The following code shows how to add a new topic to the system.

```java
// Add a new topic
TopicService.getInstance().addTopic("no");

// Add another topic
TopicService.getInstance().addTopic("no/ffi/uav");
```

Listing D.4: Adding a new topic

In the second `addTopic` call, the TopicService will identify that the `no` topic already exists. It will only generate the new topic nodes that are needed, link these together, and connect the branch to the already existing `no` topic node. After the second `addTopic` call, the `no` topic node is reachable by traversing the tree from `uav` by using the `getParent()` method of a topic node.

The following code snipped demonstrates how Topics are removed from the system.

```java
// Add some topics
TopicService.getInstance().addTopic("no/ffi/uav");

// Delete the intermediate node
TopicService.getInstance().deleteTopic("no/ffi");
```

Listing D.5: Deleting a topic

In this example, three topic nodes are created and linked together in a tree. When the `no/ffi` node is deleted, its child `uav` is also deleted. After the operation, only the `no` topic node remains in the system.

The following examples show a selection of the service methods exposed by the TopicService.

```java
// Retrieve all leaf topic nodes
HashSet<Topic> leafNodes = TopicService.getInstance().getAllLeafTopics
();

// Retrieve all root topic nodes
HashSet<Topic> rootNodes = TopicService.getInstance().getAllRootTopics
();

// Get all topic nodes
HashSet<Topic> allNodes = TopicService.getInstance().getAllTopics();

// Check existance
if (TopicService.getInstance().topicExists("no/ffi")) {
    /* Perform some action */
}
```

Listing D.6: Additional TopicService methods

It is worth mentioning that all the returned sets of topics are shallow clones of the originating data structure. This is to ensure that no accidental modification of the internal data structure is made.

D.4.5 Interacting with the SubscriptionService

The SubscriptionService is responsible for maintaining an overview of the currently registered subscribers and publishers. The SubscriptionService also periodically removes entities that have expired. Finally, the SubscriptionService is responsible for modifying the parameters connected to a specific entity.

The Subscriber object itself contains a selection of sets and attributes to service possible future needs. The following code sample illustrates some of these features. Publisher objects contain a subset of these features.

```
// Host of the subscriber
String host = "123.123.123.123";
// Port of the subscriber
Integer port = "52612";
// The topic of the subscriber
String topic = "no/ffi";
// The originating protocol of the subscription
String origProtocol = "AMQP";

// Create the subscriber
Subscriber sub = new Subscriber(host, port, topic, origProtocol);

// Set an expiration time for the subscription
// in milliseconds since UNIX epoch.
sub.setTimeout(1234567890);

// Set an attribute on the subscriber object
// for some specific use
sub.setAttribute("key", "value");

// Get an attribute from the subscriber object
String value = sub.getAttribute("key");
if (value != null) {
    /* Perform some action */
}

// Store some sort of filter expression used by
// the subscriber
sub.addFilter("regex:^[a-zA-Z]+");
sub.addFilter("//rootElement/*");

/* These filters do not have any purpose but to
store the information for later retrieval, and
list them in the administration interface */

// Retrieve some general attributes from the subscriber
sub.getHost();
sub.getPort();
sub.getTimeout();
sub.getTopic();
sub.getOriginProtocol();
if (sub.shouldExpire()) { /* ... */ }
if (sub.hasExpired()) { /* ... */ }
```

Listing D.7: Properties of the Subscriber object

A subset of these fields and methods are also available for Publisher objects.

The following code sample illustrates how to add, update, and remove subscribers from the SubscriptionService.

```java
// Create a subscriber
Subscriber sub = new Subscriber("10.0.0.1", "8080", "no/ffi", "AMQP");

// Add it to the SubscriptionService
SubscriptionService.getInstance().addSubscriber(sub);

// Pause the subscription
SubscriptionService.getInstance().pauseSubscriber(sub);

if (sub.getAttribute("paused").equals("true") {
    /* true, subscriber is paused */
} else {
    /* subscriber is not paused */
}

// Resume the subscription
SubscriptionService.getInstance().resumeSubscriber(sub);

// Renew the subscription
Long newTerminationTime = 1234567890;
SubscriptionService.getInstance().renewSubscriber(sub,
newTerminationTime);

// Remove it again
SubscriptionService.getInstance().removeSubscriber(sub);
```

Listing D.8: Subscriber actions

It is important to note that these methods only alter the internal state of the OKSE SubscriptionService and its Subscriber objects. It is up to the specific protocol implementations that have need of these features to use them where applicable.

The SubscriptionService also exposes a set of extra service methods that provide aggregation and selection capabilities. Some examples are listed in the code sample below.

```java
// Cache the SubscriptionService reference for ease of access
SubscriptionService ss = SubscriptionService.getInstance();

// Get a subscriber by its unique ID
Subscriber sub = ss.getSubscriberByID("a517...1c3f");

// Get all subscribers or publishers for a topic
HashSet<Subscriber> subscribers;
HashSet<Publisher> publishers;
subscribers = ss.getAllSubscribersForTopic("no/ffi");
publishers = ss.getAllPublishersForTopic("no/ffi");

// Get all subscribers and publishers
HashSet<Subscriber> allSubscribers;
HashSet<Publisher> allPublishers;
allSubscribers = ss.getAllSubscribers();
allPublishers = ss.getAllPublishers();
```

Listing D.9: Additional SubscriptionService methods

As with most collections returned by the services, sets are shallow copies of the internal structure in the service itself. This is done to prevent accidental modification of the internal structure.

When Subscribers or Publishers are created, modified or deleted, all registered listeners to the SubscriptionService are notified. They will receive either a SubscriptionChangeEvent or a PublisherChangeEvent. The following code sample illustrates how these events might be used.

```java
public class SomeListener implements SubscriptionChangeListener {
    public SomeListener() {
        SubscriptionService.getInstance().addSubscriptionChangeListener(
    this);
    }

    @Override
    public void subscriptionChanged(SubscriptionChangeEvent e) {
        SubscriptionChangeEvent.Type type = e.getType();
        Subscriber sub = e.getData();

        // If we for example are only interested in AMQP
        // related subscribers
        if (!sub.getOriginProtocol().equals("AMQP")) {
            return;
        }

        if (type == SubscriptionChangeEvent.Type.UNSUBSCRIBE) {
            /* Do some action for unsubscribe events */
            someLocalMapping.remove(sub);
        }
        /* else if (type == ...)
            Additional types are:

            SUBSCRIBE,
            PAUSE,
            RESUME,
            RENEW
        */
    }
}
```

Listing D.10: Use of the SubscriptionService listener support

D.5 Adding new protocols

The OKSE system was designed such that extending the system with additional protocols should be as easy as possible. Some boilerplate code is still needed to have a minimum foundation on which the protocol server is intended to run.

To add a new protocol to the OKSE system, the first step is to extend the `AbstractProtocolServer` class. This will prompt the developer to implement a set of methods needed in a protocol server. Additionally, some extra static methods are needed for implementing it as a singleton.

Below is an example of the structure of a class extending from the `AbstractProtocolServer` class.

```java
public class NewProtocolServer extends AbstractProtocolServer {

    // Generic statics
    private static Logger log;
    private static NewProtocolServer _singleton;
    private static Thread _serverThread;
    private static boolean _invoked;

    // Internal defaults
    private static final String DEFAULT_HOST = "0.0.0.0";
    private static final Integer DEFAULT_PORT = 1337;

    // Custom fields

    // Private constructor
    private NewProtocolServer(String host, Integer port) {
        init(host, port);
    }

    // Private init method
    private init(String host, Integer port) {
        // Host and port
        if (host == null) this.host = DEFAULT_HOST;
        else this.host = host;
        if (port == null) this.port = DEFAULT_PORT;
        else this.port = port;

        // Init the logger
        log = Logger.getLogger(NewProtocolServer.class.getName());

        // Protocol server type
        protocolServerType = "NewProtocol";

        // Update invoke flag
        _invoked = true;
    }

    // Public singleton object factory
    public static NewProtocolServer getInstance() {
        if (!_invoked) {
            String configHost = null;
            Integer configPort = null;

            /* Read from config file if exists... */

            _singleton = new NewProtocolServer(configHost, configPort);
        }

        return _singleton;
    }

    // Public boot method
    public void boot() {
        if (!_running) {
            _running = true;

            // Instantiate necessary objects
            // Start necessary support modules etc

            // Set up server thread
            _serverThread = new Thread(() -> this.run());
            _serverThread.setName(protocolServerType);
            _serverThread.start();
        }
    }

    // Public run method
    public void run() {
        while (_running) {
            // Main run loop that does what it is supposed to do.
            // Blocking on a queue, or awaiting connections etc
        }
    }

    // Stop method
    public void stopServer() {
        _running = false;
        _invoked = false;
        /*
            Stop server thread in here
                                                */
        _serverThread = null;
        _singleton = null;
    }

    // Get protocol type
    public String getProtocolServerType() {
        return this.protocolServerType;
    }

    // Send message method that is called from MessageService
    public void sendMessage(Message msg) {
        // Check that the message came from another protocol
        if (!msg.getOriginProtocol.equals(protocolServerType) {
            // Perform needed actions to send msg here
        }
    }
}
```

Listing D.11: Adding a new protocol

The only thing needed to include the new protocol in the OKSE system is to add it in the `Application` class, through the `CoreService`.

```java
            // Initialize main system components
            webserver = new Server();
            cs = CoreService.getInstance();

            /* ... */

            /* REGISTER PROTOCOL SERVERS HERE */
            cs.addProtocolServer(WSNotificationServer.getInstance());
```

```
        cs.addProtocolServer(AMQProtocolServer.getInstance());
        cs.addProtocolServer(NewProtocolServer.getInstance());
```

Listing D.12: Registering a new protocol

D.6 Adding new core services

The OKSE system was designed such that extending the system with additional core services should be as easy as possible. Some boilerplate code is still needed to have a minimum foundation on which the core service is intended to run.

To add a new core service to the OKSE system, the first step is to extend the `AbstractCoreService` class. This will prompt the developer to implement a set of methods needed in a core service. Additionally, some extra static methods are needed for implementing it as a singleton.

Below is an example implementation of a new core service, which extends the `AbstractCoreService` class. No clear purpose is defined for this service other than for the intent of clarifying the structure.

```
public class NewService extends AbstractCoreService {

    // Generic statics
    private static NewService _singleton;
    private static Thread _serviceThread;
    private static boolean _invoked;

    // Event queue
    private LinkedBlockingQueue<SomeEvent> queue;

    // Custom fields
    Properties config;

    // Protected constructor (for unit test subclassing purposes)
    protected NewService() {
        // Pass the classname to super to init the logger
        super(NewService.class.getName());
        if (_invoked) throw new IllegalStateException("Already invoked");
        init();
    }

    // Initialization helper method
    private init() {
        queue = new LinkedBlockingQueue<>();
        config = Application.readConfigurationFiles();
        _invoked = true;
    }

    // Static object factory for singleton
    public static NewService getInstance() {
        if (!_invoked) _singleton = new NewService();
        return _singleton;
    }

    // Public boot method
    public void boot() {
        if (!_running) {
            log.info("Booting NewService");
            _serviceThread = new Thread(() -> this.run());
            _serviceThread.setName("NewService");
            _serviceThread.start();
        }
    }

    // Public run method
    public void run() {
        _running = true;
        while (_running) {
            try {
                Event event = queue.take();
                /* Handle event here... */

            } catch (InterruptedException e) {
                log.error("Interrupted during event queue fetch");
            }
        }

        log.info("NewService stopped");
    }

    // Stop method
    public void stop() {
        _running = false;
        /* Stop serviceThread here */

        /* An example is to send a no-op event to the queue */
        _serverThread = null;
        _singleton = null;
        _invoked = false;
    }

    // Register listener support
    public void registerListenerSupport() {
        /*  Add all commands to register this new service
            as a listener to other services here
            as it will be called after all services
            are booted.
        */
    }
}
```

Listing D.13: Creating a new core service

The only thing needed to include the service in the OKSE system is to register it in the `Application` class, through the `CoreService`.

```
        // Initialize main system components
        webserver = new Server();
        cs = CoreService.getInstance();

        /* REGISTER CORE SERVICES HERE */
        cs.registerService(TopicService.getInstance());
        cs.registerService(MessageService.getInstance());
        cs.registerService(SubscriptionService.getInstance());
        cs.registerService(NewService.getInstance());

        /* ... */
```

Listing D.14: Registering a new service

D.7 Extending the administration interface

This section describes how the administration interface might be extended with new content and functionality. Figure D.1 visualizes how the system handles events and exposes information to the users. The key concepts to notice here are the AJAX-requests and the controllers. On an event, whether it is a page auto update or a button-click, the request URL is accessed by the AJAX-request. Springs dispatcher maps the URL and forwards it to the correct controller. This controller interacts with OKSEs services and returns a response.

- **Dispatcher** - This is the class that is exposed to the user. It forwards request URLs to the mapping handler to reach the correct controller. It then calls the controller responsible for the request. Finally it returns the response to the user.
- **Mapping handler** - This class maps the request URL to the controller responsible for the request, and returns it to the dispatcher.
- **Controller** - This is the class responsible for executing tasks and getting information from the message broker. It returns a JSON response [9] string to the dispatcher.

[9] `http://json.org`

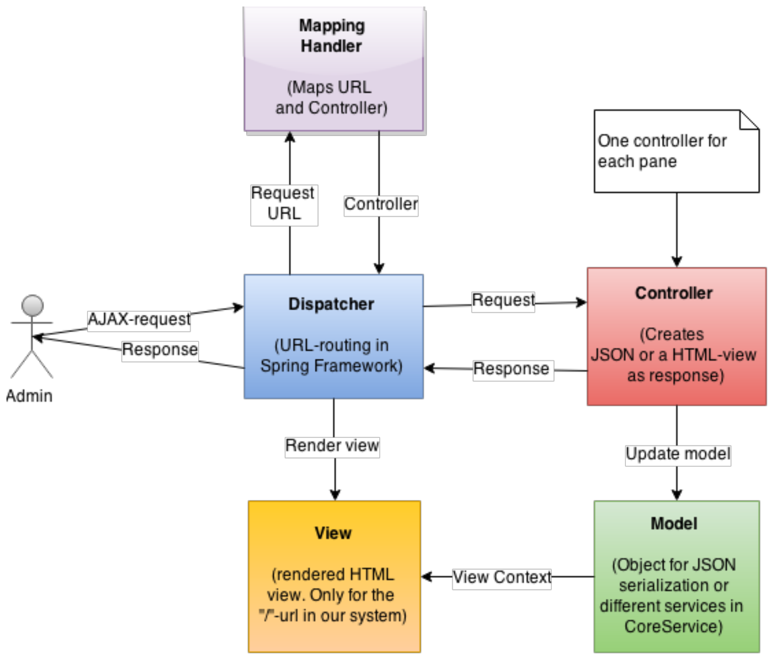

Mapping Handler
(Maps URL and Controller)
Request URL
Controller
One controller for each pane
AJAX-request
Response
Admin
Dispatcher
(URL-routing in Spring Framework)
Request
Response
Controller
(Creates JSON or a HTML-view as response)
Render view
Update model
View
(rendered HTML view. Only for the "/"-url in our system)
View Context
Model
(Object for JSON serialization or different services in CoreService)

Figure D.1: REST-API structure

The following steps explains how to actually extend the application. In addition, the following steps provides a guide to Thymeleaf templates, Spring, JavaScript and jQuery.

- Thymeleaf - see `http://www.thymeleaf.org/documentation.html`
- Spring framework - see `https://spring.io/guides/gs/rest-service/`
- jQuery - see `http://jqfundamentals.com`
- JavaScript - see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript`

D.7.1 Administration interface structure

All files for the administration interface are located in three main folders, `static/js`, `templates` and `web`. See figure D.2 for a detailed overview. To extend the application, the existing files can be copied or new ones created based on the skeleton provided in section D.7.1. The files that needs to be copied or created are:

- HTML template files in `src/resources/templates/`
- JavaScript files in `src/resources/static/js/`
- Spring controllers in `src/no/ntnu/okse/web/controller/`

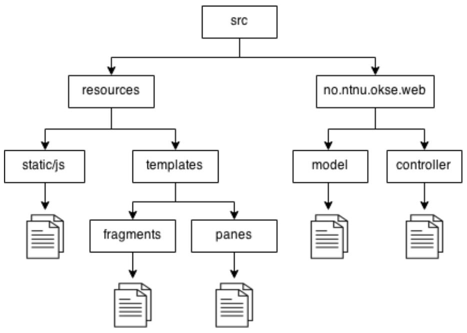


Figure D.2: Folder hierarchy for the administration interface

OKSE contains of seven JavaScript modules:

main.js
: Responsible for the ”Main”-pane. This module also contains all logic for setting the update interval all panes. Additionally, this is the location of the AJAX-module.

topics.js
: Responsible for the ”Topics”-pane. This module contains all logic regarding topics, and includes refreshing the table containing all topics.

subscribers.js
: Responsible for the ”Subscribers”-pane. This module contains all logic regarding subscribers. which includes, but are not limited to, refreshing the table containing all subscribers.

stats.js
: Responsible for the ”Statistics”-pane. This module contains all logic for refreshing the statistics from OKSE.

logs.js
: Responsible for the ”Logs”-pane. This module requests the given log from OKSE and visualizes it.

config.js
: Responsible for the ”Configuration”-pane. This module is responsible for the configuration of OKSE. This includes the possibility to add topic mappings.

paginator.js
: This module controls all logic regarding the paginator that appears when the topic/-subscriber tables exceed 25 elements.

okseDebug.js
: This is a jQuery-plugin that appends a logger to the document.

Skeletons

Below is a skeleton for adding a new REST controller. The same applies for normal controllers, with the `@RestController` annotation changed to `@Controller`. When using this annotation, Spring will automatically find the controller on boot, and add the mapping to the dispatcher.

```java
@RestController
@RequestMapping(value = "/api/MYPANE") // Base URL
public class TestController() {

    private static final String TEST_URL = "/get/all";

    @RequestMapping(value = TEST_URL, method = RequestMethod.GET)
    public @ResponseBody void getAllInfo() {
        return "This is a test";
    }

}
```

Listing D.15: Skeleton for a REST-controller

Below is a skeleton for adding a new JavaScript module to the web page for displaying information from a controller. This only shows briefly how one would start to create new modules. To add the JS-file to the administration interface on rendering, append `<script th:src="/js/<MODULE>.js"></script>` to the `th:block` identified with "scripts" in the `indexLoggedIn.html` template.

```javascript
var MYPANE = (function($) {

    var initateModule = function() {

    }

    return {
        init = intiateModule
    }

})(jQuery)
```

Listing D.16: Skeleton for a JavaScript module

```html
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/
      layout"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras
     -springsecurity3"
      layout:decorator="layout">
<head>
</head>
<body>
<th:block layout:fragment="MYPANE">
<div class="tab-pane" id="MYPANE">
...
```

```
</div>
</th:block>
</body>
</html>
```

Listing D.17: Skeleton for a Thymeleaf-template

When adding a new pane it is important to register the panes corresponding JavaScript module in the `main.js` file. This is done by appending a `switch-case` for your pane. It is also important to register the panes template in `indexLoggedIn.html`

D.8 AMQP Send and Receive

Along with the source code, two simple test classes for AMQP are bundled. One that implements a receiver and one that implements a sender. The classes are used for amqprecv.jar and amqpsender.jar files that can be used to test the broker as mentioned in C.8.2 and C.8.1. The example classes are located in the folder `src/main/java/no/ntnu/okse/examples/amqp` starting at the root of the project.

Both classes has Apache Qpid Proton as a dependency and it is recommended that they are used together with Maven to fetch the needed dependencies.

In the following sections, a walk-through of the most important parts of the implementation are provided. To get a deeper understanding of the implementation, the Qpid documentation[10] should be consulted.

D.8.1 AMQPSender

The example below implements the needed code to send a message on a topic to an existing AMQP 1.0 server.

```
// variables
String address = "amqp://localhost:5672/example"
String body = "exampleMessage";
String subject = "exampleSubject";

Messenger mng = Messenger.Factory.create();
mng.start();
Message msg = Message.Factory.create();
msg.setAddress(address);

if (subject != null) msg.setSubject(subject);
msg.setBody(new AmqpValue(body));
mng.put(msg);
mng.send();
mng.stop();
```

Listing D.18: Example use of Messenger to subscribe

[10] `http://qpid.apache.org/releases/qpid-proton-0.9.1/proton/java/api/index.html`

1-3

The first three lines declare variables, the server, port and topic. The second and third declares the message and the subject.

6-7

Lines 6 and 7 creates a message and starts it. The messenger is the client software that actually connects to the server.

8-9

Lines 8 and 9 creates an empty message and sets the address.

12-13

Lines 12 and 13 finishes the message and gives it to the manager.

14

Finally, line 14 will send the message to the given address.

D.8.2 AMQPReceiver

The example found below implements the needed code to subscribe to an existing AMQP 1.0 server.

```
1  // variables
2  String address = ”amqp://localhost:5672/example”
3  int maxMessages = 200;
4  
5  Messenger mng = Messenger.Factory.create();
6  mng.start();
7  mng.subscribe(address);
8  
9  int messages = 0;
10 boolean done = false;
11 while (!done) {
12     mng.recv();
13     while (mng.incoming() > 0) {
14         Message msg = mng.get();
15         ++ct;
16         System.out.println(msg);
17         if (maxMessages > 0 && messages >= maxMessages) {
18             done = true;
19             break;
20         }
21     }
22 }
23 mng.stop();
```

Listing D.19: Example use of Messenger to subscribe

1-2

The first two lines describe the declaration of variables. The first one is the AMQP server, the next is the amount of messages this particular implementation should accept before stopping.

5-7
Lines 5 to 7 creates an instance of the messenger, starts it and subscribes to the address defined earlier. Except for receiving the actual messages, this is basically what must be implemented to actually connect to a server.

12
Line 12 tells the messenger to start receiving from the server.

13-14
Line 13 to 14 receive messages and creates messages as long as there are any to receive.

15 The last lines check if the receive limit is met and stops the program if it is.

Appendix E

Apache Apollo

E.1 Introduction

This appendix focuses on documenting the collected information and impressions of Apache Apollo. As the research phase was a considerable large part of the project and was in a great degree focused on Apache Apollo, it is considered important. The groups customer were also interested in a thorough documentation of the research done. This document is meant to assess that, and is mainly targeted towards them.

This appendix can be used as a standalone document, and will repeat some of the information in the main report.

E.2 About Apollo

Apache Apollo is a message broker system developed by the Apache foundation. Apollo is considered a subproject of Apache ActiveMQ[1] and is considered the next generation of ActiveMQ. Apollo is a protocol agnostic broker that can freely translate between the supported protocols. As of right now, the supported protocols are MQTT, AMQP, STOMP, OpenWire and WebSockets.

Apollo is developed by the Apache software foundation. Most of the development of the project has been done by software developers from Red Hat[2]. As of now, the software is still under continuous development.

E.3 Key features

The following are some of the key features that were emphasis during research on the product.

[1]`http://activemq.apache.org`
[2]`http://www.redhat.com/en`

E.3.1 Architecture

The Apollo architecture is built around the idea of having a multi-threaded, non-blocking implementation using the reactor pattern[3]. The non-blocking approach in a single thread is an approach used to achieve a high amount of connections and throughput. Applying this concept to multiple threads that handles jobs non-blocking gives the application the possibility of scaling very well on computers with several CPU cores.

The code is organized in modules, that are as small and specific as possible. For example, the handler for each protocol is its own module, and can be taken out or added to provide the given functionality. The core functionality like the Web administration interface and the command line interface is also an example of functionality that has been implemented in its own modules.

E.3.2 Scala

As stated by the developers of Apollo, they needed to think differently when designing the application. All tasks that were to be executed needed to be non-blocking, and should be lock and wait free. Changes in the network I/O handling were also needed. To achieve this goal, the object-functional language Scala[4] was chosen by the developers. Scala supports many features from functional programming, and has a good way to express callbacks when writing asynchronous code.

E.3.3 Protocol Agnostic

Apollo supports the same protocols as ActiveMQ, but the way that it handles conversion between them is rewritten. There was a need to have all the tasks running asynchronously, and the way ActiveMQ handles this was quite inefficient. A message sent on a protocol would initially be converted to the OpenWire format (ActiveMQs internal format), and then to the correct protocol.

In Apollo, the handling of protocol conversation is different. It saves a message in its native format until it actually has to convert it to the other protocols. When the conversion is done, it converts directly between the protocols. The way this is done, is quite a bit more efficient.

E.3.4 Message Swapping

When running Java applications, the Java Virtual Machine is usually configured with a fixed amount of memory. Since a high traffic message service can end up having millions of messages, smart memory and message queue handling had to be written. In Apollo, message queues that are not going to be needed for a while will be taken out of memory and put in a message store. When it is time to use the queue, it will be put back into memory. This gives Apollo the possibility to handle a lot of messages with a limited amount of memory.

[3]`http://en.wikipedia.org/wiki/Reactor_pattern`
[4]`http://www.scala-lang.org`

E.3.5 Modules

Apollo is written as a modular application. Every single component and protocol is implemented as its own module. This is a great feature when considering extendability. The module dynamic allows a module to be compiled into a separate Java archive file, which is loaded during start-up. This allows modules to be created or modified without having to recompile the entire application.

E.3.6 WSN support

Apollo is designed to be completely protocol agnostic, using the same port for all connections. Thus it analyzes bit patterns to identify the protocols. The already implemented protocols are all persistent connection protocols, which differ from the open-close behaviour of WSN. It is therefore recommended that any implementation of the WSN protocol use its own dedicated port. This will add some complexity in synchronizing the implementation with the rest of Apollo, but drastically reduce complexity related to connection handling.

The group did not manage to identify any aspect of Apollo which strictly prohibits implementing a protocol such as WSN. Nonetheless, the WSN protocol is a quite different protocol than MQTT, AMQP or STOMP. These inherent differences will make it more difficult to create seamless integration, like the challenges discussed in the Architecture and implementation chapter of this report.

E.4 Evaluation for the project

Apollo is a well written and very scalable broker with many of the features the customer wanted. Apollo would be a good foundation to build on. The main concern with building upon Apollo is the complexity of the broker. The non-blocking asynchronous approach puts a lot of constraints on the developer. Using this pattern, the code becomes quite complex and is difficult to get a grasp on. In addition to the pattern, knowledge of Scala and functional programming is needed.

As the group had limited amount of time available, little knowledge about Scala and no experience with the used programming pattern, the risk of failure was too high. Apollo would take to much time to get a grasp on, and the possibility of not properly understanding the source code and obtain the knowledge needed was an issue.

E.5 Evaluation for later use

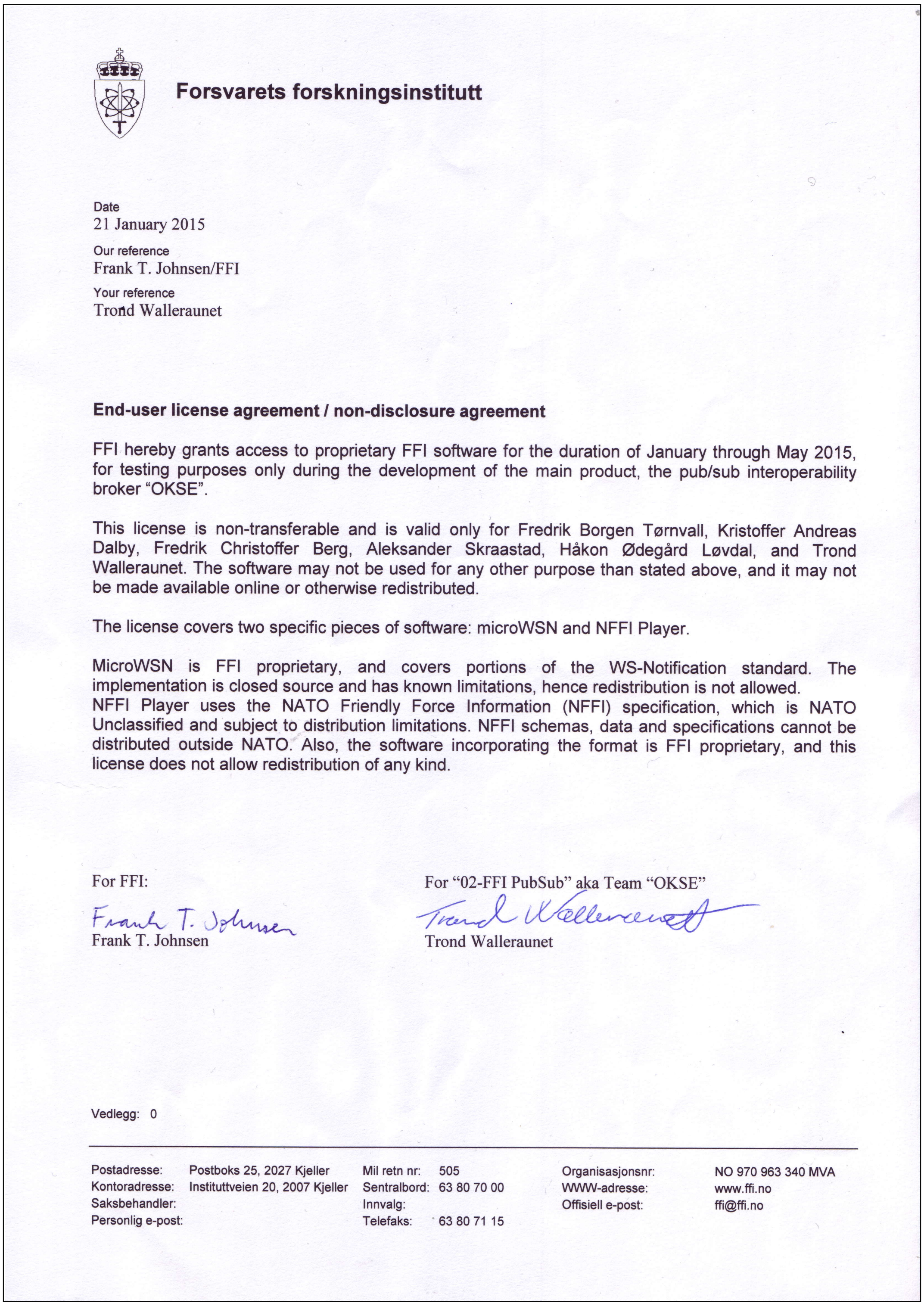
Apollo is a good product, and recommended for further research and use. It is possible to extend, by adding new modules to the source code. If scalability and performance is an important part of the feature set, then Apollo or the ideas and patterns Apollo uses should be considered as a foundation to build upon.

Appendix F

End User Licence Agreement

F.1 Introduction

FFI provided the group with software they could use for testing during the development. To get access to the software the group had to sign a End-User Licence Agreement. The document states that the group can not redistribute the software provided from FFI.



Forsvarets forskningsinstitutt

Date
21 January 2015

Our reference
Frank T. Johnsen/FFI

Your reference
Trond Walleraunet

End-user license agreement / non-disclosure agreement

FFI hereby grants access to proprietary FFI software for the duration of January through May 2015, for testing purposes only during the development of the main product, the pub/sub interoperability broker “OKSE”.

This license is non-transferable and is valid only for Fredrik Borgen Tørnvall, Kristoffer Andreas Dalby, Fredrik Christoffer Berg, Aleksander Skraastad, Håkon Ødegård Løvdal, and Trond Walleraunet. The software may not be used for any other purpose than stated above, and it may not be made available online or otherwise redistributed.

The license covers two specific pieces of software: microWSN and NFFI Player.

MicroWSN is FFI proprietary, and covers portions of the WS-Notification standard. The implementation is closed source and has known limitations, hence redistribution is not allowed.
NFFI Player uses the NATO Friendly Force Information (NFFI) specification, which is NATO Unclassified and subject to distribution limitations. NFFI schemas, data and specifications cannot be distributed outside NATO. Also, the software incorporating the format is FFI proprietary, and this license does not allow redistribution of any kind.

For FFI:	For “02-FFI PubSub” aka Team “OKSE”
Frank T. Johnsen	[illegible]
Frank T. Johnsen	Trond Walleraunet

Vedlegg: 0

Postadresse:	Postboks 25, 2027 Kjeller	Mil retn nr:	505	Organisasjonsnr:	NO 970 963 340 MVA
Kontoradresse:	Instituttveien 20, 2007 Kjeller	Sentralbord:	63 80 70 00	WWW-adresse:	www.ffi.no
Saksbehandler:		Innvalg:		Offisiell e-post:	ffi@ffi.no
Personlig e-post:		Telefaks:	63 80 71 15		