NTNU – Trondheim
Norwegian University of
Science and Technology

Robust Low-level ITS Architecture using OSGi, Reactive Blocks and OPC-UA

Snorre Lothar von Gohren
Edwin

Master of Science in Communication Technology
Submission date: Januar 2014
Supervisor: Frank Alexander Krämer, ITEM
Co-supervisor: Jo Skjermo, SINTEF

Norwegian University of Science and Technology
Department of Telematics

Robust low-level ITS architecture using OSGi, Reactive Blocks and OPC-UA

Many ITS stations today are still in the research phase and “Statens Vegvesen” are moving forward with new improvements. This thesis will study how OSGi and Reactive Blocks can aid in creating an application which is robust and ready for improvements on the fly.

In the end, a scenario is desirable in which ITS applications can be built from Reactive Blocks, where specific capabilities of an ITS station are represented by corresponding building blocks, and the specific application logic can be expressed by combining the blocks accordingly. Generic functions like lifecycle management, service discovery, consistent startup and graceful failures should also be modeled in an understandable way, so that it is easy to upgrade and develop new functions for the ITS stations.

This task will work directly with “Statens Vegvesen” and some of their test stations they have in their regulation. Source code should be made available and virtual test environments can be used.

Interesting questions:

- How can the application be made robust in terms of error handling and edge cases?
- How can the application be upgraded without any inconvenience?
- How can the application be expanded without any inconvenience?

Professor: Frank Alexander Kraemer, Department of Telematics (kraemer@item.ntnu.no)
Supervisor: Jo Skjermo, SINTEF (Jo.Skjermo@sintef.no)

NTNU – Trondheim
Norwegian University of
Science and Technology

Robust Low-level ITS Architecture using OSGi, Reactive Blocks and OPC-UA

Snorre Lothar von Gohren Edwin

Submission date: January 2014
Responsible professor: Frank Alexander Kraemer, ITEM, NTNU
Supervisor: Jo Skjermo, SINTEF

Norwegian University of Science and Technology
Department of Telematics

Abstract

The vehicle traffic flows on a daily basis, and it works. People get from A to B, sometimes using longer time than expected because of traffic or accidents. Transport vehicles arrive without a notice and valuable traffic and environmental data is not collected. How should the future traffic control be and what are the possibilities?

These are issues which concern Statens Vegvesen on a daily basis. Currently there are ongoing research figuring out how a general Intelligent transport system (ITS) station should be specified. A high level functional specification has been written, and propose multiple use cases and functionality for the future. This thesis will embark on the low level technical issues regarding a general ITS station. Focusing on robustness, upgrading and expanding, combining Reactive Blocks (RB), Open Services Gateway initiative (OSGi) and Open Platform Communications Unified Architecture (OPC-UA) in a technical architecture.

Statens Vegvesen have pointed out that these three terms are important factors to be handled, for them to be able to develop ITS stations to the satisfactory level they strive for. So if this thesis can show that it can become easier to create new applications with RB, opening up for application updates without downtime through OSGi and providing a familiar communication layer with OPC-UA. That will benefit Statens Vegvesen in many ways.

The reason for the selection of these technologies are that the domain is a perfect match for OSGi, providing the lifecycle aspect in to the architecture. Statens Vegvesen is familiar with OPC and OPC-UA, it provides a way of standardizing hardware and enabling an advanced communication protocol out of the box. RB is a technology which is created at the institute this thesis is written for, and merge two difficult domains by visualizing code through blocks and flows, meaning that better code control is provided. So they all have their specific traits which can be combined into a valuable consolidation.

A prototype architectural application was made through out several iterations discussing the different issues which arose during development. It became a foundation platform which allowed for expansion by deploying additional functional OSGi bundles.

The technological combination with the specific focuses became a satisfactory prototype architecture. It enables the application to react on edge cases in a useful way, upgrade the software to handle the edge cases without downtime, and improve the station with new applications for different use cases.

This original technological combination have paved the way for a solution to be contrived in the coming years. This thesis is meant as a foundation to decide if this is a fitting road towards a distinguished ITS station architecture.

Sammendrag

Trafikken flyter til daglig, og det fungerer. Folk kommer seg fra A til B, noen ganger på lengre tid en forventet, på grunn av trafikk eller ulykker. Nyttetransport ankommer uten forvarsel og verdifull trafikk- og miljødata blir ikke registrert. Hvordan skal fremtidens trafikkontroll være, og hva er mulighetene?

Dette er utfordringer som Statens Vegvesen jobber med til daglig. Det pågår nå forskning på dette feltet, hvor man prøver å finne ut hvordan en generell Intelligent Transport System (ITS) stasjon skal være spesifisert. Det er blitt skrevet en høynivå funksjonell spesifikasjon som foreslår flere bruksområder og fremtidig funksjonalitet. Denne oppgaven skal ta for seg lavnivå tekniske problemer når det kommer til ITS stasjoner. Den vil fokusere på robusthet, oppgradering og utvidelse, ved å kombinere RB, OSGi og OPC-UA som en teknisk arkitektur.

Statens Vegvesen har påpekt at de tre nevnte termene er viktige faktorer som må håndteres for at det skal bli mulig for Statens Vegvesen å utvikle ITS stasjoner som er på det nivået de streber etter. Hvis denne oppgaven kan vise at det vil bli lettere å utvikle nye applikasjoner med RB, åpne opp for applikasjonsoppdateringer uten nedetid gjennom OSGi og legge til rette for et familiært kommunikasjonslag med OPC-UA. Så vil dette gagne Statens Vegvesen.

Grunnen til at disse teknologiene er valgt er fordi dette domenet er en perfekt match for OSGi, ved å tilby et livssykel aspekt til arkitekturen. Statens Vegvesen er kjent med OPC og OPC-UA, som tilbyr en måte å standarisere maskinvare og legge til rette for en avansert kommunikasjonsprotokoll på en enkel måte. RB er en teknologi som er laget på instituttet denne oppgaven skrives for, og slår sammen to vanskelige domener ved å visualisere kode gjennom blokker og flyt, som gir bedre kontroll over koden. Så alle har deres spesifikke trekk som kan bli kombinert til en verdiful løsning.

En arkiteturisk prototypeapplikasjon ble laget gjennom flere iterasjoner hvor de forskjellige vanskelighetene, som dukket opp, ble diskutert. Den ble et fundament som legger til rette for utvidelse gjennom å installere flere funksjonelle OSGi bundler.

Denne teknologiske kombinasjonen med de spesifikke fokus områdene, ble en tilfredstillende prototypearkitektur. Den åpner opp for applikasjo-

nen til å reagere på ytterpunkt på en bra måte, oppgradere applikasjonen til å håndtere ytterpunktene uten nede tid og forbedre stasjonen med nye applikasjoner for forskjellige brukerscenarioer.

Denne orginale teknologiske kombinasjonen har lagt veien for at en fremtidig løsning kan bli utarbeidet. Denne oppgaven er ment som et fundament for å kunne avgjøre om dette er den riktige veien å gå for å nå en utmerket ITS stasjonsarkitektur.

Acknowledgments

"Appreciation is a wonderful thing: It makes what is excellent in others belong to us as well."

— Voltair

First of all I want to thank my supervisor Frank Alexander Kraemer who has had two hats on during this process. He has represented a supervisor from ITEM as well as technical expert from Bitreactive. He has provided me with valuable questions and important info regarding ReactivBlocks.

Second, Statens Vegvesen have been an important factor in this thesis by providing an assignment which was based on real world problems and future development. Thanks to Erik Olsen in the lead from Statens Vegvesen.

In cooporation with Statens Vegvesen is SINTEF IKT og samfunn. They have provided with valuable insight in the research projects related to my assignment. Thanks to Jo Skjermo and Trond Foss at SINTEF.

The OSGi community have been a supportive organ in the questions I have asked on the different channels available. Thanks to these people, they include, Jeff Goodyear, Peter Kriens, Richard S. Hall, Neil Bartlett, Jean-Baptiste Onofré, Justin Edelson and David Jencks.

Lastly I want to thank Jouni Aro from Prosys OPC who have been a valuble sparring partner when it comes to technological questions regarding OPC-UA. He made himself available through the Prosys OPC forum.

Contents

List of Figures

List of Code Snippets

List of Glossary

Android This is the Google OS for tablets and smart phones.

Apache Felix This is an OSGi spesification implementation by opensource company Apache.

BND Tools an easy, powerful and productive way to develop with OSGi. Based on bnd and Eclipse.

Bundle The name convention of the JAR files OSGi deploy in the framework.

C is a general-purpose programming language, which is one of the most widely used programming languages of all time, and C compilers are available for the majority of available computer architectures and operating systems.

C++ is a programming language that is general purpose, statically typed, free-form, multi-paradigm and compiled.

Configuration Admin The OSGi Componendium Configuration Admin Service specifies a service, which allows for easy management of configuration data for configurable components.

Declarative Services (DS) is a component model that simplifies the creation of components that publish and/or reference OSGi Services.

ESM This is the external state machine which encapsulates a building block.

Freeware is software that is available for use at no monetary cost or for an optional fee.

IRC stands for Internet Relay Chat whic is a protocol for live interactive Internet text messaging (chat) or synchronous conferencing.

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible.

Java Enterprise Edition The platform provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications.

JavaDoc is a documentation generator from Oracle Corporation for generating API documentation in HTML format from Java source code.

.NET is a software framework developed by Microsoft that runs primarily on Microsoft Windows.

PID This is an OSGi persistance identifier which is used to identify the different services within a confiuration admin.

SINTEF The SINTEF Group is the largest independent research organisation in Scandinavia.

Web Services This is a method of communications between two electronic devices over the World Wide Web.

List of Acronyms

API Application programming interface.

EC European Commission.

EJB Enterprise JavaBean.

EU European Union.

ITS Intelligent transport system.

JAR Java Archive.

JBI Java Business Integration.

JMX Java Management Extensions.

MVC Model view controller.

NTNU Norwegian University of Science and Technology.

OPC-UA Open Platform Communications Unified Architecture.

OS Operation system.

OSGi Open Services Gateway initiative.

OWL Web Ontology Language.

RB Reactive Blocks.

SDK Software development kit.

SOA Service oriented architecture.

SOAP Simple Object Access Protocol.

TCP Transmission Controll Protocol.

URL Uniform Resource Locator.

Chapter 1
Introduction

"The introduction is like a first impression, either it seems interesting or plain boring"

— Snorre L.v.G Edwin

1.1 Project background

Statens Vegvesen, the state-owned company in charge of the Norwegian road network, is in a process of developing highly technological ITS stations. The work is done closely together with "SINTEF Teknologi og samfunn", who also is the authors of the specifications of functional and technical requirements. [67] This project was initiated by multiple people within ITS section in Statens Vegvesen. And as Erik Olsen said, one of the initiators, "There are multiple people who have a foot in this project and cooperative systems would not be a result of a single persons effort, rather a cooperative effort by multiple people". It has been initiated because ITS is something that is an important center of attention for Statens Vegvense. An early focus on the future problems and possible architectures has been important for Statens Vegvesen.

1.2 Problem outline

This thesis will study how the Open Services Gateway initiative (OSGi) spesification [62] and Reactive Blocks (RB) [30] can aid in creating an application which is robust and ready for improvements on the fly for roadside ITS stations.

In the end, a scenario is desirable in which ITS applications can be built from RB and OSGi bundles, where specific capabilities of an ITS station are represented by corresponding building blocks, and the specific application logic can be expressed by combining the blocks accordingly. Generic functions like lifecycle management, service discovery and graceful failures should also be modelled in an understandable way, so it is easy to handle errors, upgrade and develop new functions for the ITS stations. The communication framework which will be used is the OPC-UA specification, which is currently in version 1.02, meaning very bleeding edge. The

Software development kit (SDK) which will be used is made by a Finnish company called Prosys OPC [64]. They provided an evaluation SDK which was used during the thesis. This is the same SDK that Statens Vegvesen is using.

There will be direct communication with SINTEF and Statens Vegvesen to figure out their edge cases and how they want to see the application work. If possible, Statens Vegvesen will try to have a test station available, so that the test application can run on live stations. Source code will be made available[1], and through the development phase, virtual test environments and dummy data will be used.

The methodology used is based on iterative experimenting with the chosen technologies and reading documentations and specifications for the different technologies. As well as reading information and specifications covering Statens Vegvesens view on ITS stations, and other related work that has been done in Europe. [10]

It was earlier proposed to focus on how too gather data from these specific stations, but it was early decided that this is not in the scope of this thesis. Rather be moved to another independent thesis.

1.3 Research questions

The following problems are based on discussions with Statens Vegvesen and some of the issues they are experiencing with the current state of the research project. The questions are also influenced by the point of view that the professor and the writer of this thesis have. They both come from the Institute of Telematics at Norwegian University of Science and Technology (NTNU), and have an interest in dynamic architectures and state-full applications.

RQ1 How can the application be made robust in terms of error handling and edge cases?

RQ2 How can the application be upgraded without any inconvenience?

RQ3 How can the application be expanded without any inconvenience?

1.4 Limitations

There have not been many limitations because the technologies work pretty well together. But some limitations are based on what phase Statens Vegvesen is at, that the specification itself is currently being written as we speak, meaning that a

[1] The repo is private so contact the writer for access, through the information on his GitHub account.[21]

concrete direction for this project has not yet been set. Giving some uncertainness on were to focus.

The thesis have worked with simulation data based on an earlier project and not any real data from sensors meaning that some of the discussion have not taken into account all the details which might be available on hardware itself.

As for RB who just started to support OSGi, meaning that there had to be some workarounds with OSGi and it is of course still to be discussed on how much of the OSGi technology should be incorporated into the RB framework.

Another limitation is that there are three advanced technologies and there are endless possibilities with these technologies meaning that the writer does not have had the opportunity to devour all three technologies completely. Which results in some discussions based on experience and what background level the writer is.

1.5 Scope and organization

The thesis have been organized in a structure which first introduce the technical background of the technologies, and then works on to related work and literature evolving the three technologies. After that it will discuss how these results becomes reality through the methodology. Moving on to the discussion involving the experiments and our research questions. It will first compare the three terms from the research questions against the technologies, then it moves on to multiple iterations which experiment with different scenarios and unsolved problems. It all ends with a evaluation and conclusion.

1.5.1 Technical background

This thesis will not dive into the specifics of OSGi, Reactive Blocks nor OPC-UA. It will give a light introduction to what makes these technologies what they are today, and refer to other articles and pages were it is possible to learn more about them. It will rather focus on what might make these technologies a good match with the current problem outline 1.2.

1.5.2 Literature and related work

This chapter will go through some related work based on the technologies used and connecting them to the research questions this thesis is facing. There have not been a lot of work directly related to what this thesis is covering because it is something new. These technologies have never been used together in a project, and especially not within the ITS station. But there are plenty of others who work on ITS on a

higher level, which is mentioned, as well as others who are trying to cover some of our research questions using OSGi.

1.5.3 Methodology

This chapter provides the current methodology used and how the work of this thesis have been structured. It also mentions and early phase architecture which includes a design for the control station. This is included just to convey how the ITS station could communicate with the control station. But the technicalities and more detailed architecture surrounding the data retrieval from the stations is not in the scope of this thesis.

1.5.4 Iteration 0: Technology study

This chapter will address the problem outline and compare it to our technologies. It will try to explain what the different technologies can aid with, in a solution for our research questions.

1.5.5 Iteration 1: Connection and pair-ability

This iteration is the initial iteration which will concentrate on the fundamental connection of the three technologies and what might be a proper fit. There are multiple ways to solve this and they can be used in numerous ways. This iteration also sets some guidelines for future work.

1.5.6 Iteration 2: Implementation of simple swap example

Here the thesis will hammer away at a special scenario, which is an implementation of a simple hardware swap example focusing on OSGi and implementing some of the OPC-UA features. This iteration will try out the guidelines and measure them up with a test application. There will be some experienced problems and proposed solutions in this chapter.

1.5.7 Iteration 3: OPC-UA Client and hardware approach

The application from the last iteration had some obstacles which had to be taken care of for a better understanding. The main focus here is to use OPC-UA connected to the hardware and propose a fitting implementation. It will focus on objectmodels for OPC-UA, and better connection between the hardware & OPC-UA client, and OPC-UA client & RB.

1.5.8 Iteration 4: Reactive Block focus

RB does not have had a complete focus through out the iterations. Therefore its time to further develop the application with a focus on RB. This iteration will continue from the state of the last application and go through a developing phase were the application becomes upgradable in different modules and use the OPC-UA layer completely in cooperation with the RB layer.

1.5.9 Evaluation and conclusion

It all ends with an evaluation and conclusion from the latter chapters. Some guidelines for future work will also be set here.

Chapter 2
Technical background

"One's mind has a way of making itself up in the background, and it suddenly becomes clear what one means to do."

— A. C. Benson

2.1 Intro

This thesis uses multiple different technologies which may be new to the reader. Therefore the technologies will be lightly introduced in this chapter with referrals to other comprehensive articles.

As mentioned in the introduction OSGi and RB are used as architectural technologies and OPC-UA as an communication protocol. The reason these three technologies are interesting to combine into an architecture which is robust and expandable is because they all provide their own important functionalities and benefits. For example OSGi provides the opportunity to create a modular and dynamic application where all the normal JAR files, which is called bundles in OSGi, might have their own lifecycle. This will be greatly introduced in the section 2.2.

RB on the other hand accommodate the application with a structural overview of how the application is built up. A user creates building blocks which gives the opportunity for reuse and connects the blocks with flows to display how the event movement and logic creates the specific application. It will be covered further in section 2.3.

The last technology, which is OPC-UA, originally stands for Object Linking and Embedding(OLE) for Process Control - Unified Architecture. It is an important part that paves the way for users to be able to build the next generation of software automation solutions. It is a communication model which is cross-platform Service oriented architecture (SOA) for process control. [57] Can be understood as a way to standardize the communication towards hardware. It is introduced further in section 2.4.

2.2 OSGi

The writer of this thesis did a pre study on OSGi which can be obtained from [70]. The focus was modularity and lifecycle in an OSGi application. Based on this article the interest to involve OSGi in this thesis was high.

”Open Services Gateway initiative (OSGi) technology is a set of specifications that defines a dynamic component system for java. These specifications reduce software complexity by providing a modular architecture for large-scale distributed systems as well as small, embedded applications.” [62]

2.2.1 OSGi growing up

The OSGi specifications was started in 1998 and its main intention was the home automation market. They wanted to solve how to build applications out of independent components.

This model is said to be the first model which ever achieved their promise to create a component system that now solves many real dilemmas in everyday software development. Developers who see the benefits of OSGi after its adoption can rapport that it has saved them multiple hours in all aspects of development. For example: reuse, code is easier to handle, deployment becomes effortless and developers could start to think differently.

Because of this positive change to development, many applications have adopted this technology, and OSGi is used in popular applications like Eclipse, Spring and multiple applications servers. [41, 22]

2.2.2 Related work

There has been a lot of related work trying to handle the inadequacy that java creates when it comes to modularization. The volume of related work has grown over the years but there are a great deal of the different solutions which is not directly comparable to OSGi. Some of these technologies are Java Enterprise Edition which started almoast at the same time as OSGi, but on the other side of the computing spectrum. But in the Java Enterprise Edition space, the thing that is closest comparable to OSGi, is the Enterprise JavaBean (EJB). [38]

Moving on there is Java Business Integration (JBI) which provides the possibility to plugin components and consume services. Which might seem pretty much like OSGi with a quick glans, but OSGi is so much more, and its more natural to use OSGi with JBI to create the modularity wanted. This actually happened in ServiceMix from Apache. [44]

In 2006 JSR 294, [37], was introduced under the name "Improved Modularity Support in the Java Programming Language". This was a derivate from JSR 277, [53]. During this time a project called "OpenJDK Project Jigsaw" was introduced, hosts the Reference Implementation of JSR 294. This technology is the only one that can be said to try to solve the same problem as OSGi. It is going to be baked right into the java platform itself.[54] Some problems that have been discussed around Jigsaw, is that it is not very mature and has a long way to go before it reaches the stages that OSGi is at right now. But if it is done right, it can eventually become a popular competitor of OSGi.[63] It was first planned to ship with java 7, but has now been deferred to java 9 to allow more time for both development, review, testing and feedback.[54]

But one important factor to have in mind, is that the two technologies should to be compatible with each other. It would be a shame if these two platforms couldn't run unitedly, just because of tension existing between the OSGi and Jigsaw communities. [52, 54]

2.2.3 OSGi architecture

OSGi can basically be viewed as a set of specifications, usually made possible through the standard java manifest file, MANIFEST.MF, with the OSGi metadata. This defines a dynamic component system for the java technology. The environment enables applications to be built with components that can be dynamically added and composed by other reusable components. They all hide their internals and communicate through services, 2.2.3, which control objects called bundles, 2.2.3, that can specifically share internals between each registered bundle using the manifest file. It will utterly be explained in the following text.

Layers

The OSGi architecture is developed in layers as you can see in 2.1. This provides benefits to creating and managing java applications since there can be developed applications and bundles across each layer, and have better control of the system. All layers can be used without the layers that are above them, but not visa versa.

OSGi bundles are the main part of the OSGi specification and is why it works so well. Bundles might look like regular JARs but differentiate with OSGi metadata in the MANIFEST.MF file,2.2.4, which declares essential information about the bundle.

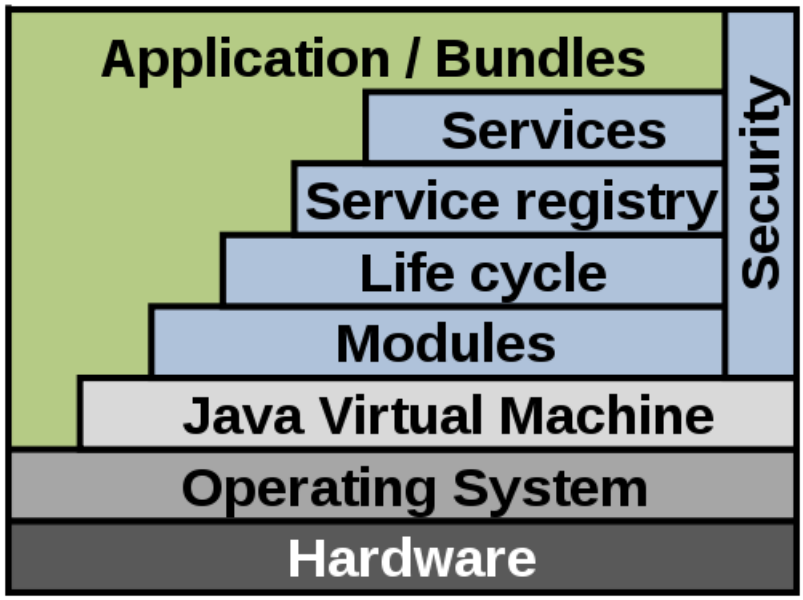

Figure 2.1: OSGi layering, taken from [73]

Bundle definition

"A physical unit of modularity in the form of a JAR file containing code resources, and metadata, where the boundary of the JAR file also serves as the encapsulation boundary for logical modularity at execution time." [47]

The bundle have to run inside an OSGi container which handles the MANIFEST.MF properly. Meaning that the container will do actions based on the properties the MANIFEST.MF file has. These properties are explained utterly in 2.2.4. Bundles can be deployed dynamically and have a lifecycle attached to it, 2.2.3. This allows the environment to be dynamic and services and bundles can come and go. As for services, 2.2.3, this is something that a bundle can provide. The reason the architecture is specified this way, is to endure the loose coupling and have modularization in focus. Providing opportunities to upgrade and expand an application without having to restart a system.

Services is an important part of OSGi. Java makes it complicated to write a collaborative system with only class sharing. A standard solution for this problem is to use factories. It works, and it is possible to provide the system with f.ex a *DocumentBuilderFactory* using this technique. But as a developer there is no

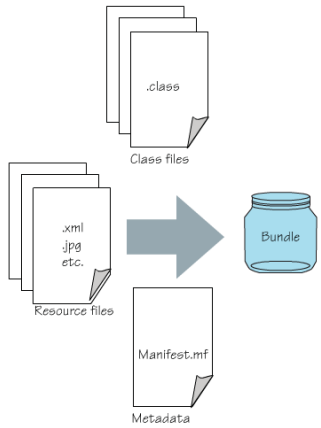

Figure 2.2: What a OSGi bundle is composed of, taken from [47]

opportunity to influence the implementation that is used, and by implementation, refer to the service model, properties and conventions in the current class.

A service is an interface, with an implementation behind which a bundle can register with the service registry, 2.2.3. Since this is separated into an interface and implementation, it gives developers the possibility to use mulitple implementations of the same interface. This can for example be different versions with new functionality. This opens up for easier ways to upgrade dependent of the situation. Services also facilitates better expansion of the application whit dynamically tracking the services needed.

Service registry is the reason services work in OSGi. The factory model mentioned in, 2.2.3, is a passive model, which means that the implementation just exists, with one implementation available. It is not possible for the model to announce that it is available for use, list the different possible implementations, nor be dynamic.

This is where the service registry solves a very important problem. For a service registry overview, see figure 2.3. A service will always register with the service registry when its ready to be deployed. By using the service registry the developer can list possible implementations available in the registry, or even wait for a callback when a specific service appears. Also the service can be removed from available services when it is deprecated.

Since bundles are dynamic, as been mentioned earlier, they can suddenly be withdrawn and other existing bundles using its service may crash because the service is no longer available. To avoid this might seem like a truly complicated task, but

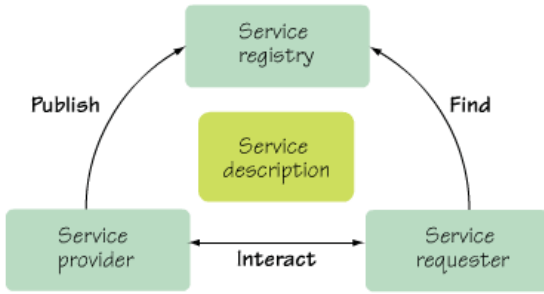

Figure 2.3: The SOA view of OSGi, taken from [47]

it exists classes, utilities and callbacks in OSGi which minimize the complexity utterly. Some utilites, which was tested out in this thesis, are Service Tracker and the Managed Service Factory. It later moved on to Declarative Services which is coverd in chapter 8. [24, 23] Chapters 5 and 701, and chapters 104.6, and chapter 112 in correct order.

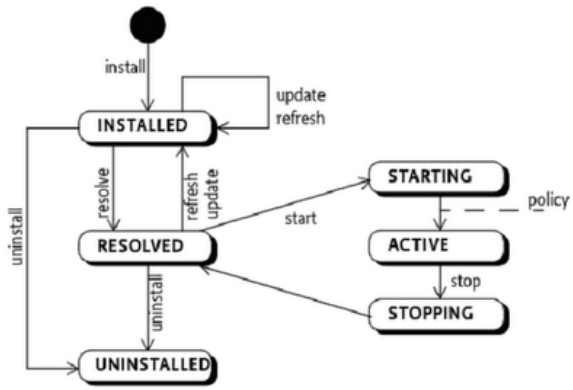

Figure 2.4: OSGi Life cycle, taken from [66]

Life cycle is the OSGi layer which defines how bundles are dynamically installed and managed. For example, if you were creating a car, the modules, 2.2.3, are the engine and chassis, while the life cycle would be described as the wiring of the car which provides life to the car. State diagram, 2.4, shows very easily what states that can be transitioned too and from. It also shows that a bundle only have one start state and one end state. So basically a bundle has to be installed before it can be resolved. For a bundle to become resolved, it has to be able to access all its declared dependencies. If a bundle is missing a dependency, it will stay in state installed and provide the developer with an error. This makes it easy handle dependency troubles.

There are many questions around a bundles life cycle. For example what if a dependency doesn't exist in the current environment? What if a bundle, that has been acquired by other bundles, suddenly decides to stop? These can be further studied in the pre studies earlier mentioned. [70]

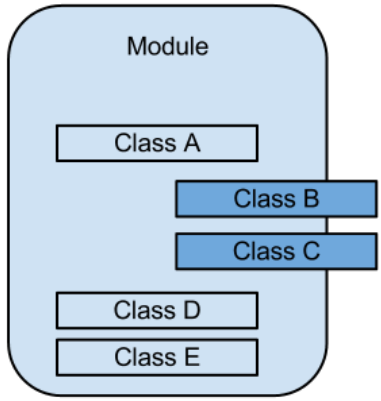

Figure 2.5: Representation of a module, adapted from [47]

Module is the layer which is the foundation on which everything else rests upon in the OSGi world. The module layer addresses all shortcomings of java's dependency handling and further improves it. Giving OSGi great opportunities and benefits of thinking differently when designing and developing applications.

Module definition

"A set of logically encapsulated implementation classes, an optional public Application programming interface (API) based on a subset of the implementation classes, and a set of dependencies on external code." See figure 2.5. [47]

There might be some confusion on the difference between a module and a bundle. Module is a more abstract term, and this layer has strict rules for logical boundary, as mentioned partially in the definition above. Bundle is how OSGi refers to their solution of the module concept, and is more the implementation, the JAR itself.

Modularization aids drastically in reference to the research questions and terms. It is what encapsulates the application and gives the developer options to provide modules with different types of functionality. Making it easier to upgrade specific functionalities or adding new functionality to the system.

Security layer underlies all the different layers as displayed in Figure 2.1, and is an optional layer. It is based on java 2 security, but OSGi has added a number of constraints and fills in some of the blanks that is left open by the standard java. The security layer defines the runtime interaction with java 2 security layer as well as a secure packaging format. [24]

Some essentials for this layer is that it provides the infrastructure to control applications running in an fine-grained OSGi framwork. Also the security layer does not define an API for itself, rather the management of this layer is left to the lifecycle layer, 2.2.3. But it is possible to use this layer to its own advantage, by controlling which bundles can access what services and so on.

2.2.4 MANIFEST.MF

The MANIFEST.MF file is already existing in the java world, but OSGi has decided to take advantage of it. This is where developers define some important static properties for their current bundle, and it is important for the whole OSGi concept. Here is a current manifest file.

Properties 2.1 An example of MANIFEST.MF

```
Manifest-Version: 2.0
Bundle-Name: DataService
Bundle-SymbolicName: no.dataservice.service
Bundle-Version: 1.0.0
Bundle-Description: Demo Service
Bundle-Vendor: Snorre
Bundle-Activator: no.DataService.impl.Activator
Bundle-Category: example
Import-Package: org.osgi.framework
Export-Package: no.DataService
**Service-Component: OSGI-INF/component.xml**
```

The first line of this MANIFEST.MF file indicates the OSGi metadata syntax version. Where 1.0 defines that this follows OSGi release 3 and version 2.0 is OSGi release 4 and later. Bundle-name defines a readable name for this bundle. This should be a short, human-readable name that can contain spaces. The Bundle-SymbolicName header specifies a non-localizable name for this bundle. The bundle symbolic name together with a version must identify a unique bundle, though the bundle can be installed multiple times in a framework as different version. The bundle symbolic name should be based on the reverse domain name convention and must be set. The next line specifies which version this bundle is and the description

gives a short explanation to the bundle. The Bundle-Vendor header contains a human-readable description of the bundle vendor. The Bundle-Activator header specifies the name of the class used to start and stop the bundle if activator is the way to go. The Bundle-Category header holds a comma-separated list of category names. [24]

The two package lines of the MANIFEST.MF are very important! They define what packages this bundle should make visible to other packages, and what packages it should import. This has to be included if a bundle should depend on some other class, and only adding to the build path is not enough. If a bundle has an API it should be added to the export path.

The last line is added because its important to see how Declarative Service components are activated, because it is used within the thesis. If this line is added the *Bundle-Activator* can be removed. The handling of a life cycle is taken care of by the Declarative Service component.

Many of these lines are just for human eyes and is not interpreted by the OSGi framework. There are only 5 lines which will be interpreted here, and that is the version, symbolicname, activator, and the two package lines. To be picky, there are more headers which is interpreted, but it depends if DS is used or not. There may be many more headers in a MANIFEST.MF file but its not in the scope of this rapport. If interested, take a look at the OSGI specifications [24].

2.3 Reactive Blocks

Reactive Blocks (RB) SDK is an engineering tool to build solid, highly concurrent and event-driven systems from reactive building blocks, fig 2.6. [30] It allows developers to create applications which can be deployed to any device or hardware running java. Bitreactive with RB solves three fundamental software design principals which cannot be called new. They just combine them in a proper way.

The list of principals:

- Divide and conquer, then reuse effectively
- Visualize your system
- Automatically and formally verify everything, always

The engine behind, the RB SDK is a set of tools currently integrated into eclipse which allows the developer to use a combination of java code and a graphical editor to create complete applications.[36] The graphical editor provides the overview needed to solve problems without loosing track of the code. And because of the seamless

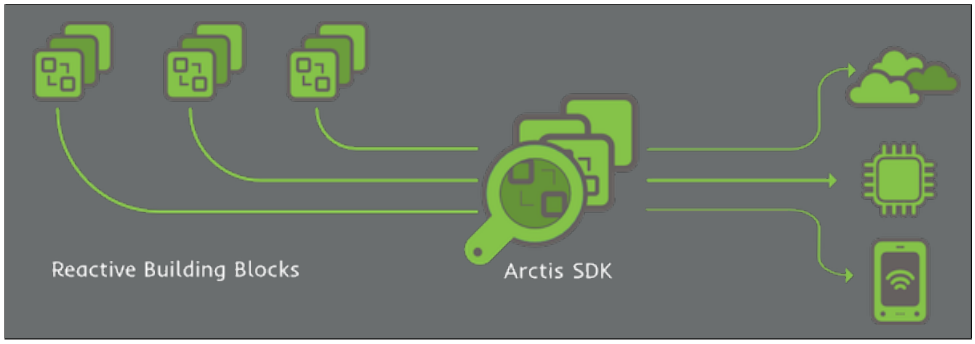

Figure 2.6: Overview of reactive blocks [30]

integration with Eclipse JDT it opens up for java coding which removes the thoughts of limitations which might occur when the developers are provided with a graphical editor.

2.3.1 Building blocks

This term has been mentioned, and its important to clarify what a building block is. It is a reusable software module, which is designed to handle event-driven systems with ease. A block may seem like a simple API, but they also have all the logic, states and code to handle that give use this building block has been designed for. Blocks combine graphical data flows and code. The data flow takes care of concurrent behavior and synchronizations. Code describes detailed operations on data.[30]

In figure 2.7 it is possible to see a building block provided from a library. The *Simple Service Tracker* is a implementation of an OSGi *Service Tracker* which is very easy to reuse. All that is done by the user is to define what kind of service it is suppose to handle. And of course draw up the flow to create the actions from the different output pins of the *Service Tracker*.

There exist different types of building blocks dependent of what the developer wish to achieve.

The list of building blocks:

- *General blocks*
  - System block
  - Local block
  - Shallow block
- *Android blocks*

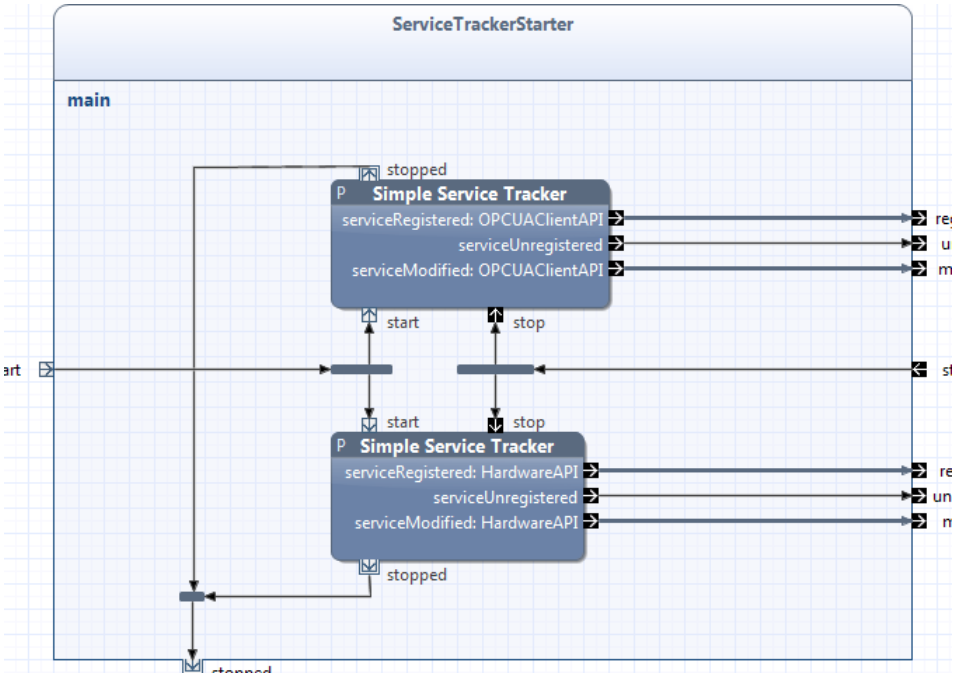

Figure 2.7: Building block usage

- Android Application
- Android Building block

A system block is the upper most level of an block or application. The application always populates from the root system block and wanders down into combinations of local blocks, singeltons and shallow block. It also exist Android blocks for developing Android applications with their respective SDK. But this is not something that will be further discussed.

System block

The system block is the initiator of the whole application, the main method so to say. There exists only one system block per generated JAR file. A system block is instantiated with an initial node, figure 2.8, which generates an initial control flow, that can lead to multiple local blocks which becomes the application. The running JAR is a compilation of all the blocks within the system block, so there will not be a JAR for each block used. It is also possible to compile into an OSGi bundle, which is what this thesis will be doing. The only difference is that it will be possible to

use OSGi services inside the application and the bundle can be deployed in an OSGi environment. But the system block will still only be one bundle.

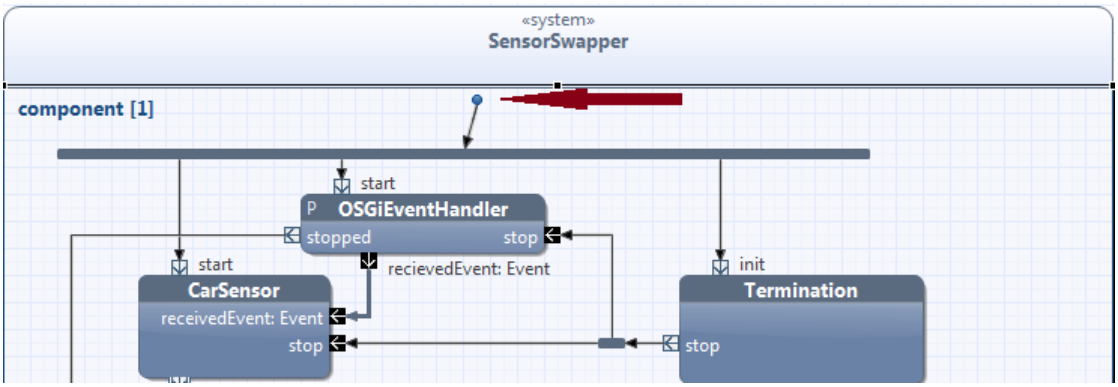

Figure 2.8: System block with an initial node RB

Local block

This is the block most commonly used because these are what creates the application inside the system block. Since RB is built up of an hierarchy, there can be multiple levels with RB. Meaning that a local block may have a complete set of blocks within, except for a system block. This thesis will not go into details of how these blocks can be modified or added parameters and such, please refer to the developer page at Bitreactive. [32]

Usually, each building block is instantiated once, which probably is expected. But besides this normal case, there are some other multiplicity patterns (or session patterns) that are possible, and that allow to handle various cases. The possibilities are singleton, multi-session, single-session and of course normal. Depending on the session type, communication with the session works differently, for more detailed explanation see [35]

Shallow block

A shallow block only has parameter nodes and external behavior (ESM). It does not contain any Java code. This is just like a local block which shall be reused inside another local block or a system block. Its only function is to regulate and organize flows in a more robust way than regular activity nodes like merge and join can do. For example the *Switcher* in figure 2.9 is a simple flow switcher which starts on default *out1* for the first incoming flow. If there has not been any flow to the *switch* input, the next flow will also head out of *out1*. With this *Shallowblock* its possible to control where flows should go, just like a train.

External Behavior (ESM)

Source State	Parameter Nodes	Target State
<initial>	in / out1	<final>
<initial>	switch /	out2
out2	switch /	<final>
out2	in / out2	out2
out2	/	<final>
out2	in / out2; switch	<final>
<initial>	in / out1; switch	out2

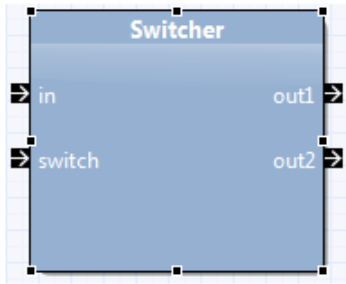

Figure 2.9: A shallowblock called *Switcher* from the library

2.3.2 Reuse

The RB SDK allows you to use existing building blocks from other providers, or yourself, by simply drag and dropping from an existing library or project. This facilitates a high focus on reuse when creating building blocks, which allows for more efficient coding and the implementation of building block standards. [33]

Currently in the provided library from Bitreactive, it exists heaps of already implemented blocks which can be used directly into an existing project. Which is kind of an ”app store” for blocks.

2.3.3 Visualization

This is an important factor for having an overview of the existing code. People generated diagrams all over the place to create an overview of the specific code but an issue is that the code can be refactored thoroughly, and then its important to update the diagrams. This provides extra work and may often be forgotten. That is were RB generates the diagrams as we create the logic. RB let you have the cake and eat it too.

2.3.4 Verification

Its important to get feedback on errors if something is wrong with the graphical model or the code base during development. Because its important to reduce errors at run time by avoiding them in the first place, catching them at development time. This is solved in RB with an analyzing tool provided in the graphical editor. It is possible to analyze each block seperatly because they are encapsulated by their contracts. [36]

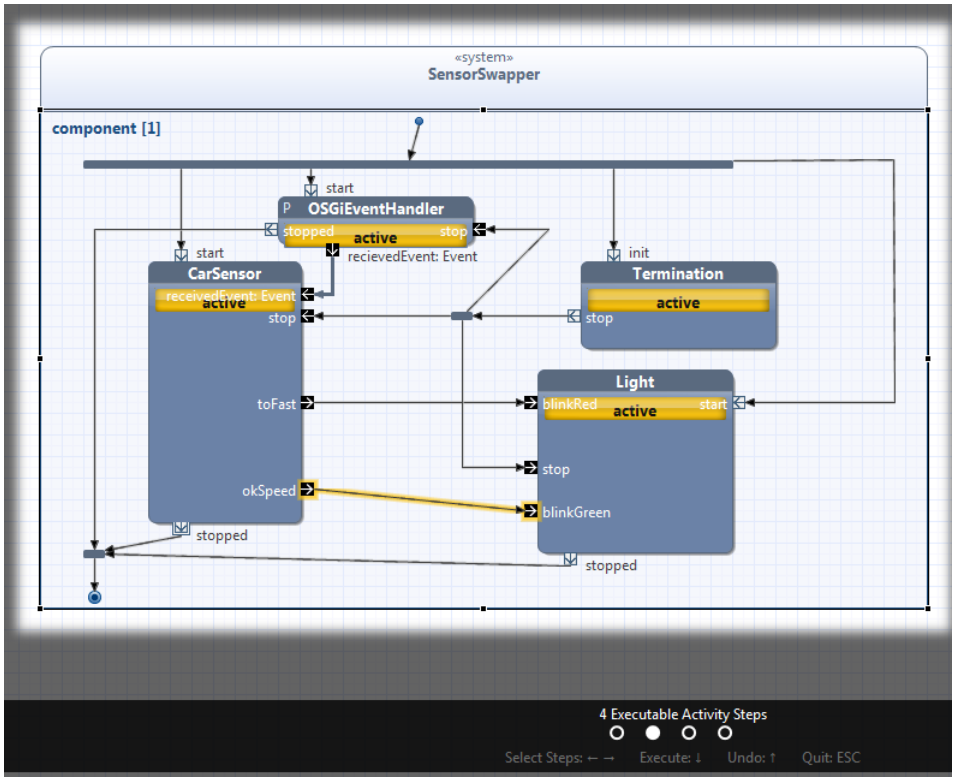

Figure 2.10: The analyzing tool in action

As figure 2.10 displays, its possible to walk through all the steps which might happen within the application to verify that your thoughts actually are realized. For example what happens to the activity diagram when *okSpeed* is activated and the flow goes over to *blinkGreen*? This tool will then visualize this by ”executing” this flow and see what state and possibilities the flow diagram then have. Parts of this analysis also happens at coding time providing the developer with useful feedback on what is possible and not.

2.3.5 Why use Reactive Blocks?

Because complete applications are created reusing multiple building blocks and system blocks which uses event driven data flows. This is the perfect combination for concurrent applications, because it solves event driven problems visually which promotes a descriptive control. Aiding in the possibility to have a shared understanding of the current application which is a wanted feature for users from other domains than the developers.

2.4 OPC Unified Architecture

OPC-UA is a result through a long lasting collaboration of industry leaders that had the goal to create an open standard which enables the exchange of information in a rich, secure and object-oriented way. There is an already existing version of OPC specifications which rely on Windows technology and have a lot of restrictions because of that. For example it is dependent on windows technology, lack of security, no configurable time-outs and so on. It had alot of smart functionality but considering the architecture and the times that were coming, it was important to find another solution. That is why the work on OPC-UA began. The new and cross-platform capable architecture, OPC-UA, has all OPC functionality, pluss additions, and is backward compatible with OPC, which was an important factor for Statens Vegvesen.

OPC-UA has been developed based on missing factors from the older OPC specifications.

Future important factors: [39]

- Microsoft's COM and DCOM, the foundations of earlier OPC specifications, are now officially legacy technologies.
- Web services now offer the primary mechanism for data transport between computers (and also provide a better option for communications with plant-floor devices).
- Earlier OPC specifications failed to provide a single coherent data model - e.g. the Data Access item hierarchy was totally disjoint from that offered by Alarms and Events. This needs to be a focus.
- Backwards compatibility with earlier OPC specifications is key to acceptance of any new standard.

The old legacy technology for communication is not a very wide standard anymore, therefore OPC-UA have decided to use web services for primary transport. See section 2.4.2. The old OPC provided alot of the same functionality but in different disjunctive interface definitions, such as OPC Data Acces, OPC Alarms and Events, OPC Commands and so on. Now OPC-UA unifies these OPC definitions into one

model and providing a common interface. See section 2.4.4. Regarding the earlier official backend OPC technologies, which now is legacy, the focus have moved over to providing multiple implementations, 2.4.3. Information regarding the following sections strains from the OPC-UA specifications[1]

2.4.1 What is OPC-UA

OPC-UA is a way to standardize communication towards hardware, making it easier for users to abstract the hardware handling. The following text is an explanation from Simone Massaro regarding OPC-UA and how it affects this situation:

”The information carried within an object is far richer than the information carried with simple row data. In a typical automation application, you rarely wish to analyse single, isolated row data. Its far more interesting to analyse the data in terms of its relationship with other data, and in terms of the operation that can be performed.

Any real-life object carries a tremendous amount of information within it. For example, when thinking in terms of objects, the information carried by a ”boiler” object is far superior to the simple combination of individual row data for pressure and temperature. A physical boiler is an object that we can physically interact with by turning it off or on, by changing a reference temperature setpoint or by analysing how a change of a parameter affects the others. This information is logically grouped and must be analysed all together.

In software terms, an object is a collection of properties (temperature, pressure), methods (turn off, turn on) and events (temperature is too high, pressure is too low). Objects are organized in hierarchies in such a way that an object can contain simpler smaller objects as properties (the valve of a boiler can, itself, be an object that exposes properties, methods and events). When thinking in these terms, its clear how beneficial it would be to map the data of the factory floor into a hierarchy of objects.” [58] Exactly what OPC-UA has done.

It is a well structured information model which is created by address spaces. The address spaces is defined by a structure of elements which consists of Objects, Types and Views. The model structure is defined hieratic but is a so called full mesh network which allows for cross referencing to other parts of the current hierarchy 2.11. [55, 46]

When using the SDK from Prosys OPC, and creating a server, there are some default objects which reside within the created server. This is based on OPC-UA

[1]OPC-UA specifications website: http://www.opcfoundation.org/default.aspx/uaspecdownloads.asp?MID=Developers

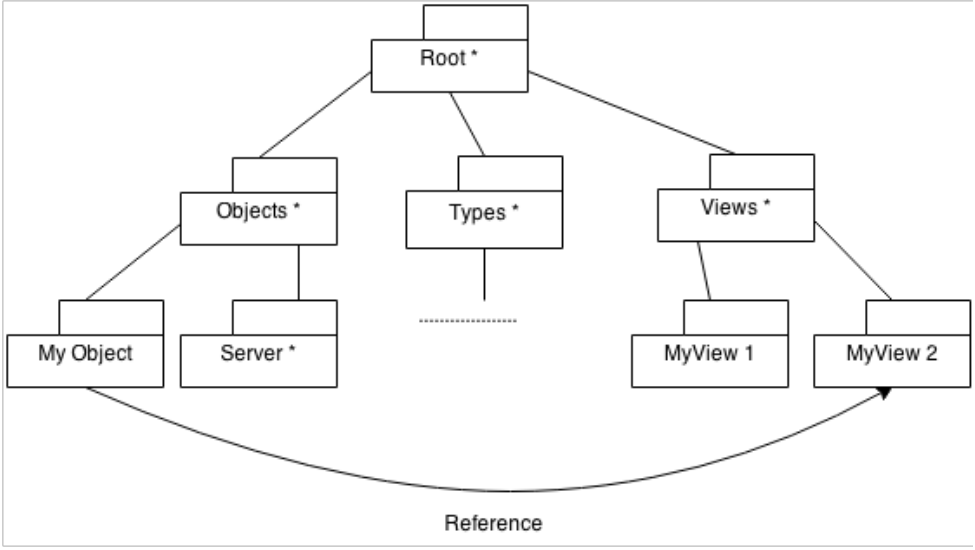

Figure 2.11: OPC-UA hierarchy presented with reference

specifications. The objects which is created within the server is noted by an * in fig 2.11. The first four objects are mainly folders which have its own node manager, 2.4.8. The last default implementation is a server object, which represents the current OPC-UA server with its properties and methods. This allows for controlling and monitoring the server. There are already multiple functions and listeners implemented which aid in these actions.

The last three objects are created by the user. The *MyObject* can be whatever, a boiler, a sensor, a light, with its properties, alarms & events and methods. The *MyView* objects are a way of mapping specified OPC-UA objectmodel items into one object, see section 2.4.10.

2.4.2 Protocols

This is a big improvement for the OPC-UA standard compared to old OPC. Now OPC-UA supports two protocols, Transmission Controll Protocol (TCP) and Web Services, which is only recognizable to the user by looking at the Uniform Resource Locator (URL). OPC-UA is totally transparent to the API. The two protocols have their differences and exist for a reason, for example the TCP has best performance, lowest resource consumption and also use the arbitrarily TCP port for communication which means that it is better suited for tunnelling and enablement through firewalls.

As for Web Services, Simple Object Access Protocol (SOAP), it is better suited

for using with environments such as java and .NET and is firewall friendly because its using HTTP/HTTPS ports. [57, 69]

2.4.3 Implementations

As for now there exist some commercialized implementations of the specifications in both java and .NET. Companies such as Prosys OPC actually provide SDK for pretty much all languages, .NET, java and C/C++. They also deliver a client for the Android OS. Other companies providing commercialised implementations are Ignition, Matrikon or Softing. Next to commercialised are open source libraries. These also exist around the internet, such as opcua4j(java), OpenOpcUA(C/C++), Unified Automation(C++,ANSI C, Java) and OPC Foundation. But OPC Foundation may need membership to download the desired code.

Because all of these different implementations exists it allows for a much widespread cross-platform hardware handling. Now that java exists, it can run anywhere a java program can run. Opening plenty of opportunities. The same goes for the C and C++ implementations which basically runs on any computer architecture. [50, 51]

2.4.4 OPC-UA Object model

The object model is what makes up an object in the address space. An object can be pretty much anything within the hierarchy 2.11, but it will all have its own properties, variables, references, methods, alarms & events. The following figure,2.12, compares the new UA object model and its properties to the old OPC disjunctive interfaces and displays how everything is now combined into one object model compared to the the older OPC.

This is one of the most important functional units within the address space [49]

2.4.5 OPC-UA Node Model

This node model is closely interconnected with the object model. Because as mentioned objects and their information is what the server provides to the client in the address space. And the way that objects and components is defined within the server space, is through a node model which is described by attributes and references which can be connected in a mesh network. [55]

The attribute of the node model describes the node and has something called a *Node Class*. This defines what class this node is representing, which can be object, variable, method and so on. Read more at [49, 26, 27]

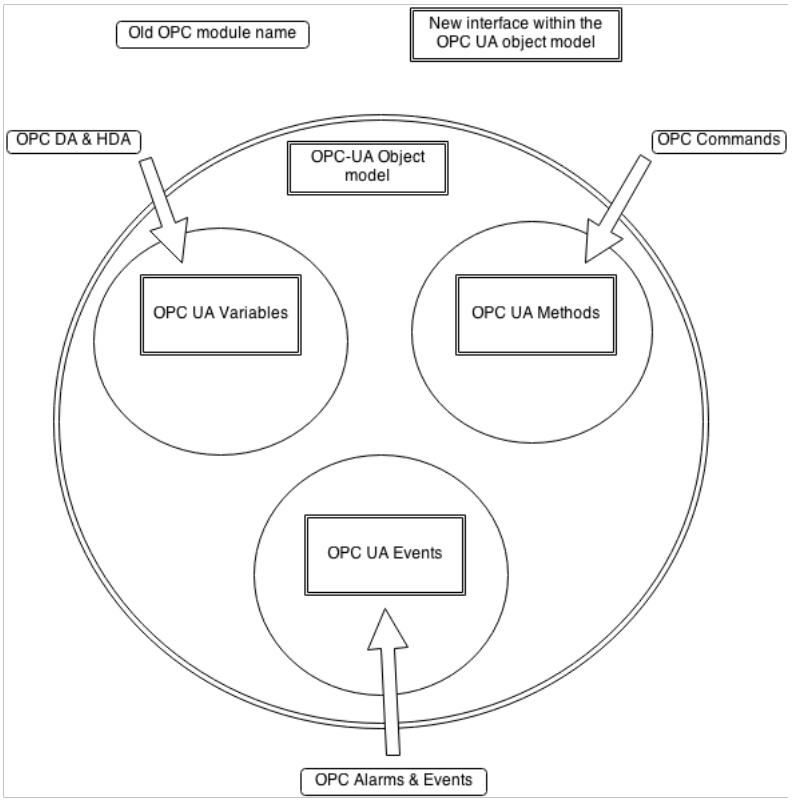

Figure 2.12: OPC-UA Object model vs old OPC modules

2.4.6 OPC-UA server

OPC-UA has defined their connection stack as server and client. Where the OPC-UA server is the object which holds the address space and models, and also pass along the events and alarms which might happen during runtime. The server is what usually is connected to the hardware and holds the current datamodel and address space for that hardware. The server can of course have multiple address spaces meaning that multiple hardware can have their data models reside in the same OPC-UA server. This approach is something this thesis will try out. The server is also responsible for the history access of the readings which has happened between the server and component.

2.4.7 OPC-UA client

The client on other hand is the component which has been designed and implemented to browse the address space of an OPC-UA server. The client can be viewed as a

middle man for data- collection and -browsing. For example a company has multiple sensors which is placed in the terrain collecting valuable information. The client on the other hand resides at the company base and is connected to the servers receiving updated information and has the opportunity to browse the current sensor remotely. During this scenario the client connects to multiple servers and have access to their current address space, events and alarms.

This thesis is going to use OPC-UA client in a particular way. It will be used as an object for looking up namespaces/addresspaces which belong to different hardware connected to the ITS station. This is because an OPC-UA client is especially designed for browsing and data-listening.

2.4.8 OPC-UA NodeManager

The OPC-UA *NodeManager* is the interface which allows for browsing and managing the address space. As mentioned earlier each default object on server instantiation has its own node manager. Through that node manager, it is possible to manage all children nodes. But this is not optimal, so it may be clever to generate node managers for different bigger objects. Each node manager is created with a name space to identify that particular node manager. Which brings us to the next section.

2.4.9 OPC-UA address space & Name Space

These two terms can be very similar and might be used somewhat interchangeably. The address space actually represents the whole server address space. Including all the name spaces, objects and nodes which exists within the server. While a name space is a string identifier passed a long when creating a node manager. So if one want to differentiate different spaces within the server, one have to map up name spaces or use OPC-UA views.

2.4.10 OPC-UA View

Views are a way to define an information model which is restricted to only the objects and values the user chooses. It is a very unified denomination within the software development industry. Views are used in multiple techniques, like Model view controller (MVC), or defined views in a Couch DB [2]. It means the same everywhere, a restricted data set. It is also how its used here in OPC-UA.

As specified earlier the information gathered in the latter sections strain from the OPC Fountdation specifications [2]

[2]OPC-UA specifications website: http://www.opcfoundation.org/default.aspx/uaspecdownloads.asp?MID=Developers

Chapter 3
Literature and related work

"If literature isn't everything, it's not worth a single hour of someone's trouble."

— Jean-Paul Sartre

As mentioned in the introduction the literature basically evolves around specifications and documentation for the possible features of the technologies chosen. OSGi, OPC-UA, RB [24, 43, 32]. The reason for this is that related work mainly focus on the high-level architecture, meaning that they dont go down into the technical spesifications. This thesis on the other hand will focus on the low-level architecture, and as mentioned before, SINTEF is currently writing a high-level architecture spesification for Statens Vegvesen. There is also some literature on how to create a dynamic environment, robustness and upgrading using OSGi.

Connecting these three technologies has never been done before, and because of this, there is an restriction on how many papers that exists. That is why the focus on specifications and documentations has been so high. But Statens Vegvesen has already been doing some work with some of these technologies separately. 3.3

3.1 OSGi

Its an advanced topic which could have been written about in multiple papers depending on what angle is chosen. But since the research questions focus on robustness,upgrading and expansion it has been important to look into some information regarding these focus areas.

3.1.1 Robust architecture OSGi

When it comes to robustness, which is elaborated in perspective to the technolgies in, 5.2, it basically means that things should work. That is also why OSGi is a usefull framework for handling robustness. Why, will be explained more detailed in section 5.2, but here are some references discussing this particular subject. [56, 72]

The first citation is an OSGi community event from 2013 talking about robustness within collaborative services using OSGi. It brings up some subjects that has been mentioned in this thesis and confirms some important views. The second citation is an article regarding robustness using OSGi and how to handle hardware exceptions and unexpected exceptions at runtime in a proper way.

3.1.2 OSGi and update/expansion handling

Updating is an important feature for all applications and systems. There might be new features or existing problems which should be handled. This is covered more extensivly in 5.3. OSGi has a dynamical environment, because of this it expedites the feature of upgrading. But one important factor when updating bundles in OSGi is having a focus on the states. Three guys from University of Shanghai wrote a paper about "ASM-based Model of Dynamic service update in OSGi". [74] This provides ideas regarding RB, which is using an ESM. RB might be able to be used as an OSGi updating platform, or just build some smart blocks focusing on updating. This should be for a future paper. But its food for thought.

3.1.3 Dynamic OSGi architecture

This field is a very interesting topic because its an ongoing debate on what is the best solution and how this can be done properly. But of course it is very dependent on what the context and environment is. Since this thesis focus on telematics and ITS perspective of dynamic architecture, there have been some interesting articles confirming some of the thoughts this thesis brings to light.

The article ”OSGi Based Service Infrastructure for Context Aware Automotive Telematics” [75] mentions architectural solutions of problems which Statens Vegvesen have proclaimed as a functional and technical requirement in their document [67]. Referring specifically to their vehicle ITS sub-system. This article from Daqing Zhang and Xiao Hang Wang also confirms our OSGi thoughts of having bundles covering each connection interface and placing a model, an OPC-UA model to be specific, on top of that specific hardware interface. They handle their context awareness using Web Ontology Language (OWL). But this thesis will endeavour on handling the different types of hardware using OPC-UA models and events.

3.2 OPC-UA

OPC-UA is a very widespread technology which is used in multiple domains. So finding articles or papers within this specific domain has been difficult. For now this thesis can only relate to the work Statens Vegvesen have done with OPC-UA. As for literature there have been mostly specifications and documentation.

Most of the work that is beeing done with OPC-UA can be related to some particular fields within this theis. Such as device integration [45] or specifically address space research [49]. The last article gives some good ideas regarding thoughts about this thesis suggested architecture evolving OPC-UA and hardware object model. There can of course be found plenty of work regarding OPC-UA but nothing particularly more specific.

3.3 Statens Vegvesen

Statens Vegvesen have already been working a lot with OPC-UA in a project were they focus on data collection from the Norwegian road network. [65] This is a project which paved the way for another project which is now in the production state and is being developed as we speak. This project can be related to the control station this thesis decided to not focus on, which is mentioned in the introduction, chapter 1. They have currently outsourced the production of roadside equipment to other producers. But they have given a sense of OPC-UA data model standard they wish to connect to.

So Statens Vegvesen is not unfamiliar with the technology and OPC-UA is something they are going to use in the future years. They are already using OPC for certain tunnel control systems, and want OPC-UA to graduatly replace this. That is why backward compatibility was so important.

OSGi has been used in a research project which SINTEF have been engaged in. It was not very widely used, but they have tried it out and have some sense of how it works. The project was called "Test-side Norway" or "Wise Car" [68]. The implementation was Q-Frees[1] responsibility with Runar Søråsen.

Reactive Blocks from Bitreactive[30] has never been used in an existing project and is a new introduction to Statens Vegvesen. It was introduced because ITEM NTNU has a close contact with Statens Vegvesen and Bitreactive was founded based on technology which was created at ITEM NTNU. It is desirable to have a close cooperation.

3.4 ITS focus around the world

ITS stations is not an unfamiliar term outside Norway. It a wide spread opinion that traffic and travel need to combine information and communication technologies with the current transport picture today. [10] But most work has been high level architecture. Not a lot of specific technical work.

[1] Q-Free homepage: http://www.q-free.com/

3.4.1 EU

As European Commission (EC) states, "The main innovation is the integration of existing technologies to create new services".[10] By this its understood that its not necessary to invent the wheel all over again, just create some smart connections to be able to provide new services and receive new information. The state of the current ITS is limited and fragmented were European Union (EU) have acted to bring the state into becoming EU-wide. [10]

3.4.2 US

US have developed a national ITS architecture which is currently at version 7. [20] This can be compared to the specifications which SINTEF is currently writing for Statens Vegvesen. [67]

The architecture defined by the US is a very extensive specification so this thesis wont go into any detail about this, but it is a very high level architecture specification, compared to thesis's low-level architecture.

3.4.3 China

Its not only the western world who focus on ITS systems but also Beijing have had ITS in focus. [71]. This is not directly related to this thesis but its a higher level article about ITS in Beijing.

Chapter 4
Methodology

"I think you can have a ridiculously enormous and complex data set, but if you have the right tools and methodology then it's not a problem."

— Aaron Koblin

4.1 General

This thesis focus on creating an architecture which is robust, expandable using three spesific thecnologies.

How the work has been structured during the process of this thesis has mostly been a literature study of the three technologies and their possibilities. As well as trying to define an architecture based on the existing knowledge, and experimenting with the chosen technologies. It has been an iterative process with defined focus points for each iteration. This methodology can be compared to design science methodology.[48]. But the guidelines does not match perfectly for this type of process. For example it is hard to follow the guideline which involves research rigor. Since this is the first time it has been done and there is not much relevant work on this subject.

Second it will not be possible to design a viable artifact, since it was to much technological connections and designing to be done for a product to be delivered to a test station in time. This will be mentioned in further work, 10.4. Because some of these guidelines does not match the designe scienece methodology, this thesis will set some specific guidelines for the methodology used here, 4.2.

The first iteration unfolded how the three technologies may be interconnected and used together. And the following iterations have targeted different focus points in regards to our research questions, with more advanced situations as the iterations have progressed.

An early architectural draft was made to set a sense of direction for the assignment and important factors that should be considered. See Figure 4.1 When looking back

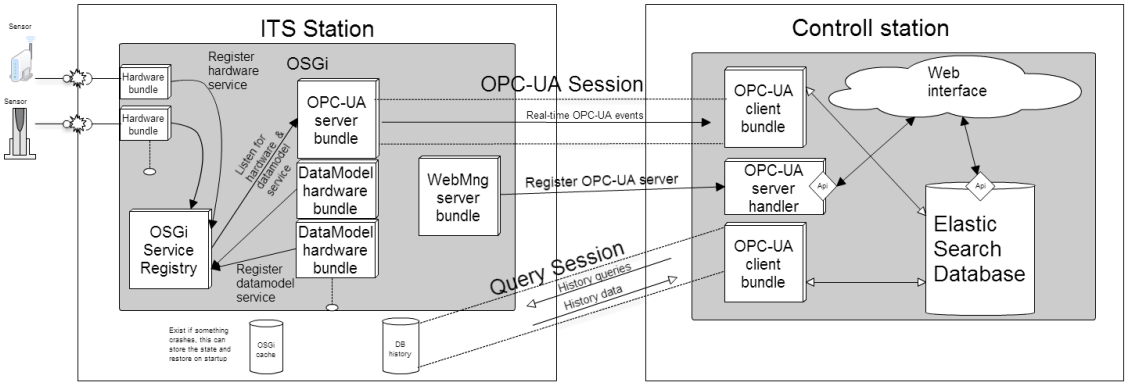

Figure 4.1: First draft architecture

at this architecture its easy to see that the knowledge of the different technologies was not profound. Refer to architecture figure 6.4 to see where it was headed after a broader technology study.

This architecture conveys a simplistic view on how OPC-UA and OSGi could work together, but does not mention anything about RB. That is because RB is a way of designing your java application. This architecture also provides an insight in how the control station could be designed, but this is not in the scope of the thesis as mentioned in the introduction.

4.2 Method guidelines

According to our methodology, which is iterative experimentation with given focus points and a pre technological literature study, it makes sense to have a some guidelines on how to work. These are based on discussion with study mates, supervisor and logical sense:

Nr.1 Exploration of the given information from Statens Vegvens and SINTEF is important to get a better understanding of ITS

Nr.2 Study the technologies which should be used to get a sense of possibilites

Nr.3 Define some given focus points to be explored

➥ **Iteration start**

Nr.4 Begin an iterative process with experimenting on the simplest focus point of the ones left

Nr.5 End iteration with feedback on results from the current experiment

Nr.6 Rethink the future focus points, add new ones if necessary

Nr.7 Write about the iteration

➦ **Iteration end, redo**

This can go on and on until the whole assignment has been solved. But there is a time restriction so we will see how many iteration it is possible to finish during this thesis.

4.3 Literature study

The structure of the literature for the thesis has been mostly specifications and documentations from the three technologies employed in this thesis. They can all be found on their respective sites. [62, 30, 61] The documents had to be studied to give a good impression of what the technological possibilities were. Knowing these possibilities would give the opportunity to create the most intelligent architecture that would be possible with the features provided. This would also give an advantage in covering the questions that 1.3 mentions.

Specifically for OSGi, the writer has written a pre thesis assignment on "Modularity and Lifecycle of OSGi Applications" [70] which builds the foundation of the interest towards OSGi.

Statens Vegvesen also have literature that was worth reading to get the necessary background information for the current ITS project. Such as [65, 67]. They also provided with a sample application which helped a lot in the understanding during the initial development.[1]

By studying what other companies have done throughout Europe gave insight in how the future ITS stations should work, giving this thesis some food for thought. Most of this information could be retrieved through the European Commission web pages regarding transport and ITS system. [10] If other sources is used they will be cited at that place.

[1] Can be obtained by contacting the people involved with this thesis from SINTEF IKT og samfunn or Statens Vegvesen ITS Trondheim

4.4 Development tools and technology

4.4.1 Eclipse

As for technologies, these are pretty much covered in the next chapter and mentioned considerably through the thesis. But as for development tools it is an easy setup. The setup is dependent of Eclipse 3.7[42], which can run in any Operation system (OS) that support java. But to setup the eclipse is sort of a process. Browse down to the appendix to read about the eclipse setup. Appendix C.1

4.4.2 GitHub

There were written some lines of code and designed some building blocks during the thesis which is available on GitHub.[2] To be able to get access to the code you have to contact the writer to get approval on downloading the code. Please refer appendix C.2 for more detailed instructions on how to retrieve and import project.

4.4.3 OPC-UA tools

There were also used some OPC-UA tools for trying out the different modules during development. The OPC-UA client from Unified Automation - UaExpert[28], was an important tool when developing the server and the different datamodels. It was necessary to have an adequate application which could browse the address space, change variables and call methods. If this tool was not used, it would be difficult to get a proper client up and running, because the uncertainty of the servers implementation was to big. Therefore one could validate with this client tool from Unified Automation.

It was also important to have a test server in an early stage to understand how the whole OPC-UA environment worked. This was provided by the Prosys SDK. There are a bunch of freeware servers available.

[2]The repo is private so contact the writer for access, through the information on his GitHub account[21]

Chapter 5
Iteration 0: Technology study

"It has become appallingly obvious that our technology has exceeded our humanity."

— Albert Einstein

5.1 Intro

This thesis had focus on the three research questions in section 1.3. They involve robustness, upgrading and expanding an application deployed to a current ITS station. These are important terms considering that the ITS stations are often placed far out in the field, near roads, and would be a hassle if technicians had to go out to the stations every time something was wrong or Statens Vegvesen wanted to upgrade their application. Also in these days electronics and software are an important aspect of the daily lives and users should be able to trust the applications being deployed. That is some reasons why these terms are important for Statens Vegvesen, and this thesis will try to present some opportunities regarding an ITS station.

This iteration is called the iteration 0, because it does not follow our iteration steps completely, but it was a process of feedback and pondering over the future rundown. It incorporates the first step of our methodology guidelines, which was study the technologies. Through out this iteration they have been researched and examined the opportunities available . The technologies have been put into context of each term. The three mentioned categories will be considered, and for each category it will be a small discussion around what kind of opportunities each technology provides. After this introduction the thesis will portray some iterations with different focus points explaining the bumps and solutions towards a working application and proper architecture.

5.2 Robustness

Robust definition

Strongly formed or constructed; capable of performing without failure under a wide range of conditions [13]

As mentioned above its important for Statens Vegvesen to be able to trust the application being deployed along the road. This equipment and application provides reliability to traffic and serves as an important data collector. As they have stated in their specification [67]:

Statens Vegvesen spesification

"The Transportation Network Information Manager is responsible for that information about the Transportation Network, in this case the road network, is established, verified, updated and made available to other functions. This includes both static and dynamic information about geometry, regulations, condition and deviations. For example:

- Conditions caused by unplanned/unforeseen events (e.g. due to weather or accidents)
- Environmental conditions
- Transportation Network Equipment status (e.g. signaling and communication equipment)

The transportation Infrastructure Manager will pertly receive this information from the Roadside ITS stations."

In hindsight of this specification, if the application were to be down every other day for a couple of hours, it would loose important data and information, and might even cause car accidents.

That is not a desirable scenario. So considering that OSGi is dynamic and the ITS stations will be working with multiple hardware which may fail at any point, its important that the rest of the application will not crash on a simple hardware error. Its important to remove any single point of failure and have a backup solution for error scenarios that is known. This is achievable through a modular and SOA

architecture which will be provided using OSGi.

This thesis have considered an architecture which provides the main application an opportunity to handle any loss of services generated by outer circumstances. Meaning that this architecture will try to separate the necessary modules to create an environment for the application be able to respond to edge cases. The logic behind this response will be generated and visualized with RB.

Of course, not all cases can be handled, some will be forgotten, that is why we have brought in the term of upgrading, so when experiencing an unhandled egde case, Statens Vegvesen can quickly deploy a new version to fix the situation.

5.2.1 OSGi

By using OSGi, which provides the opportunity to have modules with their own lifecycle. This architecture can benefit by stopping the current bundle causing the errors, and therefore letting the application live on without this failing module. This will of course have to be handled defensively in the code which is one of the most important OSGi code principals. Also this is why RB have been included in this thesis. Visualizing the specific logic of each application module within the ITS station.

5.2.2 Reactive Blocks

The logic of an application can be difficult to visualize with pure java code, it can become messy and usually only the programmers who are participating in the project understand the code. Therefore its difficult for programmers to explain the logic of what has been created, especially towards users from other domains. Using RB, participants can easily study the architecture visualization and understand the flow of the code. Because of RB, different domains may together discuss and provide an upgrade to an edge case which arise.

5.2.3 OPC-UA

OPC-UA is also introducing a solution to the robustness problem. It provides an opportunity which is more indirect compared to the other two technologies. For example, if some hardware breaks down, or some hardware turns out to be useless, Statens Vegvesen have to change this into something that provides the quality they desire. Since the data model has been standardized it will not make any differences within the application logic what hardware is recording data in the specific situation. This is because there will be a data model with the same APIs as the older hardware. So deploying a new hardware will become a whole lot easier.

5.3 Upgrading

Upgrade definition

An occurrence in which one thing is replaced by something better, newer, more valuable, etc. [14]

Statens Vegvesen will experience unforseen happenings where some can be solved programmatically and some have to be fixed physically. Therefore its important to have the opportunity to be able to upgrade and deploy new versions of particular modules without any considerable difficulties. By difficulties it involves challenging to deploy, the need to take down the whole system and burdensome to manage.

5.3.1 OSGi

Because OSGi provides a lifecycle through their specification, it gives the architect and programmer an opportunity to create better controlled applications. Since there is a lifecycle, bundles, or JARs as known to java developers, can be deployed at any time of the day, given that the remote connection is created. Meaning that we can upgrade any individual bundle providing specific functionality, at any given time. This must be handled in the code and thought about when developing, but OSGi provides the opportunity.

During the upgrading process OSGi use smart algorithms which can be reviewed in [70] page 32 and onwards. As for easy deployment there exists a great deal of frameworks from OSGi assisting the project in deploying new bundles. This is not in the scope of the thesis, but here are some frameworks which could assist: Karaf [6], Apache Felix OBR [5]. There are of course possibilites to develop the deployment framework on your own using the OSGi spesifications.

Its also worth mentioning that OSGi provides advanced dependency handling within the container. Giving the application the possibility of multiple versions of the same bundle residing within the application. This can arrange for a very granular upgrade process to make sure that everything works between each change.

5.3.2 Reactive Blocks

Reactive blocks does not provide any opportunities for the upgrade process other than that users will always have a visual control over the flow of the application. During the related work it was mentioned an article regarding upgrading OSGi services with an abstract state machine, 3.1. This can be connected to RB in regards

to that RB does have a state machine implemented. Trying to control the OSGi upgrade process using RB. But that is far fetched and can not be related to this thesis.

5.3.3 OPC-UA

OPC-UA brings the benefits of standardization to the table. As we mentioned earlier in the robust section 5.2, OPC-UA provides robustness indirectly through the standardization they create. The same goes for upgrading the application. Because when adding functionality the users have abstracted the hardware by using OPC-UA and only work towards an API. Meaning that upgrading the application becomes easier because users only have to worry about the logic itself and not what kind of hardware is attached.

5.4 Expanding

Upgrade definition

To increase in size, range, or amount : to become bigger, to make (something) bigger [15]

Since this project is defined as a development/research project, its hard to visualize all its possibilities and its future use-cases. There are of course already defined some use-cases in the document from Statens Vegvesen [67]. But from experience its difficult to foresee all possible applications to a research project. The different applications also depends substantially on the current location or area of the ITS stations. For example an ITS station residing within a big city might have a great deal of traffic to handle, and even want to communicate with the cars to be able to redirect the traffic. This will not be the case out on Dovre, where the climate and weather is more interesting. But things will change and other variables can become interesting.

Therefore it is important to considered the prospect of expanding the application in the future. This should also be without the same difficulties mentioned in the upgrade section.

5.4.1 OSGi

Expanding with OSGi works pretty much the same way illustrated in upgrade. Because expanding and upgrading are pretty much the same activity. Its all about deploying a new bundle which have its dependencies and provides some important

functionality. But the factor that makes a difference in expanding, is that users probably will deploy multiple bundles, or to be more abstract, an application which is created by multiple bundles. To do this in a proper manor is something that have to be considered.

Frameworks to relieve the process of expanding already exists within the OSGi ECO-system. One of these specifications are called OSGi subsystem. [19]. This can also be handled through Karaf using something called a feature file. This provides the opportunity to declare lists of collection of bundles that should be deployed together. [7, 11]

5.4.2 Reactive Blocks

Reactive Blocks (RB) are an important factor in expanding the ITS station. Because expanding an ITS station implies that it will be a new application deployed into the existing station. Since the application is designed using java and RB, expanding is dependent of RB. But its not part of the process of significantly simplifying the way of expanding, that is OSGi's job.

But it allows for control and overview over the application logic and be able to get input from other domains as mentioned before.

5.4.3 OPC-UA

Open Platform Communications Unified Architecture (OPC-UA) is an important technology to wield in the application to easily expand and provide new data to the control station at Statens Vegvesen. Because OPC-UA is the communication protocol its in this domain were OPC-UA can shine. That is because of the abstraction of hardware, and the opportunity of having one connection point within each ITS station. Meaning the OPC-UA server.

The OPC-UA server will provide the interfaces for interested parties to connect and be able to retrieve the data which the ITS stations gather. When Statens Vegvesen decide they want to expand they only need to deploy a data model bundle which dynamically injects itself to the existing OPC-UA server address space. Then its possible to handle the hardware which this datamodel is connected to.

Therefore OPC-UA provides the necessities to have a smooth expansion of new applications and hardware.

5.5 Summary and concluding remarks

By recurring over the the terms influencing the research questions[1.2], it becomes clear that the technologies chosen for this task will in some way contribute to a more elegant architecture which can handle most of the different cases. Its also evident that these terms are important for an architecture to be properly tailored.

But in regard to the questions its now possible to get a feel on how this thesis may provide satisfactory solutions to the questions. They will be utterly discussed through the iteration chapters that follow.

Chapter 6
Iteration 1: Connection and pair-ability

"The ultimate connection is when you are connected to the creator of the universe."

— Joel Osteen

Chapter 4 made a remark on how this thesis have been conducted. That was utterly through experimenting with thechnology through iterations where there were a new focus for each iteration. Taking care of use-cases that has been mentioned either by Statens Vegvesen or that has been uncovered during the research phase of this thesis.

This second iteration was a process of the initial connection and analysed the possibility to get all technologies to work together as a whole application and covering our research questions. Through this iteration there were a great deal of "guidelines" that was discovered for future development.

6.1 Initial build

Through the second iteration it was important to figure out all the different quirks that each technology would provide and how this would fit together properly. But since it is a common platform where it all runs, java with OSGi, the connection was not a big problem. But it is important to get all the JARs files OSGified. Meaning that the dependencies which did not have the necessary MANIFEST.MF headers had to be converted. Since Prosys only provided access to the JARs, and not any source code, the solution was to import the file into BND Tools and let that take care of the recompiling. Se figure 6.1 It follows a really simple wizard of remaking the JAR to a bundle.

It was important to notice all the dependencies from Prosys, and include them in the OSGi build. They were provided through their sample client and sample server. These JARs was already OSGified from the vendor of the libraries, so it did not create any problems. But this is a hassel if working with a great deal of dependencies on third party libraries.

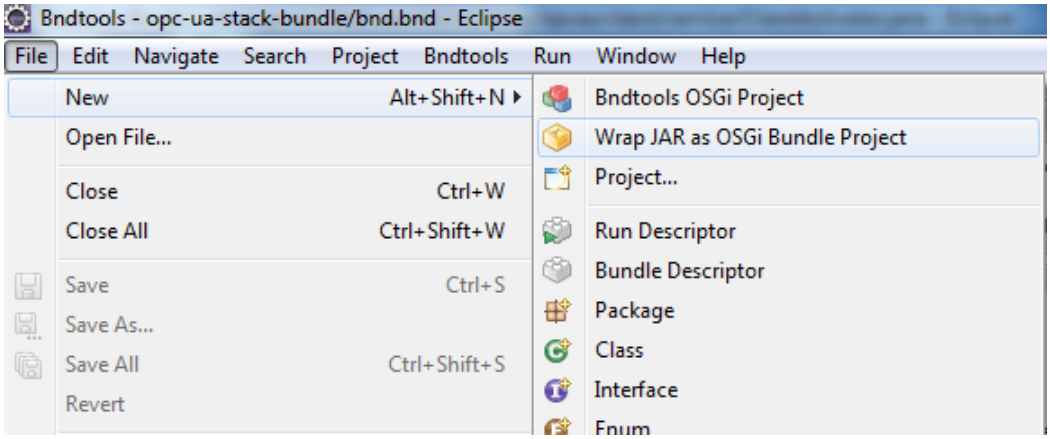

Figure 6.1: Wrap JAR as OSGi bundle

All the dependencies was easy to include in the OSGi enviroment and because RB now have added support to implement OSGi bundles out of the designed applications, they were also easy to include.

But its worth mentioning that RB just started supporting OSGi and is in a development phase to figure out how much of the OSGi spesifications they want to support. Frank Alexander Kraemer have stated that their goal is to try to abstract the thought of OSGi for the user and let the user rather focus on the logic. That is also part of why RB is included within this thesis. To work with this abstraction level and get more focus over towards the logic.

6.2 Guidelines for future development

As mentioned earlier it emerged some guidelines, through out the iterations, on how to develop this application further. By guidelines it is meant that it was smart ways to develop this application. Some of the guidelines were for example, how low level should the user design RB blocks, how should the plain OSGi bundles be designed, how can the application have a general API towards services and how shall the control of the application be designed.

6.2.1 How low level should the Reactive Blocks be

This was an issue because as mentioned before, Bitreactive [30] wants to abstract parts of OSGi from the user to make it easier to develop applications. Also since RB OSGi support is in a developing phase, they are not completely cooperative with some functionality from OSGi which is desired for the application. So the decision landed on creating an application where pretty much every module in the environment

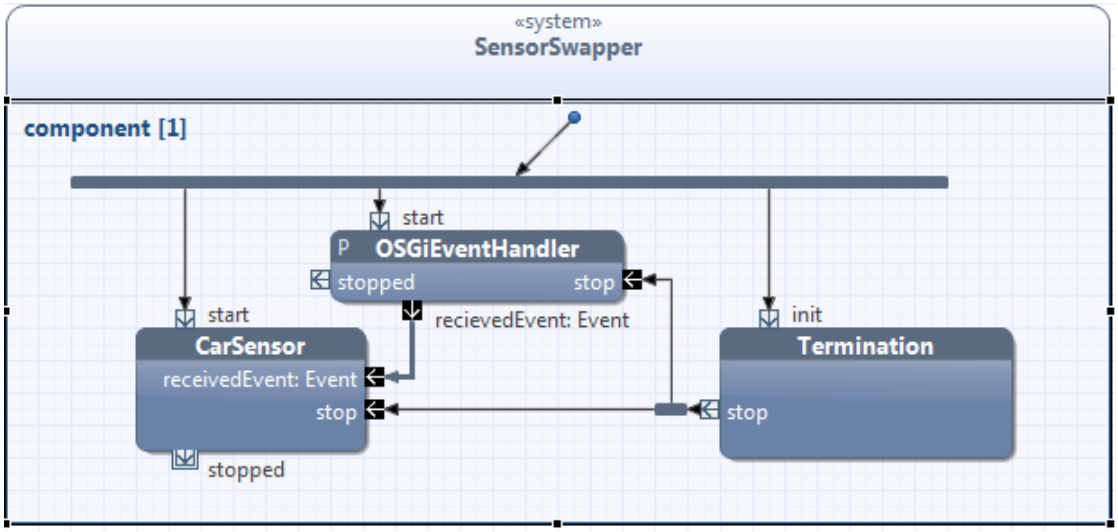

Figure 6.2: Simple first application which was created in chapter 7

provided a service for RB to be able to track. Through this the user could be able to visualize the event logic with RB and create an ITS station application, controlling and monitoring the different main modules residing within the ITS station, through services.

As for the logic for the specific applications, they would be created as their own system blocks, please refer to 2.3 to see what this is. Following this path of developing, the ITS application logic could be portrayed in their own isolated subsystem block, which is also its own bundle with a lifecycle. As you can see in figure 6.2, this is the logic of a simple sensor swapper example. And if you go into the *CarSensor* block, 6.3, there is even more logic there. This application will be utterly described in chapter 7.

For now the level of abstraction is mostly just for logic and visualizing events. For example the OPC-UA server itself is not implemented within RB. This was decided because it is an advanced task and would be very demanding, indicating that it would take too much time and not be in the scope of this thesis. It is also known that Bitreactive [30] are talking to Prosys, and something might happen there.

6.2.2 The what and how regarding the OSGi bundles

This is very coherent to the last section, because pretty much everything that is not a RB would be made as a clean OSGi bundles. So the what would be everything else that is not a RB. The difference is that there had to be some guidelines regarding

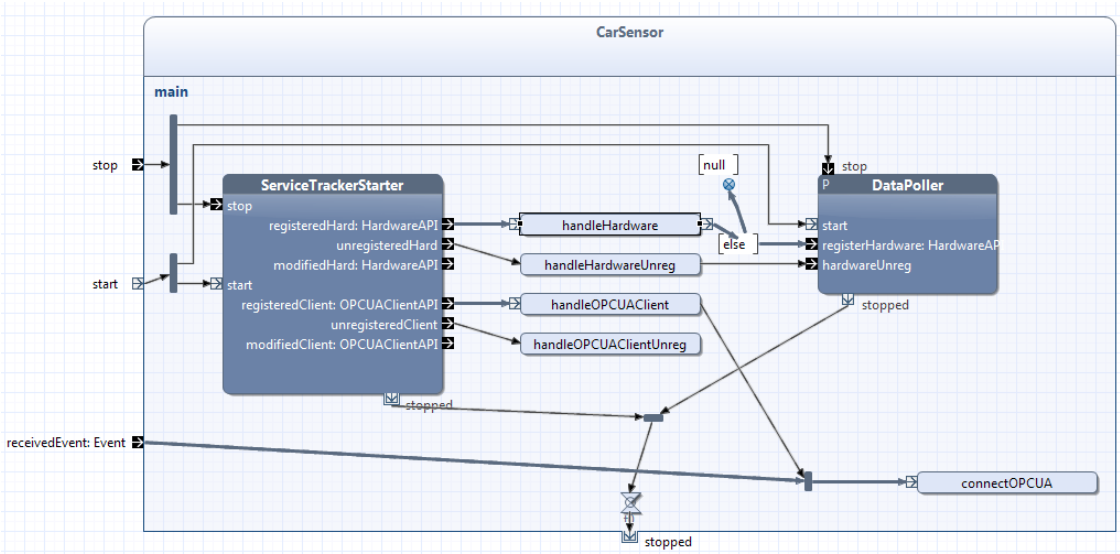

Figure 6.3: The logic whitin a *CarSensor*

how the clean OSGi bundles should be designed. What the focus should be.

This is of course very dependent to each area of application but it needed to be discussed to set a guideline towards the road for a clever modularization of the different types of system opportunities. Meaning for example how is the hardware connection to be designed, and how would the OPC-UA data model, OPC-UA server and OPC-UA client be designed.

Regarding the research questions 1.3, focusing on the design of the different clean bundles, may affect the opportunities to be able to update and expand the ITS station. Are these bundles not carefully considered, with the right services and APIs, the two mentioned terms will be difficult to achieve.

Hardware connection OSGi design

After this second iteration it was very clear on how the hardware connection would be designed. Each hardware that would be connected to the ITS station will have its own OSGi bundle. A driver bundle. This bundle will provide a service for retrieving data, as well as a service to be able to handle the hardware itself. For example if there were any exceptions, or a user wanted to stop this remotely or through OPC-UA.

This is the proposed design when creating a data model with OPC-UA. For each hardware, it is nessecary to have access to the raw hardware API. At some point the hardware has to be connected to the following OPC-UA data model.

This allows for a proper abstraction when considering the possibility to upgrade and expand. Also if the hardware fail, it will only be the hardware driver bundle which fail giving space for error handling. More about OPC-UA device integration [45, 46]

The OPC-UA server design

The decision fell on the task to not implement the OPC-UA server within RB, and rather make it as a clean bundle. This was done to save time and get it up and running as fast as possible to be able to proceed to the next iteration. This design isolates the OPC-UA server into its own bundle, with a service API to be able to control the address space dynamically. This API has not yet been properly defined, but that is the direction this thesis want to proceed. Controlling the address space during runtime is an important factor since hardware can dynamically come and go, and they will be provided with their own OPC-UA data model bundle. It is created based on example code from Statens Vegvesen[1], and java server SDK tutorial from Prosys [64, 60].

The OPC-UA client design

Iteration one did never touch the subject of a OPC-UA client. It focused mainly on getting a server up and running and using freewareclients[2] to make sure the OPC-UA server was up and running. Also to check if the test models had been implemented. But between iteration one and two it took place a meeting with Jo Skjermo. He is one of the OPC-UA specialists from SINTEF contracted to this project. He has been in charge of a couple of papers which concern OPC-UA in the thick of this project.

After the meeting with Jo Skjermo it was discussed how the architecture associated with OPC-UA would be. The result of these meetings was to go for a single OPC-UA server as mentioned before, were each hardware would have their own OPC-UA client associated to them. Rather than declaring a service for the OPC-UA server which would make it possible to browse the internal OPC-UA server address space and listen to events and data changes directly from the server. The reason for this was that the OPC-UA client is specifically designed to handle these scenarios. And even if in principal one should be able to have the event and data change information before anything is triggerd, it is not a suggested way to go. Jouni Aro, a Prosys OPC forum moderator, confirmed the thoughts for us. See conversation [9]

Another issue was that the OPC-UA client has different functionality which would support the browsing of an complex OPC-UA namespace better than if the server

[1] Can be obtained by contacting the people involved with this thesis from SINTEF IKT og samfunn or Statens Vegvesen ITS Trondheim

[2] Such as Prosys OPC-UA samle client, UaExpert client, OPC-Foundation Client Example

were to browse its own address space. Also since Statens Vegvesen have confirmed that its important for them to support backward compability to the older OPC, the OPC-UA client have facilitated the proper functionality to create such a gateway. See mail in appendix B.1. The gateway functionality of the OPC-UA client was also confirmed by Jouni Aro in [9]

The clients would be a clean OSGi bundle because it is an advanced part and it would be easier to just create a simple bundle per hardware to get it up and running. The problems experienced with this solution and how this was solved, is described in iteration 2, chapter 7.

The OPC-UA data model design

This has to be done programatically since there are no modelling tools that exists for java yet. It should also be created as a bundle and focus on the loose coupling to be able to update the data model at any point. This is because it is important to be able support expansion of the current model, as well as dynamic behaviour of hardware connected to the model.

6.2.3 General APIs

This was a discussion which arised early. Because at the current state of RB and what they supported through their service tracker, a general API was important to be able to handle dynamism. Implying that the module using the service has no recollection of the implementation behind the API. This results in loose coupling which is something we want to strive for. p.16 [70].

But after that RB started to support more advanced service trackers and the initial architecture started to change course towards using OPC-UA datamodels and clients, this did not become as important.

6.2.4 How the application shall be controlled

It has already been discussed how the application shall be controlled, through a master block which tracks the services necessary to have control over the main parts within the ITS station. By main parts it is understood that it involves the OPC-UA server, hardware and other important modules that should have a centralized control. The reason for this is to be able to uphold the preferred robustness which this thesis focuses on. The architecture of having a master and multiple slaves is a very familiar way of designating control. [1] This is not displayed in the application in a proper way, because i never became clear what kind of master slave architecture that would be necessary.

6.3 Unsolved Problems

After this iteration there were heaps of unsolved problems because there were no specific example which has been implemented. Some of the problems still are:

- ✗ Create an OPC-UA client implementation
- ✗ How to handle the OPC-UA client and hardware initiation relation
- ✗ How to provide the datamodel hardware connection
- ✗ How the APIs should look like
- ✗ Get OPC-UA events and methods running

The next to last issue is an evolving case meaning that it is hard to define the API in the beginning of the project as it will be more defined as time goes.

6.4 Summary and concluding remarks

This iteration was the start which brought up a lot of good questions and problems that had to be dealt with. It resulted in setting some guidelines for future development to be able to have an efficient way of coding. These guidelines were influenced by the research questions 1.3, earlier experience with OSGi and new involvement with OPC-UA. Following these guidelines will ensure that the terms of this thesis will be preserved.

After these findings there had to be some changes to the current proposed architecture in chapter 4. The changes, figure 6.4, convey that the focus of a central station has been removed and that the new decisions has taken effect.

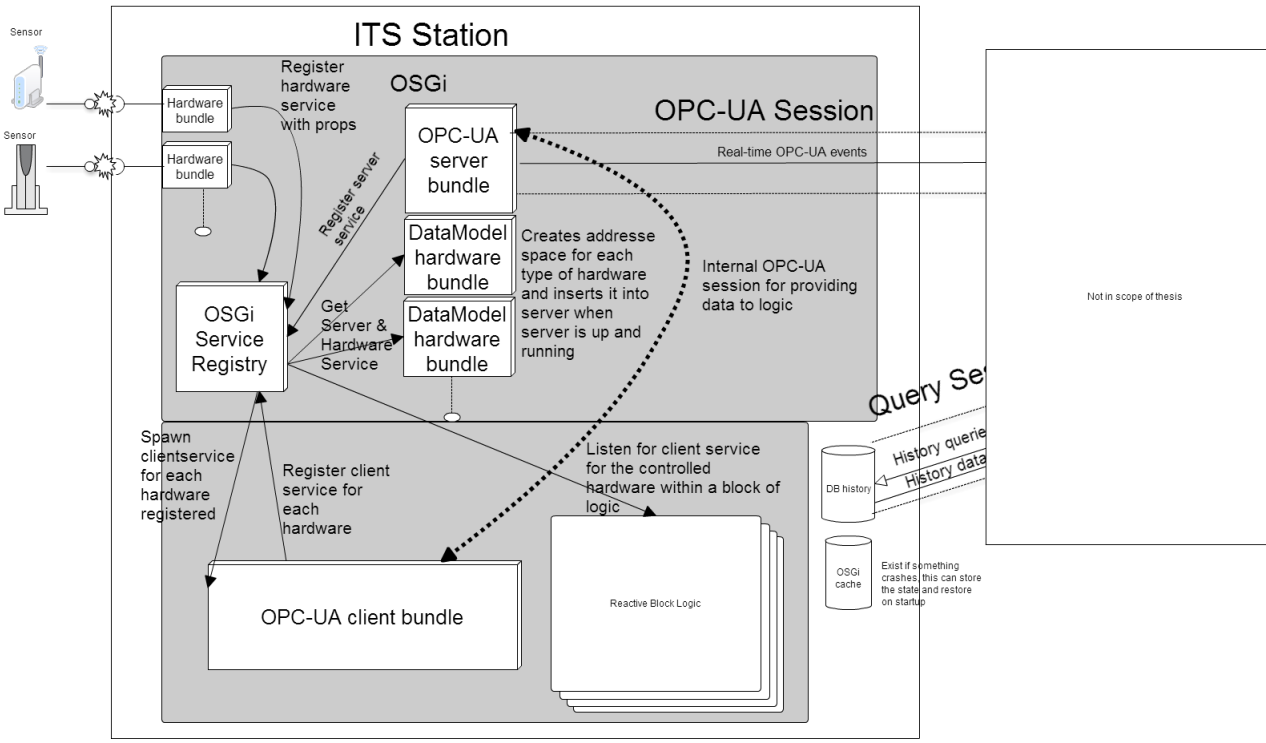

Figure 6.4: Version 2.0 of the architecture proposed in chapter 4

Chapter 7
Iteration 2: Implementation of simple swap example

"A good idea is about ten percent and implementation and hard work, and luck is 90 percent."

— Guy Kawasaki

After the second iteration it spurred some guidelines which would define parts of the future architecture. After these guidelines was set it was important to make sure that they measured up in a test application, regarding to communication, applicability and supervision.

This iteration focused on the swapping/upgrading of hardware and how that could be easily done, without the focus of an OPC-UA data model. It included the task of creating an OPC-UA client for each hardware registered. It is important to start with the small use-cases and build experience through practice.

7.1 Problems from last iteration

- ✗ Create a OPC-UA client implementation
- ✗ How to handle the OPC-UA client and hardware initiation relation
- ✗ How to provide the datamodel hardware connection
- ✗ How the APIs should look like
- ✗ Get OPC-UA events and methods running

7.2 Swap hardware application

The first proper application development was a simple RB application called *SensorSwapper*, trying out the simple use-case to swap an existing hardware when the application is running. Also try to keep it running without any exceptions or breakdowns. The bundles needed to create this demonstration was an OPC-UA server running in the OSGi container, the RB application bundle containing the

logical visualization and two hardware driver bundles implementing the standarized hardware API. This iteration was also going to try to connect an OPC-UA client to each registered hardware, which would indicate that a simple OPC-UA client bundle with its proper API, also had to be in the running OSGi environment.

7.2.1 OPC-UA server implementation

The OPC-UA server implementation follow the very basics standard implementation from a OPC-UA server tutorial [60] which was included in the SDK recieved from Prosys OPC [64], and the sample code provided from Statens Vegvesen[1]. This implementation is an OSGi bundle which also register a service API in the bundle activator. This is the simplest server that is possible to create but it servers up its own server object through the API so developers can have the opportunity to interact with the OPC-UA server.

Serving the server object through the API might not be the proper way of giving the opportunity to change a server property or add some address space, since having access to the server object provides a whole lot more operations. But it was currently the easiest solution for this iteration.

It was important to learn how to be able to add a custom address space to the server, to see how it could be done. It was not anything advanced, just a simple folder with some properties. SINTEF had provided a simple example project which was the source for address space experimentation[1].

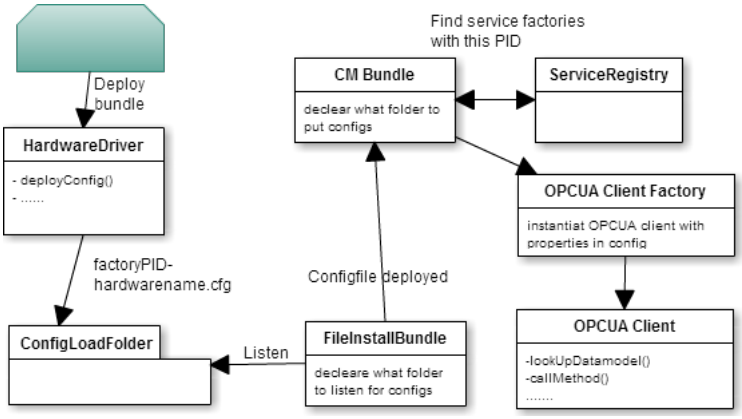

Figure 7.1: OPCUA client instantiation process

[1]Can be obtained by contacting the people involved with this thesis from "SINTEF IKT og samfunn" or Statens Vegvesen ITS Trondheim, see acknowledgements

7.2.2 OPC-UA client implementation

During the second iteration of this thesis it was never implemented an OPC-UA client bundle. Ref section 6.2.2's discussion. Therefore it was up to this iteration to handle the OPC-UA client implementation. Figure 7.1 portrays the planned instantiation process.

The implementation of the OPC-UA client was not very advanced, because at the beginning the goal was just to get it up and running, connect to a server and be instantiated by a plugged in hardware. It initially consisted of an activator which first of all registered a client service with a client api, see listing 7.1, and a simple implementaion following the very basics standard implementation from a OPC-UA client tutorial [59].

Code snippet 7.1: HardwareAPI interface

```
public interface OPCUAClientAPI {
    public void sayHello();
    public void connect();
    public void shutdown();
}
```

Unfortunately this was not enough to provide each hardware instance with is own OPC-UA client. Using just the service would result in sharing the same object and end up with a lot of troubles.

Therefore a solution which would provide each module, needing a client, with it's own instance of an OPC-UA client need to be created. The first option was an OSGi *ServiceFactory*. [18]

ServiceFactory

A *ServiceFactory* is pretty self explanatory, meaning that its a factory which provided service subscribers with a new instance of the object, without them knowing. The subscribers make a notion of wanting the specific service and will receive its own instance when it is available.

So the solution to this would be that each module would have its on *ServiceTracker*, tracking the OPC-UA client service. And when the OPC-UA client registers, the trackers would try to get the service. Then the factory would react on this and produce a new instance for each *getService* method, as the factory interface implies [18].

This would be a proper solution if there were different bundles needing the service, because a service factory only differentiates on a bundle level. Meaning that if it were

two *Service Trackers* within the same bundle, they would receive the same instance. As long as RB still only creates one bundle per *System Block*, this solution becomes a problem. Because as described earlier(2.3.1) a *System Block* will hold the logic of a whole application. Meaning that there can be multiple hardware components within a *System Block*, where each hardware component would have its own *Service Trackers* to handle.

The logic of a specific application would react to an event of an hardware, meaning that the OPC-UA client would send events based on the clients they are attached to. But since *System Block* is implemented into one bundle the different hardware components within, would not be provided with its own instances of an OPC-UA client, using a *Service Factory*. This led to further research which discovered something called a *Managed Service Factory*. [17] The point of source for this idea was one of the developers on the Apache Karaf project[6], Jeff Goodyear. See appendix B.2 for conversation and an alternative source for information.

Manged Service Factory

Managed Service Factory is a more advanced service factory which could produce instances dependent on other factors rather than which bundle called the *getService* method.

The reason that *Managed Service Factory* is able to create services based on other factors is that it leans on the *ConfigurationAdmin* [16] service. As the JavaDoc states:

OSGi Managed Service Factory JavaDoc

"Bundles registering this interface are giving the Configuration Admin service the ability to create and configure a number of instances of a service that the implementing bundle can provide. For example, a bundle implementing a DHCP server could be instantiated multiple times for different interfaces using a factory.
Each of these service instances is represented, in the persistent storage of the Configuration Admin service, by a factory Configuration object that has a PID. When such a Configuration is updated, the Configuration Admin service calls the ManagedServiceFactory updated method with the new properties. When updated is called with a new PID, the Managed Service Factory should create a new factory instance based on these configuration properties. When called with a PID that it has seen before, it should update that existing service instance with the new configuration information."

Based on this documentation and our current issue this seemed the way to go. So how could this be done? According to the documentation of the *Configuration Admin Service* [16], it was possible for each hardware to be implemented with its own configuration which could be used to differentiate the OPC-UA clients based on the precise hardware properties.

The solution which was advised from extensive research, was to generate a property file for each hardware and use an Apache Felix *File Install* to manage OSGi, directory based[4]. Now the hardware bundle could produce a configuration file for the needed OPC-UA client instance, and the service would be generated based on the proper config file.

Using this solution the generation of OPC-UA client services was abstracted to a simple configuration file, meaning that it was a simple way of changing the particular service later on. Each configuration file will have its own unique PID as mentioned before, which will be stored within the *ManagedServiceFactory* implementation. There were some issues with auto generating services which was hard to debug, which provided a *NullPointerException*. It will be handled in the next iteration.

7.2.3 Reactive Blocks Swap Hardware application

As decided earlier in the guideline section, creating a standalone application for each possible function within the ITS station was the proper usage of Reactive Blocks (RB). So this application had the logic for a simple car sensor which was counting cars driving by. See figure presented in iteration one, figure 6.2

Based on the figure there a exists Termination block, OSGi Event Handler block and a CarSensor block.

Termination block

The Termination block is a simple OSGi bundle stop handler. A bundle can be stopped either programatically, or through the console using the command stop together with a bundleId. It emits a stop signal which can be processed by the different blocks in the application. That is easy to see.

CarSensor

This block handles the logic enclosing the *CarSensor*. As you can see in Figure 6.3 the block is implementing a combined *ServiceTracker*, a *CarPoller* and the logic encircling these two blocks.

Its up for discussion on how the *CarSensor* should be implemented, for example if the poller should be one level up within the system block or right were it is. The

desicion fell on within the *CarSensor* because it was most logical to let the sensor control the poller as well. But its easy to see what is happening, and that this sensor is in need of different services to be able to work properly.

An important observation here is that the *Car Sensor* directly communicates with the hardware API which is not something that is desired in the future application. Its desired to connect the hardware and the OPC-UA client API to be able to administer the hardware.

OSGi Event Handler block

This is a fascinating block that occurred during this iteration. After trial and error there was some concurrency problems on how the services was produced and when the OPC-UA server was available for connection. The experienc was that the OPC-UA client service was registered and ready to connect before the OPC-UA server was properly started. So there had to be some kind of mechanism to be able to handle this concurrency.

That is where the light weight pub-sub subsystem, *Apche Felix Event Admin* could be of assistance.[3] This is an implementation of the OSGi Event Admin Service Specifications.[24]

What this specification does, is basically that it allows the developer to define channels where one can pass events and data across bundles residing within an OSGi container. The needed implementations is a form of *Event Publisher* which gets the *EventAdmin* service from the context, and publishes an event on a specific channel, code 7.2. To receive the event its needed to implement an *Event Handler* which deals with an event on a specific channel and is registered within the OSGi registry. See Appendix A.1

Code snippet 7.2: Event propagation example

```
ServiceReference ref= ctx.getServiceReference(EventAdmin.class.getName());
EventAdmin eventAdmin = (EventAdmin) ctx.getService(ref);
Dictionary properties = new Hashtable();
properties.put("serverStarted", server.isRunning());
Event serverEvent = new Event("ntnu/opcua/server/STARTED", properties);
eventAdmin.sendEvent(serverEvent);
```

The latter code was all that was needed to notify the clients when the server is properly started and ready to receive connections. Meaning that it can either implement as a service with an interface or be used within the code like this. It is important to have in mind a plan for mapping the different custom channel names that ends up being used. The summary discuss another solution 7.5

7.3 Hardware service bundle implementation

The whole idea behind this bundle is to communicate with hardware in a proper way. Creating a driver with a service that can be reached by the proper modules. The RB logic will directly use the hardware service to be able to handle the update scenario that has been proposed. There will be no focus on how to implement the OPC-UA client with the proper properties and models, but rather how an hardware can controll the instantiation of an OPC-UA client.

7.3.1 Hardware driver bundle

The hardware OSGi bundle driver, has very little logic implemented because there were no hardware attainable to test at this moment. But it has implemented a simple hardware API which would be the suggested standard API towards all existing hardware. See 7.3.

Code snippet 7.3: HardwareAPI interface

```
public interface HardwareAPI {
  public void start();
  public void shutdown();
  public String[] getData();
  public HashMap<String, Object> getProperties();
}
```

This is a simple API to be able to demonstrate how to control the hardware which was connected to the ITS station. The method *getProperties()* returns the specific properties for an hardware. This had to be added because the current RB OSGi *SimpleServiceTracker* does not support the properties that a service is registered with. Which was needed to separate the different types of hardware when using the standardized hardwareAPI towards all the different attached equipment.

As discussed in section 7.2.2 the OPC-UA client is driven by the "belonging" hardware and is instantiated and killed through a configuration file. The solution for now is that the hardware bundle will provide the proper configuration settings and deploy a configuration file into the designated folder. This would end up in spawning an OPC-UA client with services to browse, listen to and control the following piece of hardware, figure 7.1. The listening and browsing has not yet been implemented.

7.4 Unsolved Problems

- ✓ How to handle the OPC-UA client and hardware initiation relation.
- ✓ Create an OPC-UA client implementation

- ✗ How to provide the datamodel hardware connection
- ✗ How the APIs should look like
- ✗ Get OPC-UA events and methods running
- ✗ Proper OPC-UA server service
- ✗ Avoid errors using OSGi file install

7.5 Summary and concluding remarks

After this iteration it has become clear that these technologies are working better and better together. It is now possible to upgrade and switch out the hardware without any application critical troubles. Also deploying a hardware will now instantiate a client service for the hardware to be associated to. The next that will have to be handled is to add this into a datamodel which allows the hardware to be controlled by the OPC-UA client instead.

In regard to our research questions, its now becoming clearer and clearer on how to implement an application which will have the robustness wanted, and possible to upgrade and expand as desired.

Now that the OPC-UA client and hardware initiation relation have been handled, its provides the necessary abstraction needed to be able to avoid the single point of failure mentioned earlier. This is an important subject in regards to question one. It has also been proven that using OSGi gives the foundation of properly handling the upgrading process, as well as expanding the application.

In section 7.2.3 it was discussed an OSGi Event Handler. This solves the experienced concurrency problem but it was not the only solution to handle this. It is also possible to rely on a server status service which does not register unless the server has been initiated. This is easily done through a bundle context reference. This approach was shelved because it was much more interesting to try out OSGi events.

Chapter 8
Iteration 3: OPC-UA Client and hardware approach

"Again, you can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something - your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life. "

— Steve Jobs

During this iteration it was important to focus on combining the spawned OPC-UA client with the current datamodel for the attached hardware, that was going to be controlled and updated on datachanges. If this works well the hardware abstraction will be handled by OPC-UA, and every action from the application will go through the server and can then be registered and stored.

This iteration will try to expand the current objects in the "HardwareSwap" application from the last iteration. The plan is to abstract the hardware from the current RBs and let OPC-UA take full control over the hardware. Then the OPC-UA client will provide a service that lets the logic in RB react on specific changes and control the logic based on events and subscriptions.

8.1 Problems from last iteration

- ✓ How to handle the OPC-UA client and hardware initiation relation
- ✓ Create a OPC-UA client implementation
- ✗ How to provide the datamodel hardware connection
- ✗ How the APIs should look like
- ✗ Get OPC-UA events and methods running
- ✗ Proper OPC-UA server service
- ✗ Avoid errors using OSGi file install

During this iteration there is a lot of the problems from last iteration which will be handled. Such as connecting OPC-UA datamodel and the hardware, using OPC-UA methods and events, as well the experienced problem from last iteration, error during file install. The two problems with a proper API will not be handled here because this is very vague at this point.

Its also important to not forget that this thesis focus on robustness, upgrading and expanding, and we will have concluding remarks on how these solutions will aid in this aspect.

8.2 OPC-UA datamodel

After trying and failing it has come the that point were this thesis implements the OPC-UA datamodel towards the hardware. Meaning that the hardware which is connected will provide a service for this datamodel bundle which bridge these aspects and links itself into the OPC-UA server.

Everything is loosely coupled through services meaning that this has to be taken into consideration! So to mention again one of the most important aspects of OSGi and dynamism, code defensively. Have in mind that modules and services might disappear at any time, even not be existing at startup. This needs to be be handled in a proper fashion.

So the datamodel, this is basically a namespace creator for the current model of the present hardware. It creates the necessary properties, attributes, and connections to be able to have full control over everything that happens when the hardware is hooked up. It basically maps up the hardware properties and methods into a familiar OPC-UA datamodel. Have a look at section 2.4.4 for some basic insight.

The code is too long for appendix, so please refere to the writers GitHub account[1]

8.2.1 Why a datamodel?

First of all its the only way to implement OPC-UA because the object model maps every detail about the current hardware into an understandable abstraction. It is of course possible to dropp OPC-UA and have an API directly towards the hardware and use this out to the rest of the application. But this gives a tightly coupled application and a whole lot more development for the ITS implementer. They would have to implement everything that has to do with security, communication, history and especially a very good mapping towards what kind of values the API provides.

[1]The repo is private so contact the writer for access, through the information on his GitHub account.[21]

The last point is a very important factor for using OPC-UA and its datamodel. Consider that there is an API which provides the method *getSpeed()*. This returns an int which can be whatever. This is very possible to solve with just different types of methods, such as getSpeedInKM and getSpeedInMiles, but that means a more advanced API and more complex to handle dynamically.

But, having the OPC-UA datamodel/objectmodel, it all becomes browsable and its only necessary to agree on a general datamodel. Then its not important to know what the API is anymore, because you browse objects, find an object which is speed, then this has a value, properties, attributes, methods or whatever might be interesting. This will give the opportunity to look up what kind of measurements this object has, giving opportunities to have a much more dynamic environment. [46]

So the OPC-UA browsing API is already standardized, one factor removed, and then its only a standard datamodel with attributes/properties that can convey the differences behind the values.

8.2.2 Example datamodel

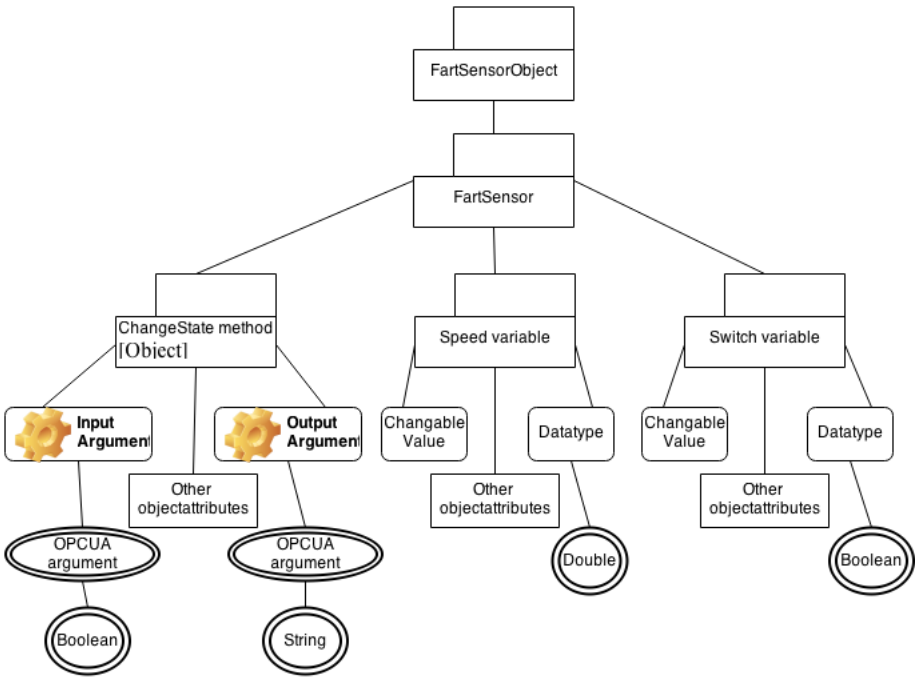

Figure 8.1: The FartSensor datamodel/objectmodel

The current datamodel, 8.1, is a very simple variant which has a variable speed object, a method and a variable switch object. The method and switch variable object is basically the same, its just two different ways of turning on or off the current hardware. As mentioned, it is a mapping of the hardware, so there have to be an implementation reacting to the methods or property changes within the hardware as well.

The example is connected to the hardware through a service registered by the hardware, with the opportunity to register a listener on the changes within the hardware. This follows the whiteboard pattern which is OSGi's way of implementing the observable pattern. [25]

To implement a datamodel is a big hassel because there is heaps of code for something that does not feel that important. But this datamodel provides freedom towards the hardware and its important to know about the different opportunities which OPC-UA provides. It is not in the scope of this thesis to cover all the possibilities which OPC-UA provides.

8.3 OPC-UA method and events

It was mentioned that this thesis is not going to cover the details of OPC-UA but this iteration started with an error of being able to use methods and events. These are important objects to be able to gain the proper control over the hardware and provide the needed attributes and properties that brings the wanted separation which upholds the robustness and possibilities to upgrade the hardware. Objects like events and methods are also inserted into the objectmodel, but since they are important and have been a task itself, they received its own section.

Its important to describe properly what can be done with the hardware through methods and not only attributes and properties. To be able to handle uncertainties and edge cases is what events are for. Therefore OPC-UA provides a nice framework for events which gives more control over happenings than the simple listener used on the speed value in the application. If not using OPC-UA, the developer would have to implement the wheel again with listeners to be able to provide the opportunities which OPC-UA already do.

8.4 Mapping of namespaces for each hardware

As mentioned before a goal for this application was to have an objectmodel for each hardware. But since OPC-UA is magically made up of the same objects, which is nodes, it would be difficult to browse all nodes and filter out on a specific string

property within these nodes. This would be time consuming its much more interesting to have a direct access to specific node which holds the information wanted.

Each time a node manager is created, which usually happens when a new hardware is added, a string has to be passed along, which is the namespace. By the way there already exists managers for the root nodes and standards nodes that the OPC-UA server provides.

The namespace cannot be used as a browsable unique identifier, because its only possible to browse on *NodeId*s which every node in the environment have. Therefore it was important to map the specific *NodeId* to the objectmodel of the hardware, to some familiar identity.

To solve this there was created a *NameSpaceController* service. This holds two hash maps containing the *NodeId*s and this familiar identity. The *NameSpace*s passed along with the creation of the node manager is also contained with the same familiar identity. So its possible to look up the specific *NodeId* with either a *NameSpace* or this familiar identity.

8.4.1 Familiar identity

Just to clarify what this identity might be. Earlier in the thesis it was decided that each hardware will create its own instance of a OPC-UA client, to be able to browse the hardware objectmodel in a proper way. The reason for this internal OPC-UA client was to provide a service to RB which could control the application logic.

But for a hardware to instantiate an instance of the OPC-UA client service based on something unique, it had to provide some properties. A config file was deployed with properties, and on of them were *hardware.name*, which is the familiar identity mentioned earlier. This identifies the current hardware. The *node manager* which holds the hardware object model would generate the string based on the property of the hardware, as well register its root *NodeId* with the namespace and with the property from the hardware.

For now its nothing more than a simple string, but this can of course be done a lot smarter. But that is to detailed for the application and this was a proper solution.

8.5 Other fixes and improvements

Along the iteration it became clear that some things had to be fixed. First of all file install produced a strange *NullPointerException* which no one could answer for. It also became complicated to work with the OPC-UA client instantiation and delivering other services to the instance.

8.5.1 OPC-UA client conversion to Declarative Services

The way the instantiation of the OPC-UA client was currently handled was with a *Managed Service Factory* and a bundle activator. Now that the OPC-UA client was dependent of another service, which was the *NameSpaceController*, problems started to arise. The code started to become ugly, there was a lot to have in mind when coding, and the decision was that *ManagedServiceFactory* and *bundle activator* was immensely low level API. It was difficult to expand and that is the opposite of what this thesis is focusing on.

Therefore it was important to convert to *DS* which provides the handling of the low level API so the devloper only have to focus on what he wants to do. Neil Bartlett has written a good book on starting with OSGi which is open and free. [29] The explanation to the the expantion problem was in his book, which seemed to be easily handled, by just converting to DS. The DS conversion done here is also disclosed in the appendix, A.2

The reason for this is that DS already is couple with *CM* and provides the same functionallites as the *ManagedServiceFactory* do, just hiding the low level API, and in a simpler way. Have a look in [29] at chapter 11, p.219 for a good introduction to DS. The specific solution was a combination of all sub chapters but mostly resided in subchapter 11.11 which starts on page 259.

8.5.2 File install error

After some trial and error it became clear that the loading configurations which the hardware produced, provided an unexplainable error. It did not break anything so it was ignored in the first iterations. It was annoying and had to be dealt with. The simple solution was that the configuration which was produced from the hardware could not be in the same root folder as the folder which the *Configuration Admin* listened too. After they were split up into each folder, the error never occurred. The Apache Felix mailing list was not able to provide any answers to this question. Nobody confirmed when I shared the solution either. So for now just make sure they do not share folders.

8.6 Proper OPC-UA server service

This has been mentioned as a problem because it currently only provides the whole server object to the instances who want to access the server. It is used by the objectmodel creators which need to have the server object to create the *NodeManager*. So the discussion is then if the server should provide its whole server object, or simply provide a detailed server service which allows the users of the service to add and modify objectmodels.

Passing around an object to multiple instances is not a proper way of doing java. The only place that the server object is actually needed is:

- ➥ Node manager instantiation
- ➥ Get the root folders such as object folder, type folder and views.
- ➥ Get different types available in the root manager

This could be solved through a simple OSGi service returning the desired objects: the node manager, rootfolders, different types, from the server. This is not implemented by this thesis but is food for thought and should be looked into.

8.7 Unsolved problems

- ✓ How to handle the OPC-UA client and hardware initiation relation
- ✓ Create a OPC-UA client implementation
- ✓ How to provide the datamodel hardware connection
- \- How the APIs should look like
- ✓ Get OPC-UA events and methods running
- ✓ Proper OPC-UA server service
- ✓ Avoid errors using OSGi file install
- ✗ Get update procedure to work properly
- ✗ Complete a RB application showing functionality
- ✗ Uphold robustness in application

8.8 Summary and concluding remarks

Things started to fall in place during this iteration. Modules were made and OPC-UA started to be involved in the project accordingly. It has been difficult to comprehend OPC-UA well, but nothing is impossible, just have to use enough time with it.

By starting to use DS with the OPC-UA client, it became easier to expand the client module and add more functionality to the current application. It also became easier to handle different egde cases such as cardinality and dependent services.

The application is starting to behave like a robust, upgradable and expandable application utilizing the technologies chosen, which is exactly what has been worked towards.

Chapter 9
Iteration 4: Reactive Block focus

"One reason so few of us achieve what we truly want is that we never direct our focus; we never concentrate our power. Most people dabble their way through life, never deciding to master anything in particular."

— Tony Robbins

Up to now there have been a lot of focus on the OPC-UA and OSGi components of the application, and getting them connected in a proper way. Its time to put some of the attention over to Reactive Blocks (RB).

As mentioned many times before, this is where the logic is going to reside. Were the edge cases will visually be handled and were one can go to understand what each application ends up doing in the different cases.

The first three proper iterations focused on a swap application, which was modified properly in the last iteration. The hardware was abstracted from the application and moved the communication to the OPC-UA layer which was the plan all along.

9.1 Problems from last iteration

- ✓ How to handle the OPC-UA client and hardware initiation relation
- ✓ Create a OPC-UA client implementation
- ✓ How to provide the datamodel hardware connection
- \- How the APIs should look like
- ✓ Get OPC-UA events and methods running
- ✓ Proper OPC-UA server service
- ✓ Avoid errors using OSGi file install
- ✗ Get update procedure to work properly
- ✗ Complete a RB application showing functionality
- ✗ Uphold robustness in application

This iteration will continue to work on the swap application and modify the RBs and the belonging OSGi bundles to visualise more logic and taking care of multiple hardware within one application. It will also focus more on upgrading and robustness. But first of all it will be discussing how the upgrading will work now that the communication is going through the OPC-UA layer rather than directly to the OSGi hardware driver bundle.

9.2 Upgrading through the OPC-UA layer

It was clear from the latter chapters that it was pretty simple to upgrade the hardware when it was directly cooperating with the hardware through OSGi services. Then it was explicit how the process should be handled, because all that had to be done was to react on *ServiceRegistration.UNREGISTER* event gracefully. This was done through the *ServiceTracker*. Having a direct link between the hardware and logic, it created a tight coupling, which is not what is desired.

Since the plan was to have everything pass through the OPC-UA layer providing a lot of wanted features, it would not be as straight forward.

The positive thing about abstracting the hardware layer with an OPC-UA layer, is that now its possible to upgrade the implementation without the application logic having to react on the activity. When the hardware API was directly connected to the RB, the service disappeared on an update, and had to be handled. Now the service only disappears in the datamodel, but reappears just as quickly. Since the service has been set to cardinality [0..1] with dynamic option, A.3, the service is not an necessity. Meaning that updating the hardware implementation have become a breeze. It wont affect the system a whole lot, as you can see in figure 9.1. And it will be easy to handle a request on the driver implementation when you know its unregistered. The time it took to update the bundle was milliseconds. This is of course just this simple hardware, but updating does not demand a great deal of time unless there is a huge startup process.

The same goes for the RB application logic implementation. It does not affect the system in a big deal when it updates, because its not a provider, rather just a consumer. It has a bigger startup process because it have to connect to the server and prepare for OPC-UA client events.

Both updates of these two bundles work fine in the demo application which is acquirable through GitHub[1].

[1]The repo is private so contact the writer for access, through the information on his GitHub account[21]

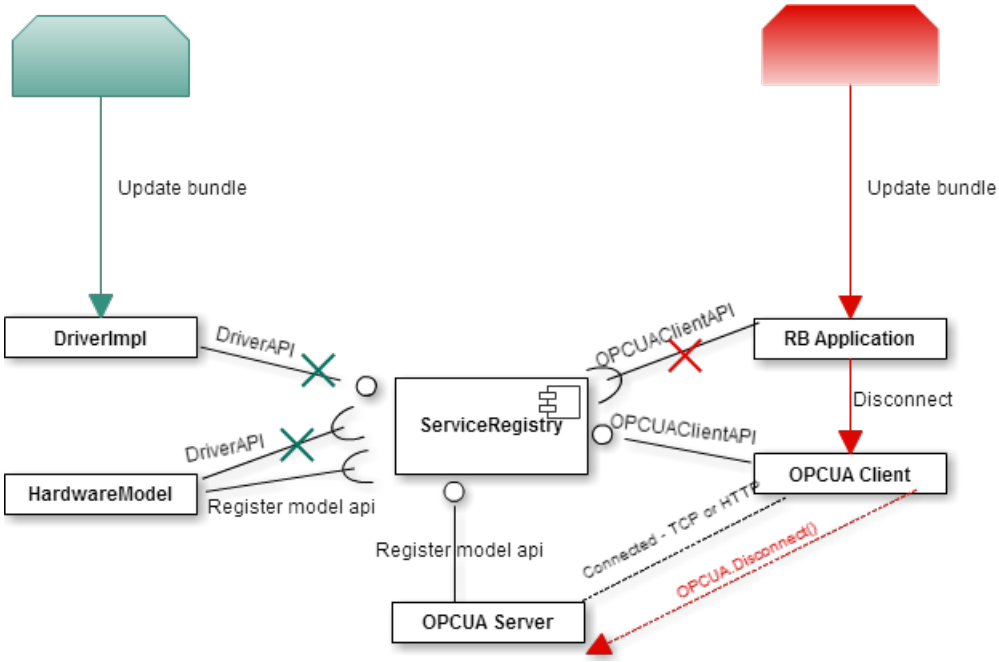

Figure 9.1: The affect of an update of the bundle impl

9.2.1 Some important factors

There are some important factors to consider when dealing with this second layer and updating and expanding. Right now expanding is not much of a subject since it will be so loosely coupled that the new application will not affect the rest of the system. It is just important to remember to deploy all the nessecary bundles to have the application running. How this can be done is discussed in 5.4.

After the OPC-UA layer was introduced, communication through listeners and the whiteboard pattern, is introduced. This will be covered more later. But these listeners becomes an issue when bundles are updated. Because bundles updating means the instantiation of a new listener. While other bundles which have used the specific listener in their instantiation, have to have their listeners reset to the new instance. If not, the reference to the current listener will be incorrect.

Since the OPC-UA client service is dependent of the configuration file of the hardware driver, this becomes an easy task to handle. It the config file does not change, the client service will not go down. Meaning that the driver implementation just updates without reacting on the java code update. This is because CM handles this properly. It is different if the bundle is stopped, but this is because of how driver activate and deactivate is implemented. Currently, when stopping a hardware, the

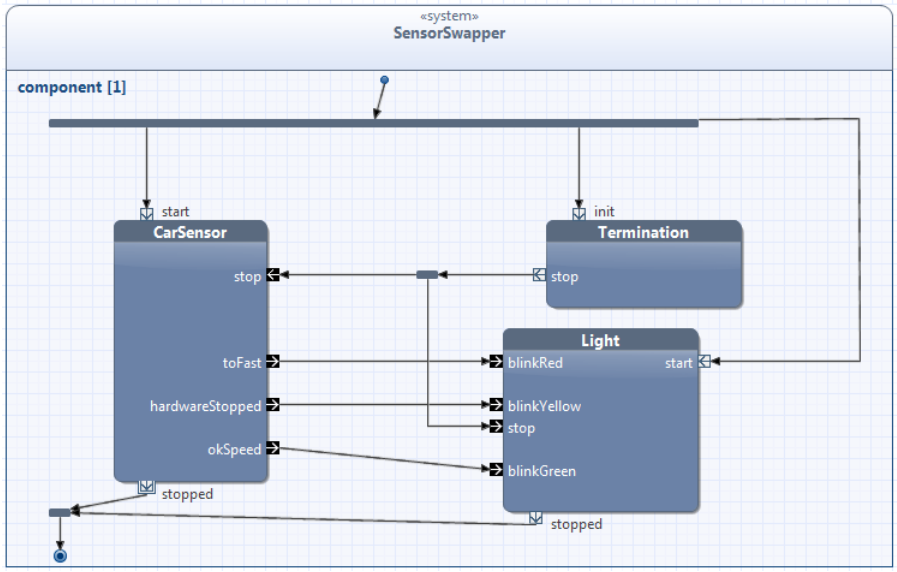

Figure 9.2: The current swap application with another hardware

adherent configuration file is deleted.

Lastly, when updating, it basically means that the bundles is going to stop, and then start over, so its important to remember the most important coding principle for OSGi, code defensively.

9.3 Reactive Block application state

Now the application is in a state where its possible to update the most important modules without taking down the system. These modules are the RB application and the hardware driver implementation. The hardware can be controlled by an OPC-UA client, which is what spawns for each hardware. But since the client is programmed by the writer and made pretty simple to get things working, it can be smart to have an advanced client connected to the server to have better control over the datamodels. Meaning a visual representation of the datamodel and the possibility to change and read attributes on the hardware, for testing purposes. The one that has been used in this thesis is mentioned in chapter 4.

9.3.1 The reactive block swap application

Figure 9.2 portrays the current RB application which uses an OPC-UA client service to communicate. It is a simple speed sensor with a light blinking red or green dependent of the speed. Yellow if something is wrong with the sensor.

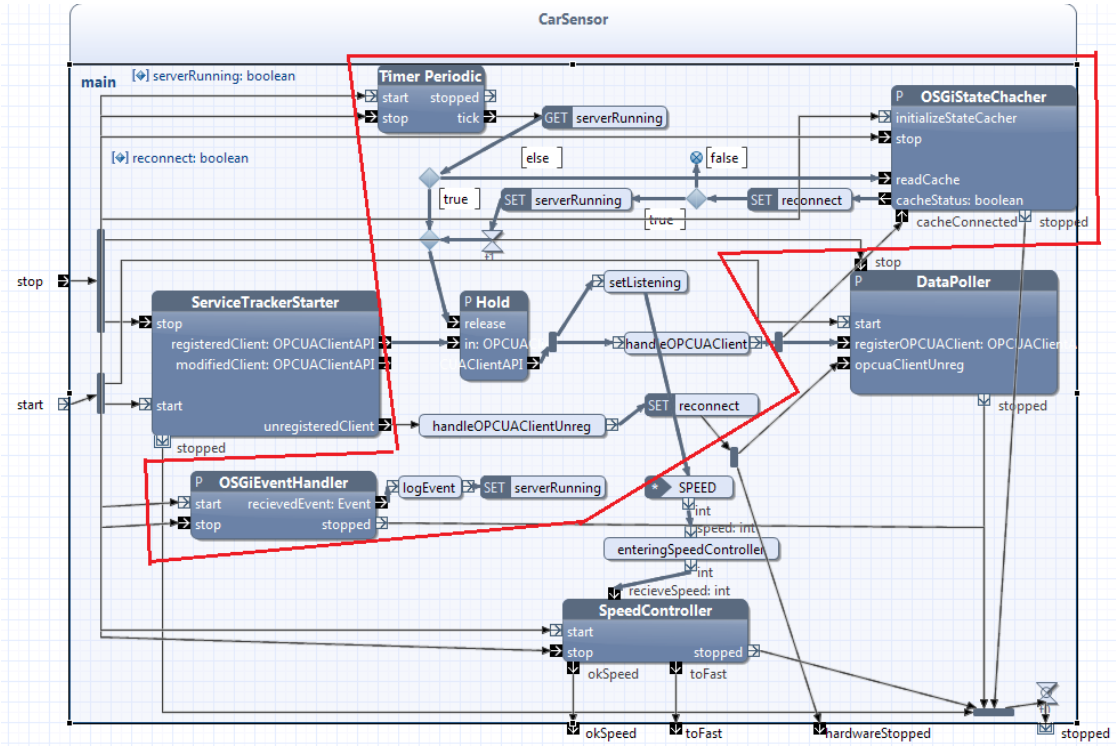

Figure 9.3: The current sensorcomponent, reusable logic is marked

This is all loosely coupled and is using the OPC-UA layer to communicate the events and methods. But this displays how easily an application can look with RB. There is of course a lot of logic behind the whole application and if one dig deeper into the two components, there is a mess at the moment.

9.3.2 CarSensor component

The car sensor component is a RB mess, 9.2, because the developer is not a good RB designer. The logic on the other hand works and is the reason it is possible to update the RB application without affecting the system.

What basically happens here is that the OPC-UA client service is picked up by the tracker with an OSGi filter which sets the lookup to be the *OPCUAClientAPI* with the property *hardware.name=this.name*. Then its possible to filter out the wanted clients. The tracker is an updated version of the one used in the beginning of this thesis. Its called a *Native Service Tracker* and is within the RB OSGi library.

After the service is acquired the component waits for the server to start. Then the *Hold* block releases the service. Then the *CarSensor* connects and sets a listener

to data updates from the server.

Using a data listener directly towards the OPC-UA client is not necessary the best way of doing things, because it is not very generic. It has been done to test if the data was received, and with this application that is confirmed. When data is received it flows towards the logic which reacts to the current speed received. The *DataPoller* can be ignored, it is just included to visualize how the first version received data from the hardware.

The *OSGiStateCacher* has been introduced to handled the update procedure for the RB application. The reason is, because the block solely depend on event propagation from the server, and since the server is already running, the state will have to be cached between start and stop. Currently the cache is cleared on each stop of the OSGi container itself, because the cache is persistent across container failures as well. This is done through a simple property in the container launch arguments: *-Dosgi.clean=true* [40]

This is considered a hack and should rather be handled with services. It might be smart to declare a *ServerStatus* DS component in the server bundle which can provide better service state handling. This was tested in 7 when looking at solutions to handle the concurrency issue regarding the OPC-UA server.

But the *OSGiStateCacher* only stores a value in the OSGi persistence storage, which is accessible through the bundle context. Then it reads the value upon request and returns a boolean so the service can be released from the hold block if the server was earlier running.

Within the red stripes of figure 9.3, there is logic which can be moved into its own block to keep the application simpler. Its also worth noticing that the marked logic is something that is also used in the light component 9.3.3. So it is prone to be created into its own block to enable reuse.

9.3.3 Light component

This component is very similar to the sensor component. It uses the same logic for stop and start as well as service loss, and it has specific logic which reacts to events received from the sensor.

This is the *ActionHandler* which uses the OPC-UA client API to retrieve the method belonging to this hardware. When the *NodeId* of the method is retrieved, it uses the client service to generate the right inputs and invoke the method based on the given NodeId. So when it receives an event from the latter *CarSensor* it can react on it through the OPC-UA layer. For information, the *CarSensor* also received

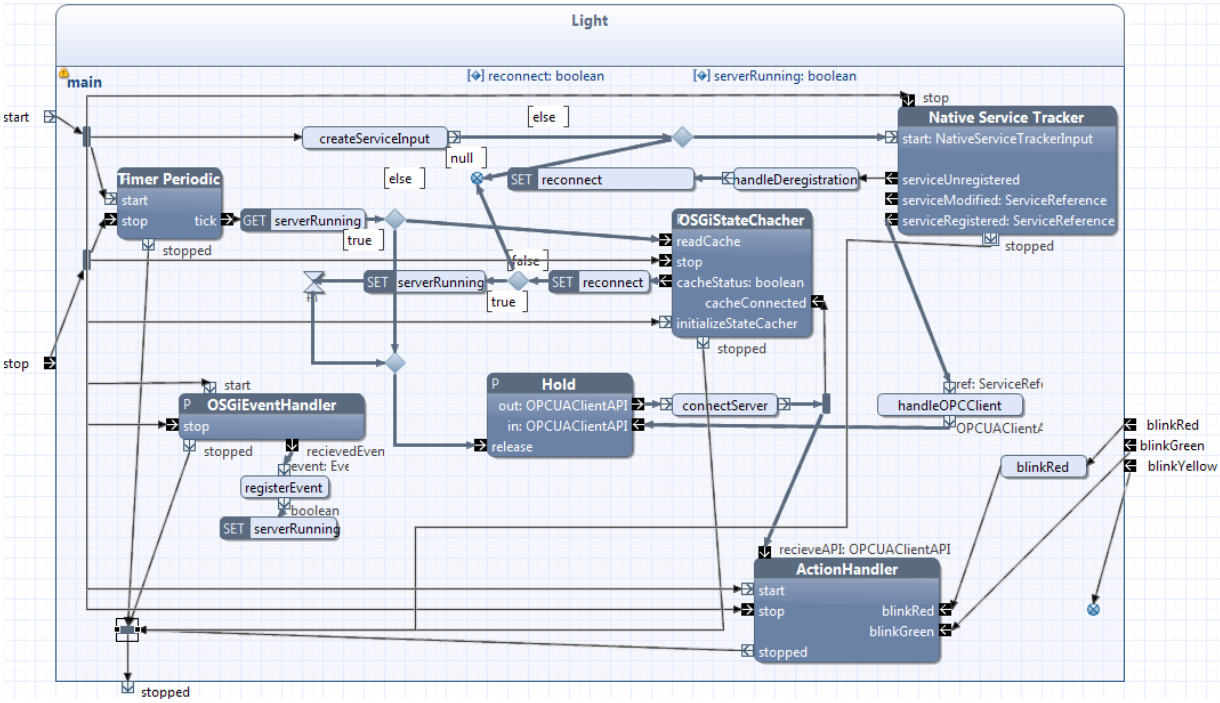

Figure 9.4: The current lightcomponent

the event and data from the OPC-UA layer.

9.4 OPCUA Client

There were some changes to this module to get everything to work properly. After studying the sample client from Prosys[64], provided through their java SDK evaluation, it was clear that having a way to call a method on the client API was the way to go. This opens up for a more generic OPC-UA client. Right now it is possible to call a method based on a NodeId and generate the inputs based on values in our logic. By implementing this in the OPC-UA client, all the different clients are able to call methods as well. And abstract the OPC-UA client from being familiar with the hardware implementations.

Right now the client is not very generic overall, but this is something that should be considered in the near future. A generic client is important for the application to work properly and be expandable. By generic client, it is meant that the client should not have any sense of what belongs behind each datamodel. It should just expose what is needed in a proper way.

Something that has been discussed is the possibility to create a service based

on the possible methods which is available through the deployed datamodel of each hardware. It has not been tried out but is of course a possibility. Take full advantage of OSGi. This is a complicated discussion and reflection has been up to consideration, but has been discouraged. It is very dependent on the overall architecture on how it should be done. But the thesis was not able to cover this subject, just slightly become familiar with the issue. It is recommended for future work. 10.4

9.5 Listeners and whiteboard pattern

The whiteboard pattern was mentioned in the latter chapter with the listener between the datamodel and the associated hardware. But the reason it is brought up again in this chapter is that it became an issue because of upgrading, and its important to highlight the different essential listeners in use. It was mentioned earlier that listeners had to be reset when an update was at hand. And this is what happens when designing with multiple references and using listeners across different layers.

The second essential listener which needs to be updated is the whiteboard pattern listener, *DataValueListener*, which the OPC-UA client registers. This listener is implemented by the *CarSensor* logic to be able to receive the necessary data values regarding the speed. This is not generic at all but is an example on how the whiteboard listener works, and needs to be reconsidered. But this is covered by the generic client future work. 10.4

This architectural solution should be reconsidered in future work, but is a functional solution for this example. Testing this solution allows for knowledge and awareness of the different issues when working with these technologies.

9.6 Discussions with experts and users

During this iteration it was initiated contact with some experts and people who work with the different technologies. Peter Kriens and Richard S. Hall are two OSGi experts with valuable information regarding the OSGi thoughts of the application. Martin Mueller is a user of the Prosys SDK and was provided to me as someone who has worked with OPC-UA and OSGi.

9.6.1 Peter Kriens

After a discussion with Peter Kriens, there were some issues that arose. For example the way this application is listening to a server state change event is not an OSGi custom. He said that the only time something should wait, is when waiting for a service. By this he means that things should not be initialized through event

propagation, but rather be dependent on services. This gives OSGi proper control and does not leave room for unforeseen errors.

He also mentioned that there exists a specification which could aid in device access for the hardware bundles, and it was something that was worth reading. This is called the *Device Access Service* Chapter.103 [23].

9.6.2 Richard S. Hall

Richard is one of the authors of OSGi in Action[47] and was glad to aid in the academics. The dialogue evolved mostly OSGi specific subjects and errors experienced in the application. But after the discussion more of the OSGi errors turned into more architectural problems rather then OSGi specific problems. One was the generic OPC-UA client dilemma. How this could be solved the best way, is an architectural question which is noted down for future work, 10.4.

9.6.3 Martin Mueller

Martin is a professional user of the OPC-UA SDK and offered some tips regarding designing an OPC-UA architecture. One of the hints from him was to put the OPC-UA server as close to the data source as possible and provide a client to each hardware. Because then each hardware would have the possibility to communicate with each other. Going an all OPC-UA architecture. Then have a OPC-UA master server for the ITS stations itself. This is not something that will be pursued in this thesis, but is an alternative for future work.

Martin also mentioned that the way this thesis have mapped the *namespaceindex* might not be a recommended approach. He suggested to look into OPC-UA types and using these more laboriously to be able to reuse and have better browsing capabilities.

It is understood that OPC-UA user specific types can be created, meaning that its possible to create a parent object for a speed sensor, and then mapping down the different speed sensors underneath. Giving a new entry point to all the different speed sensors for example.

9.7 Important developing hints

There were some errors which wasted a lot of time during this iteration, and its important that they are uncovered so the next person will not waste time on the same problems.

9.7.1 Silent Exceptions

When connecting all these different technologies and there are multiple different threads doing work around the application, it might freeze at different places if there is a bug. These freezes happens because of silent exceptions which is unclear why happens. It was experienced a silent *NullPointerException* when trying to set a listener on an object which did not exist. The application froze if the exception was not caught. Its important to have in mind what might actually happen.

It was also experienced a silence exception when creating the simulation code for the *FartSensorImplementation*. But then it was not solved through a special exception, but rather rearranging code to get it to work.

9.7.2 OPCUA monitoring value update

The way that the car sensor received speed updates was through a OPC-UA monitor value listener on the client. Some times this did not trigger even if *setValue()* was called. It was unclear why at certain points. But one important factor is that it does not trigger *onDataChange()* if the value is set to be the same as the last value. This is of course a logical way of responding to data change, but it was not clear in the beginning.

9.8 Unsolved problems

- ✓ Get update procedure to work properly
- ✓ Complete a RB application showing functionality
- ✓ Uphold robustness in application

9.9 Summary and concluding remarks

Through out this iteration there have been a focus on RB and starting to design these in a proper way. Because all of the backend/OSGi bundles that was core functionality had been developed and ready to be used with RB. The existing bundles worked properly but had traces of being developed for a very specific example, rather than focusing on the generics of the application. Meaning that there were some specific handling in the OPC-UA client dependent on which hardware being used. This iteration tried to handle it with some generic *inputGenerator* and *callMethod* but it was soon noticed that there had to be some specific handling of the different possible methods in the RB logic layer. This was of course a smarter way to handle the differentiation, rather than doing it in the OPC-UA client which is suppose to be as generic as possible.

The example application handles updating, stops at unforeseen events, as well as expansion of the application. By expansion it is understood that this can be to expand the current logic layer with another hardware that should be a factor to the current logic, and not just a whole new application logic. A new application logic block is not necessary the hardest thing to handle because everything is loosely coupled.

But through these development phases it is clear that RB is very capable of handling the logic regarding the application. Meaning that it will be easier to understand what is going on and can be easier to create new applications just using the already existing services. This abstracts a level of understanding away from the OPC-UA layer and lets the developer focus on what is important, the specific logic in the application.

Chapter 10
Evaluation and conclusion

"A conclusion is the place where you got tired thinking."

— Martin Henry Fischer

10.1 Summary

Through out this thesis there have been discussed a combination of three technologies creating an architecture which were to focus on three different research questions 1.3. There have been provided background for each technology, as well as related work. But since this particular combination of technology is entirely novel, there were not much work directly related to this thesis, rather work which touches a subject within one of the specific technologies.

The research questions had a focus on robustness, upgrading and expanding, and they were:

RQ1 How can the application be made robust in terms of error handling and edge cases?
RQ2 How can the application be upgraded without any inconvenience?
RQ3 How can the application be expanded without any inconvenience?

There have been iterations with different types of focus to achieve results in regards to the research questions. During these iterations the questions have been discussed in view to the different types of solutions and problems that has occurred on the road.

It started with a simple technology assessment to uncover the different benefits each would provide. This went rather smoothly and brought to light different traits for each technology. The different traits are discussed in contrast to the questions and terms this thesis is seeking.

Subsequent came iterations which step by step developed an application into something that had the different desired aspects implemented. During these iterations it was assessed if these technologies cooperate together in a proper way and aids in solving this thesis considerations.

10.2 Evaluation

The task at hand was very sophisticated which undertakes three excessively complex technologies into a suitable architecture. Each provide opportunities to solve both similar and different issues. It has been mentioned to the writer that solely OPC-UA takes 1 year to comprehend completely.

These three technologies are created to master their specific tasks single-handedly. And do a mighty fine job in their fields. All have their traits and a combination might be valuable, but is complex to achieve.

10.2.1 OSGi

OSGi alone provide the application with a lifecycle which opens doors for new ways to handle application errors and updates. Gives the opportunity to stop modules which crashes and update specific modules without affecting the rest of the system. But as mentioned it is a complex technology and when using this with minimal experience, faulty design is a big possibilty which again leads to errors.

10.2.2 OPC-UA

This is an intricate technology which has been developed for years by intelligent people. It provides the possibility to map hardware or other devices into a familiar concept, giving one access point to a whole new world. Hardware comes with different properties and functionallity, and its important to be able to standardize this into one portal of conduct. The positive part is that it maps up the hardware traits into familiar nodes and attributes, which can be handled easily as long as OPC-UA is a recognizable technology.

10.2.3 Reactive Blocks

Reactive blocks is the solution to messy code and handling real time applications which is dependent of state and events. It is quite advanced in handling logic on a visual level, but is missing the lifecycle layer as well as the possibility of standardizing, which makes it a complementary. RB has to be accounted for in the architecture which adds yet another layer with opportunities.

10.3 Conclusion

As been mentioned many times through out this thesis, the technologies do match each other in a good way and do benefit from each other. Aiding in resolving the three terms, robustness, upgrading and expanding. Please refer to chapter 5 to see which technology complements what term.

But all in all it is a good match which, if done properly, can create a powerful architecture. There may be a lot of bumps along the road because of the complexity of each technology and to be able to use them in the best functional way. But in the end beneficial value will prevail.

This thesis have covered some of these bumps and proposed some ideas which can be considered. But considering the complexity, one thesis will not cover this a 100%. Delving into each technology has not been a purpose, that can be split up into each its own paper. The focus have been to get a good overview of benefits, and being able to corporate these technologies in an example, which provides thoughts enclosing the different questions and issues.

Through experimenting and trying out the configurations and possibilities that OSGi provide, such as complex service factories, whiteboard pattern listeners, event admin, configuration admin and lifecycle handling. It has become clear that OSGi is a powerful technology which provides possibilities that is a perfect match for the scenarios and research questions introduced in this thesis. These evolves robustness, upgrading and expanding. The focus has been to try out the opportunities, to be able to modularize and provide a lifecycle for each specific module. Resulting in an application which handles dynamism that might occur when working with other factors such as hardware. This ends up in covering the robustness needed in an application like this. Because of the modularization and lifecycle, OSGi provides the opportunity to create new versions of an existing bundle resulting in that upgrading becomes a breeze. After discussions with experts it was also confirmed by Peter Kriens that this is a situation which OSGi is a good fit.

OPC-UA is not directly an aid in these three terms, but indirectly assist in keeping robustness, abstracting some layers when considering updating and overall provide a lot of out of the box functionality. The provided services is something that is a hassle to develop all over again and may be error prone if done yourself. Things such as modelling/mapping the hardware, secure connection to each server, browsing capabilities, concurrency in data reading and proper event handling. It is features which takes time to develop. But using OPC-UA, it becomes painless. OPC-UA is not always a breeze, its complicated and is something that has to be properly learned before implemented into a project. Because the possibilities are endless, and the way nodes are connected into a full mesh network, it provides positive possibilities. There

should not be a need to develop a lot of aiding modules towards OPC-UA because OPC-UA is quite complex and will handle most cases one will ever consider. It was mentioned to focus more on OPC-UA types rather than mapping up self developed *namespaceindexes*.

Reactive Blocks (RB) is not a necessity but a much wanted feature because clean code is not a simple thing to do. RB visualizes the code while one actually develop functionality. Merging two steps into one, the architecture design and the implementation itself, which results in having the best of both worlds. It is important to design the architecture to implement RB. Its not something that just is merged into the application, but would have to be designed to be able to append in the architecture. But if the main modules, such as OPC-UA client, OPC-UA server and hardware, are designed properly with concrete services, creating applications using RB would not be a problem at all. The focus would be reuse and development of the logic.

Because this architecture is so modularized, and application logic is created based on services, it is pretty simple to engage multiple developers into this project. Meaning that hardware modules, OPC-UA datamodels, and RB application logic can all be developed on its own as long as the OPC-UA client is generic enough. Meaning that the OPC-UA client does not have to be changed each time a new hardware or datamodel is added. This is possible because these modules does not have a tight coupling.

10.4 Future work

There is still things to improve and ideas to develop on the architecture, but a foundation and experiments have been made. Right now it cannot be deployed, but that can be blamed that this thesis currently do not have any example hardware to work with, and did not get to a stage where it was worth working with hardware. If real hardware was obtained, a driver and a test datamodel could be implemented which means a real hardware working example. The road to this stage is not long and is something that should be pursued.

After feedback with OPC-UA users, alternative architecture solutions have been hatched. Some focusing on a purer OPC-UA architecture, which was an example from Martin Mueller, 9.6. He was not familiar with OSGi or RB but based on his OPC-UA skills, he suggested that this was the way to go. So it is worth to make up some ideas regarding this solution.

Considering OSGi feedbacks from experts such as Peter Kriens and Richard S. Hall, there are issues that should be taken into consideration. For example, the

architecture now is mixing a service based model with an event model. It has been mentioned that it would be better to focus on one or the other. It has been tried out to replace the event handling with services, and was not a complicated task. Its just important to remember to track those new services.

The current OPC-UA client is not at this point perfectly generic, and it has been mentioned multiple times that this is something which should be engaged into. It is a complicated task and might be suited to spur out to a assignment of its own. The conversation with Richard S.Hall brought up some consideration on how to handle the generic client method functionality. Meaning that using a generic OPC-UA client is difficult because its not possible to have a contract over the potential methods of a hardware. Reflection was discussed but not necessary the best way to go. But this is a complex architectural choice for future work.

References

[1] Analysis of Control Architectures for Teleoperation Systems with Impedance/Admittance Master and Slave Manipulators. http://ijr.sagepub.com/content/20/6/419.short.

[2] Apache CouchDB. http://couchdb.apache.org/.

[3] Apache Felix Event Admin. http://felix.apache.org/site/apache-felix-event-admin.html.

[4] Apache Felix File Install. http://felix.apache.org/site/apache-felix-file-install.html. Apache Felix documentation.

[5] Apache Felix OBR. http://goo.gl/KkB3ty.

[6] Apache Karaf. http://karaf.apache.org/.

[7] Apache Karafe Features. http://karaf.apache.org/manual/latest-2.2.x/users-guide/provisioning.html.

[8] BND-Tools. http://bndtools.org/.

[9] Conversation between Snorre Edwin and Jouni Aro regarding OPC-UA client architecture. http://www.prosysopc.com/blog/forum/opc-ua-java-sdk/opc-ua-and-osgi-providin-server-querying-as-a-osgi-service/.

[10] European Commission. http://goo.gl/Kn7UDn.

[11] Introduction to Karaf Featuers. http://icodebythesea.blogspot.no/2012/03/making-osgi-deployments-easier-with.html.

[12] IRC Org. http://www.irc.org/.

[13] Merriam-Webster dictionary. http://www.merriam-webster.com/dictionary/robust.

[14] Merriam-Webster dictionary. http://www.merriam-webster.com/dictionary/upgrade.

[15] Merriam-Webster dictionary. http://www.merriam-webster.com/dictionary/expand.

[16] OSGi Configuration Admin. http://www.osgi.org/javadoc/r5/cmpn/org/osgi/service/cm/ConfigurationAdmin.html. Java doc.

[17] OSGi Mangaged Service Factory. http://www.osgi.org/javadoc/r5/cmpn/org/osgi/service/cm/ManagedServiceFactory.html. Java doc.

[18] OSGi Service Factory. http://www.osgi.org/javadoc/r5/core/org/osgi/framework/ServiceFactory.html. Java doc.

[19] OSGi Subsystem. http://www.osgi.org/javadoc/r5/enterprise/org/osgi/service/subsystem/Subsystem.html.

[20] The US national ITS architecture. http://www.iteris.com/itsarch/.

[21] Writers GitHub account. https://github.com/Snorlock.

[22] OSGi Alliance. http://www.osgi.org/Technology/WhyOSGi.

[23] OSGi Alliance. *OSGi compendium spesifications - Release 5*. OSGi Alliance. http://www.osgi.org/Download/Release5.

[24] OSGi Alliance. *OSGi spesifications - Release 5*. OSGi Alliance. http://www.osgi.org/Download/Release5.

[25] OSGi Alliance. Listeners Considered Harmful: The "Whiteboard" Pattern. *Technical Whitepaper*, 2004. http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf.

[26] Unfified Automation. *Unified Automation OPC-UA documentation*. Unfified Automation. http://documentation.unified-automation.com/uasdkcpp/1.2.1/main.html.

[27] Unfified Automation. *Unified Automation OPC-UA documentation Node Classes*. Unfified Automation. http://documentation.unified-automation.com/uasdkcpp/1.2.1/L2UaNodeClasses.html.

[28] Unified Automation. Unified automation. http://www.unified-automation.com/.

[29] Neil Bartlett. Himself, 2009. http://njbartlett.name/files/osgibook_preview_20091217.pdf.

[30] Bitreactive. http://www.bitreactive.com/.

[31] Bitreactive. Bitreactive get started guide. http://www.bitreactive.com/get-started.

[32] Bitreactive. Developer page. http://www.bitreactive.com/developer. This is were the refrences and guides are.

[33] Bitreactive. Overcoming the Challenges of Reusing Software. http://www.bitreactive.com/download.php?filid=15204d533672d3-.

[34] Bitreactive. Reacting on external events. http://reference.bitreactive.com/doc/reacting_on_external_events.

[35] Bitreactive. Sessions and multiplicity page. http://reference.bitreactive.com/doc/multipicity_of_blocks.

[36] Bitreactive. The Secret Twists to Efficiently Develop Reactive Systems. http://www.bitreactive.com/download.php?filid=1519b42a5df214-.

[37] Oracle Blog. https://blogs.oracle.com/abuckley/entry/jsr_294_and_module_systems.

[38] H. Cervantes and Jean-Marie Favre. Comparing JavaBeans and OSGi Towards an Integration of two Complementary Component Models. In *Euromicro Conference, 2002. Proceedings. 28th*, pages 17–23, 2002.

[39] OPC Connect. OPC Unified Architecture. http://www.opcconnect.com/ua.php.

[40] Eclipse Documentation. http://goo.gl/7Ss5ae. Under the Arguments Tab link.

[41] Eclipse. http://www.eclipse.org/osgi/.

[42] Eclipse. http://wiki.eclipse.org/Older_Versions_Of_Eclipse.

[43] OPC Foundation. *OPC UA Spesifications*. http://www.opcfoundation.org/default.aspx/uaspecdownloads.asp?MID=Developers.

[44] The Apache Software Foundation. http://servicemix.apache.org/.

[45] D. Grossmann, K. Bender, and B. Danzer. OPC UA based Field Device Integration. In *SICE Annual Conference, 2008*, pages 933–938, 2008.

[46] Thomas Hadlich. Providing device integration with OPC UA. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4053398.

[47] Richard S. Hall. *OSGi in action: creating modular applications in Java*. Manning, 2011.

[48] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Q.*, 28(1):75–105, March 2004.

[49] Lu Huiming and Yan Zhifeng. Research on Key Technology of the Address Space for OPC UA Server. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 3, pages 278–281, 2010.

[50] ISO. *C programming manual*. ISO. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[51] ISO. *C++ programming manual*. ISO. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf.

[52] JavaWorld. OSGi vs Jigsaw. http://goo.gl/54KX2k.

[53] JCP. http://jcp.org/en/jsr/detail?id=277.

[54] OPEN JDK. http://openjdk.java.net/projects/jigsaw/. Website of Jigsaw project.

[55] Joseph D. Camp and Edward W. Knightly. The IEEE 802.11s Extended Service Set Mesh Networking Standard. Technical report, Rice University, 2008.

[56] Satya Maheshwari. http://goo.gl/NvdMtg. This is a session from the OSGi Community Event 2013.

[57] Wolfgang Mahnke and Stefan-Helmut Leitner. OPC Unified Architecture - The Future Standard for Communication and Information Modeling in Automation. http://goo.gl/CW2xWz.

[58] Simone Massaro. What is OPC UA and how does it affect your world? http://goo.gl/ApordY.

[59] Prosys OPC. *Prosys OPC UA Java SDK - Client Tutorial*. Prosys OPC.

[60] Prosys OPC. *Prosys OPC UA Java SDK - Server Tutorial*. Prosys OPC.

[61] OPC-UA foundation. https://www.opcfoundation.org/Default.aspx.

[62] OSGi Alliance. http://www.osgi.org/Main/HomePage.

[63] Mitch Pronschinske. 'OSGi vs. Jigsaw: Kirk Knoernschild on Modularity'. http://java.dzone.com/articles/osgi-vs-jigsaw-kirk, 2012.

[64] Prosys OPC. http://www.prosysopc.com/.

[65] Olivier Roulet-Dubonnet. An Evaluation of OPC-UA for Data Collection from the Norwegian Road Network. Technical report, SINTEF, 2012.

[66] JBoss Application server. http://www.jboss.org/jbossas.

[67] SINTEF. *Roadside ITS station specification - Functional and technical requirements*. Statens Vegvesen.

[68] Kenneth Sørensen, Terje Moen, and Cato Mausethagen. Implementation of CVIS ITS Application in a Driving Simulator Environment. 2011.

[69] Wolfgang Mahnke Stefan-Helmut Leitner. OPC UA – Service-oriented Architecture for Industrial Applications. http://pi.informatik.uni-siegen.de/stt/26_4/01_Fachgruppenberichte/ORA2006/07_leitner-final.pdf.

[70] Snorre Lothar von Gohren Edwin. Modularity and Lifecycle of OSGi Applications. https://www.researchgate.net/publication/255703644_Modularity_and_Lifecycle_of_OSGi_Applications.

[71] Fei-Yue Wang, Shuming Tang, Yagang Sui, and Xiaojing Wang. Toward Intelligent Transportation Systems for the 2008 Olympics. *Intelligent Systems, IEEE*, 18(6):8–11, 2003.

[72] Pang-Chieh Wang, Cheng-Liang Lin, and Ting-Wei Hou. A Service-Layer Diagnostic Approach for the OSGi Framework. *Consumer Electronics, IEEE Transactions on*, 55(4):1973–1981, 2009.

[73] Wikipedia. http://en.wikipedia.org/wiki/OSGi.

[74] Jiankun Wu, Linpeng Huang, and Dejun Wang. ASM-based Model of Dynamic Service Update in OSGi. *SIGSOFT Softw. Eng. Notes*, 33(2):8:1–8:8, March 2008.

[75] Daqing ZHANG and Xiao Hang WANG. OSGi Based Service Infrastructure for Context Aware Automotive Telematics.

Appendix A
Code blocks

"Appendix - Just like the intestines, small outgrowth from large intestine or additional information accompanying main text"

— Anonymous

A.1 Event Handler implementation

Code snippet A.1: Event handling code

```
public void registerEventHandler() {
  EventHandler handler = new EventHandler()
  {
   @Override
   public void handleEvent(Event event)
   {
       logger.debug("RECIEVED EVENT SENDING IT FORWARD");
       sendToBlock("OSGIEVENT", event);
   }
  };

  String[]  topics = new String[] {
   this.registerEvent
  };
  Dictionary<String,String[]> props = new Hashtable();
  props.put(EventConstants.EVENT_TOPIC, topics);
  ctx. registerService (EventHandler.class.getName(), handler, props);
}
```

This can be done as an interface implementation as well, but is implemented like this to easily display what is needed to listen to a specific event. Inside the *handleEvent* action there is a *sendToBlock* method, this is a RB method to support internal system block events. [34] As for *this.registerEvent*, it is an instance parameter for this particular block and is registered within a *String* table, which is the table holding the channels this *Event Handler* should subscribe to.

A.2 OPCUA Client DS conversion

When working with DS there is another important file added to the soup. Its just not the MANIFEST.MF file, 2.2.4, any more. It is a component.xml file which is added and controls the configuration of each component service. See config A.2. Some important settings to notice, is the configuration-policy which is set to require. This is to avoid an arbitrary OPC-UA client being spawned at startup, if no configuration file exists. The rest is pretty self explanatory, implementation class, what service it provides and what service it which to reference.

Code snippet A.2: OPCUA Client DS configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    configuration-policy="require" immediate="true" name="ntnu.opcua.client.service">
    <implementation class="ntnu.opcua.client.service.OPCUAClientCreator"/>
    <service>
        <provide interface="ntnu.opcua.client.api.OPCUAClientAPI"/>
    </service>
    <reference bind="setNameSpaceController" cardinality="1..1"
        interface="ntnu.opcua.namespace.controller.api.NameSpaceControllerAPI"
        name="NameSpaceControllerAPI" policy="static"
        unbind="unsetNameSpaceController"/>
</scr:component>
```

The following are the important methods which makes the whole OPC-UA client spawning concept functional. As mentioned before, the reason the change from *ManagedServiceFactory* was done, was because of the *NameSpaceControllerApi*. It was difficult to append to the client when using a simple bundle activator which had to instantiate a OSGi *ServiceTracker* to handle the *NameSpaceControlleApi* properly. But using DS this became a breeze as can be viewed in A.3. Another important factor here is that the activate and deactivate method is initiated with a java *Map* variable. This implyes that this component is instantiated through *Configuration Admin*. As mentioned before *FileInstall* is used, which communicates with *Configuration Admin*, 7.1. The configuration files which the hardware bundles create is created with the same PID as the recent component configuration has declared in the *name* property. This creates the connection so that *Configuration Admin* understands that it is going to initiate this particular component with the configurations within the file that the hardware just created. And example of such a file name is: *ntnu.opcua.client.service-fartSensor.cfg*

Code snippet A.3: OPCUA Client DS activation class

```
public void setNameSpaceController(NameSpaceControllerAPI nameSpaceApi) {
    this.nameSpaceApi = nameSpaceApi;
}
```

```
public void unsetNameSpaceController(NameSpaceControllerAPI nameSpaceApi) {
    this.nameSpaceApi = nameSpaceApi;
}

public void activate(Map<String,String> config, ComponentContext context) throws
     Exception {
    this.context = context;
    if(config.get("hardware.name") == null){
        throw new Exception("MISSING NEEDED CONFIGS");
    }
    else {
        this.clientName = config.get("hardware.name");
        this.client  = new OPCUAClient(this.clientName,this.nameSpaceApi);
    }
}

protected void deactivate(Map<String,String> config, ComponentContext context) {
      this.context = null;
      this.client.shutdown();
      this.client  = null;
 }
```

A.3 DS Configurations to the model bundle

The datamodel is dependent of multiple services to connect the appropriate channels between OPC-UA and the hardware. It is currently dependent of the *ServerService* to append its model into the server address space. The *NameSpaceController* service to be able to append the current name space index which is created within the model creator. And it needs the *Hardware* service to be able to connect OPC-UA with the hardware.

When dependent of this many service the DS configuration files becomes a little more extensive than the latter displayed. As the configuration code A.4 reveal, it reference three different service. But there is on different setting here, and that is the cardinality and policy of the *FartSensor* service. It is [0..1] and has policy dynamic. If the settings were [1..1] and static, each time the hardware was updated, meaning stopped and started again, it would unset all the different services. Which is unnecessary because they do exist and function well. Therefore the hardware service has been declared this way, to be able to update itself without disturbing the other services.

Code snippet A.4: FartSensorDataModel DS configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    name="ModelFartSensorDriver">
```

```
    <implementation class="model.fartsensor.ModelFartSensor"/>
    <reference bind="setServerApi" cardinality="1..1"
        interface="ntnu.opcua.server.api.ServerApi" name="ServerApi" policy="static"
        unbind="unsetServerApi"/>
    <reference bind="setFartSensor" cardinality="0..1"
        interface="hardware.driver.api.HardwareAPI" name="HardwareAPI" policy="dynamic"
        target="(hardware=fartsensor)" unbind="unsetFartSensor"/>
    <service>
       <provide interface="org.osgi.service.event.EventHandler"/>
    </service>
    <property name="event.topics" value="ntnu/opcua/server/STARTED"/>
    <reference bind="setNameSpaceController" cardinality="1..1"
        interface="ntnu.opcua.namespace.controller.api.NameSpaceControllerAPI"
        name="NameSpaceControllerAPI" policy="static"
        unbind="unsetNameSpaceController"/>
</scr:component>
```

The reason that unset might be a unwanted feature, is because sometimes a form of cleaning up would be crucial, meaning that some services might be ordered to stop to let the application function. Therefore it would be a more ideal situation if the hardware service were to come and go as it pleased, and the handling of its absence would be managed by the datamodel.

Appendix B
Communications

B.1 Backward compability confirmation from Jo Skjermo

Gmail by Google

Snorre Edwin <lottwin@gmail.com>

Browse intern data

Jo Skjermo <Jo.Skjermo@sintef.no> Fri, Nov 1, 2013 at 9:02 PM
To: Snorre Lothar von Gohren Edwin <snorre.edwin@gmail.com>

Ja, klienten er der for andre stasjoner (men en server som skal brwse i et litt kompleks navnetom trenger "støtte for klientfunksjonalotet"), og eventuelt gateways (for eks for å kunne bruke "out of the box" OPC-classic til OPC-UA gateways for å koble inne "gamle" sensorsysyemer/typer

Jo

Sendt fra min iPhone

Den 1. nov. 2013 kl. 13:19 skrev "Snorre Lothar von Gohren Edwin" <snorre.edwin@gmail.com>:

[Quoted text hidden]

Figure B.1: Backward compatibility confirmation

B.2 OSGi discussions on IRC

During the thesis IRC[12] has been used to converse with OSGi users for tips and tricks. One channel which has been very helpful is the *#karaf* channel on *irc.codehaus.org:6667*. The idea to the OPC-UA client spawning solution came from Jeff Goodyear which was kind enough to discuss this situation, B.1.

Code snippet B.1: A chat with Jeff Goodyear on IRC

```
' [15:23:35] <jgoodyear> Cool
[15:23:45] <jgoodyear> Whats your question?
[15:25:05] <Snorre> Quick intro to the sitation: I have to use a framework for my thesis
    which creates a system, but bundles it inside one bundle. This cannot be changed. For me
    to be able to create the modularity I want i need to create different instances of a
    spesific service, hence a serviceFactory
[15:25:31] <jgoodyear> ManagedServiceFactory ?
[15:25:35] <Snorre> but what I need is that two different classes from the same bundle will
    trigger the serviceFactory to generate a new instance for me
[15:25:53] <Snorre> So this is possible with managedServiceFactory?
[15:26:02] <jgoodyear> thats what youre looking for in OSGi
[15:26:15] <jgoodyear>
    http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/ManagedServiceFactory.html
[15:26:20] <Snorre> ok, I have not gotten to that yet
[15:26:42] <jgoodyear>
    http://felix.apache.org/documentation/subprojects/apache-felix-config-admin.html
[15:26:55] <jgoodyear> :)
[15:26:57] <jgoodyear> No worries
[15:27:06] <jgoodyear> OSGi is a large framework
[15:27:22] <jgoodyear> I took a full year reading and digesting the specification and
    compendium to it
```

Appendix C
Setup

C.1 Eclipse setup

First of all the Eclipse has to be downloaded from [42], choose Indigo to be able to run RB.

RB has to be installed through Bitreactive and they have a start up process which needs to be followed. [31]

Then the second thing to do, is to set an OSGi environment for your development space. This can be done through different means but the solution this thesis follow is to define a target platform for eclipse. By doing this its easy to handle the dependencies that is needed for the project. This is not necessary if using the GitHub code, because all the dependencies and the target platform used in the thesis is part of the repo. BND Tools[8], is also a way to set up an OSGi development environment. It allows for support of other functionality, such as MANIFEST.MF generation, dependency evaluation before runtime, and it was used to generate bundles of of existing JARs, section 6.1.

First of all create a clean project, not technology specified, and add a folder plugins within that folder. Here the dependencies can be added. Also the OSGi container jars need to reside here. After that is done, create a new other->target definition within the folder, and add the plugins folder as a location in the target definition. Then its safe to click "Set as Target Platform" in the upper right corner.

When this is done the project need a run configuration based on OSGi technology. Go to "Run configurations" and create a new OSGi framework configuration. Here its possible to cross of the wanted projects from the current workspace and the wanted dependencies from the plugins folder in the target platform.

Starting arguments may be added into the arguments tab, these can be for example bundle properties when you depend on bundles from Apache. As the OSGi

cache property mentioned in section 9.3.2.

C.1.1 Logging with the current project

The current project depend on a lot of logging. So this should be set up properly. It is dependent of log4j and slf4j. So appended to the plugin folder the log4j jar file and two files from slf4j called *slf4j-log4j12-1.7.5.jar* and *slf4j-api-1.7.5.jar*. Now all the dependencies are up and running. Second there has to be added a log4j.properties file.

Code snippet C.1: Log4j properties

```
### set log levels - for more verbose logging change 'info' to 'debug' ###
log4j.rootLogger=DEBUG, R
log4j.logger.console = DEBUG, stdout


### log messages to logfile###
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=C:/Prosjekter/masteroppgave/opcserver.log
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d{HH:mm:ss,SSS} %p %t %C - %m%n

## direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.EnhancedPatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} [%t] %-5p %C{1}:%L
    %X{username} - %m%n

#log4j.logger.com.prosysopc.ua=WARN
#log4j.logger.org.opcfoundation.ua=WARN
```

These properties is can of course be changed, and that absolute path should be changed. The property file now splits ut DEBUG and INFO so that all the OPC-UA server logging will be put into a file that we dont have to worry about in the console.

Lastly to get everything connected this argument have to be added to the run configuration arguments:

Code snippet C.2: VM arguments

```
-Dlog4j.configuration=file:${workspace_loc}/equinox/log4j.properties
```

C.2 Properly retrieving master thesis code and importing into eclipse

After the eclipse has been set up the code can be pulled from the writers GitHub[21]. The repo is private and to get access one has to contact the writer through the contact information on the GitHub account[21].

Once the code is pulled, the next step is to import the code into eclipse. This is easily done through import and find the folder containing all the projects. When this is done, and the later instructions, the development environment should be ready to use. Right now, all the dependencies are attached to the GitHub repo, meaning that the target platform file will be achieved through the existing code. Therefore, creating a target platform is not necessary.