DESIGNING A PLATFORM FOR NETWORK-RELATED EDUCATION

5. Semester bachelor project report

Group members: Sigurd Traberg Moth (Simot18@student.sdu.dk)

Supervisor: Morten Hansen (moh@mmmi.sdu.dk)

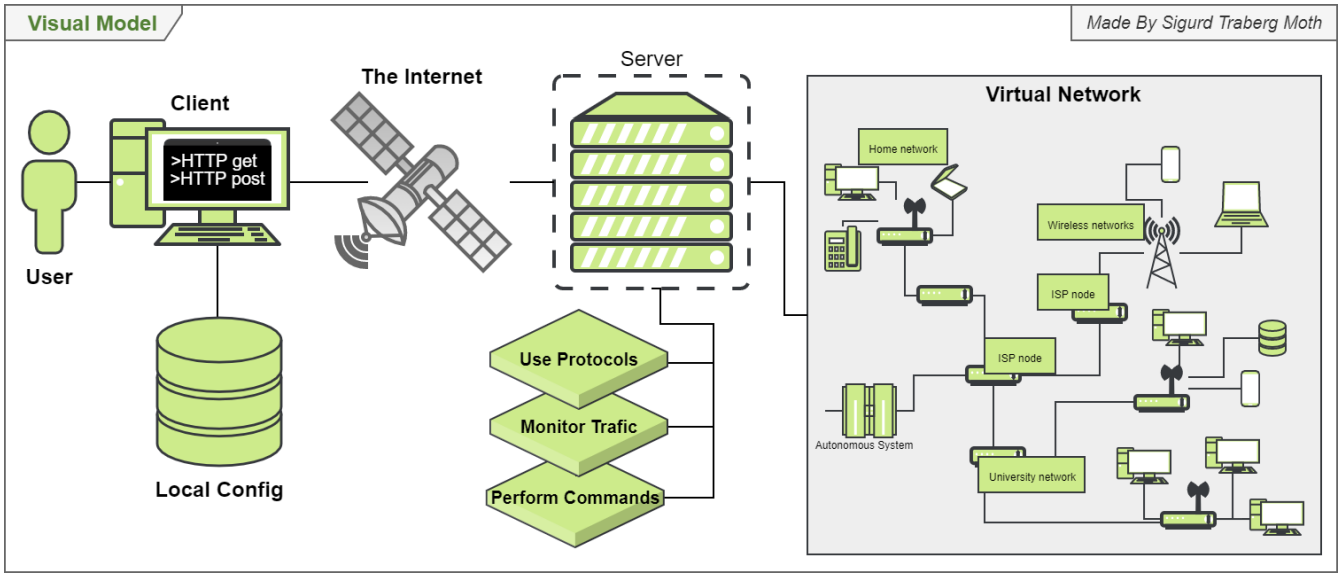
Execution period: 01. September 2020 - 04. January 2021

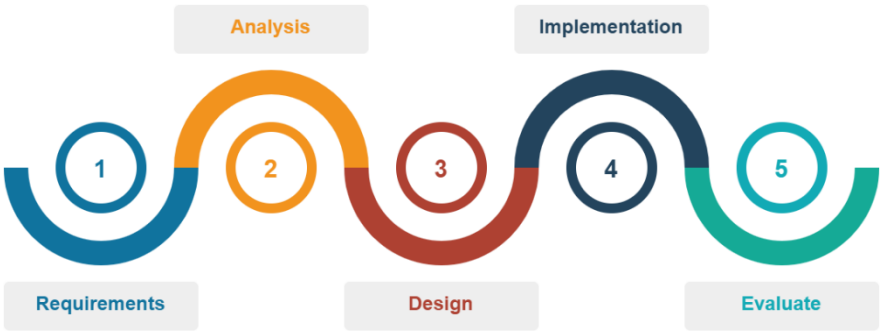
Pages: 39 pages (excluding attachments)

Course: Bachelorprojekt T510012102 (15 ECTS)

GitLab: https://gitlab.sdu.dk/HDCL/virtual-internet-platform-vip

Video: https://www.youtube.com/watch?v=hlTjT6tjulk





SDU

Syddansk Universitet Odense

ABSTRACT (ENGLISH)

In this project, a software architecture for a platform about networks is designed and created. The platform named Virtual Internet Platform (VIP) attempts to be a software that students of network-related educations can use to explore how the Internet works. The VIP is designed as a fully virtual Internet with the end goal, though out of scope for this project, of sharing all the same functionality that the real Internet has. This includes network topology, the protocol stack, routers, switches, hosts, traffic monitoring, congestion, and all the other features the Internet offers. Additionally, the VIP, being fully virtual and without merit to security, is set up to be designed with comprehensive monitoring systems for users to test and oversee network transactions with the goal of even testing other softwares' network flow.

The VIP is self-made instead of using existing products and one goal is that SDU might continue the development, of the system outsourcing the software to coming students as projects. There is therefore an enormous focus on modularity, extendibility, simplicity, and maintainability as the project will switch hands many times through its development period. Adapting to the requirements, several tools and design patterns are used, among the most important being Spring Boot, strategy patterns, and a layered microservice architecture. The delivered product in this report is a well-designed barebones software that takes user input and interprets commands through the input, performing actions such as "HTTP get" or "HTTP post" in a network of nodes.

Finally, the report covers the process of how requirements and analysis shaped the design which was then implemented in Java code and uploaded to GitLab. Much documentation, primarily requirements, and diagrams have been created in the development of the VIP and are located in separate documents, namely the Software Requirements Specification (SRS), the Software Architecture Document (SAD), and the Manuals document.

ABSTRACT (DANISH)

I dette project er en software arkitektur for en platform som omhandler netværk blevet designet og udviklet. Platformen, navngivet Virtual Internet Platform (VIP), forsøger at være en software som studerende på netværksrelaterede uddannelser kan bruge til at udforske hvordan at Internettet virker. Platformen er designet som en fuldt ud virtuel version af Internettet med slutmålet, dog ude af rækkevidde for dette projekt, der deler den samme funktionalitet som Internettet har. Dette inkluderer netværks topologi, protokol stakken, routere, switches, hosts, traffik overvågning, køteori, og alle de andre funktioner som Internettet tilbyder. Da platformen er fuldt ud virtuel og sikkerhed ikke er nødvendigt for brugeren, er det muligt at udvidde platformen med massive overvågningssystemer som brugere kan anvende til at teste og forstå netværkstransaktioner. Dertil er der også et mål om på et tidspunkt at kunne tilkoble dataflow fra andre software for så at kunne teste deres belastning m.m.

Platformen er udviklet selv frem for at låne færdige produkter med det mål at SDU muligvis fortsætter udviklingen af systemet i fremtiden. Planen er at gøre dette ved at outsource projektet løbende til nye studerende som diverse projekter. Der er derfor et stort fokus på platformens modularitet, udvidelighed, simplicitet samt vedligeholdelsesevne, da projektet løbende vil skifte hænder mange gange i løbet at produktets udviklingsperiode. For at tilpasse softwaren til kravene, er flere værktøjer og design strategier blevet brugt. Heriblandt er de mest nævneværdige, Spring Boot, strategy pattern og en lagdelt microservice arkitektur. Det afleverede slutprodukt i denne rapport er en veldesignet barebones software, som tager bruger input og gennem det fortolker kommandoer så som at ”HTTP get” og ”HTTP post” i et netværk af enheder.

Den endelig rapport dækker udviklingsprocessen af hvordan at krav og analyse var med til at skabe designet og implementationen i Java kode som derefter er blevet uploadet til GitLab. Meget af dokumentationen, primært krav og diagrammer, er blevet lavet i udviklingen af platformen og er blevet gemt i følgende dokumenter, Software Requirements Specification (SRS), Software Architecture Document (SAD), og Manuals dokumentet.

PREFACE

Stakeholders

The primary actor is Morten Hansen who can be considered the customer, co-creator, and supervisor of the project. Morten wishes to continue developing the software possibly through outsourcing jobs to students. At some point, Morten might use the software for his teachings if it manages to become mature.

As hinted above, the end goal is for students to be using the software to study networks. The software must enable them to learn effectively so the quality of their education improves, and students do not have to worry about stability and more. These are also the students to which he will outsource work.

Finally, I, the author and creator, am also included as a stakeholder since I am expected to prove that I have been taught well through my 3 years of software engineering education. This will undoubtedly influence my work as I will try to prove myself through documentation and implementation.

Audience

This document is intended to be read by the examiners including Morten Hansen. The document will also be available through the repository at GitLab for developers to read to get a better understanding of the software, but in general, they should be using (and updating) the Software Requirements Specification (SRS) and Software Architecture Document (SAD) instead.

Timeframe, Prehistory, and Project Size

This document contains the documentation of a 5th semester bachelor project for Software Engineering at the University of Southern Denmark (SDU). The bachelor project is estimated to 375-450 hours or 15 ECTS over the course of 1 semester which is about 5 months. There are more hours allocated to the project per person, but the project is also down to one person from the usual six. This means that the total project size is also 375-450 hours instead of the usual 1500-1800 hours. Additionally, the project was moved from 6th to 5th semester and therefore the report lacks input taught through the course “Project Organization & Management” (course code: T510010101) which is aimed at writing bachelor reports.

Recognition

I would like to thank Morten Hansen for his part in the project. He has been a great aid and I was very pleased when he accepted to take on my project as we had only talked about a possibility for a 6th semester project. But he was more than happy to change up his plans as late as early September and this truly saved me as I was originally supposed to go abroad, and we did not have a specific idea in mind at the time. I would like to thank Samuel Bangslund, a classmate and previous consecutive group member. He acted as a sparring partner and was of great help as I had little knowledge of the frameworks that I worked with, in the earlier stages of the project. I would also like to mention and thank all the professors that along the way have helped me to achieve the level of skill that I now possess. Especially Lone Borgersen’s teachings, I have taken to heart.

About the author

I am a 5th semester Software Engineering student with limited experience in networking. I took an additional course (Data Communication) on 3rd semester and through the course, I got to know Morten Hansen. I primarily work in Java and have recently picked up an interest in the Spring framework and through this project, I have tried to use it and implement it to the best of my knowledge. The software and all the associated documents have been created by me.

Signature: 
Sigurd Moth

READING GUIDE

After reading this document, the reader should be able to comprehend why the project was created. The reader should also be able to roughly understand what the project is about, the size of the project, and the timeline. This reading guide is supposed to give a general idea of what each chapter is going to cover.

Pretext

Abstract

A summary of the project as a whole describing the goals and what was achieved.

Preface

The preface gives a rough understanding of the most relevant **stakeholders**, who are the **audience** for the report, **recognition** of people involved besides the writer, and signatures of contributors.

Body Text

The body text consists of 10 chapters and tries to enlighten the reader as to what, why, when, and how the project was made. The chapters follow the workflow structure of the Unified Process (UP) (Jim Arlow, 2009). Each chapter contains the following:

A. Introduction

An introduction, the background, the problem statement, and a problem verification.

B. Requirements

A walkthrough of the functional and non-functional software requirements gathered from the SRS.

C. Analysis

A walkthrough of the overall system architecture, primarily reasoning between options with pros and cons on a very high level.

D. Design

A walkthrough of the chosen implementation of the system architecture.

E. Implementation

Some examples of code with belonging explanations of how the code works.

F. Discussion

A section full of discussion based on ideas and thoughts that were made throughout the project.

G. Conclusion

A conclusion upon how the project went. A direct answer to the problem statement.

H. Perspective

A perspective into what the system could achieve going forward.

I. Bibliography

A list of sources that were used in this project.

J. Attachments

A collection of documents, diagrams, and larger illustrations that did not fit directly in the report.

TABLE OF CONTENTS

A. INTRODUCTION

This chapter seeks to introduce the project as well as the reason why the project was deemed relevant and worth exploring. The introduction starts with a general introduction and then expands on the background of the issue, problem verification, and the creation of a problem statement. The scope for the project is then set, a timeline is made, and an idea of the final solution is imagined.

A.1 Introduction

This whole project started with a loose idea of "What if we could make our own 'the Internet'?". It was already September and due to some unforeseen circumstances, a project idea had not been planned yet. Through a course, I had gotten to know Morten Hansen, a Data Communication lecturer among other things, at SDU. We had earlier talked about doing a project together and I, therefore, booked a meeting, and with him, we brainstormed a bunch of ideas. As we talked, we went over the subject of how a new lab for network students had been created, and that they needed a way of monitoring network traffic in a closed-off environment. We thought that it would be interesting to explore creating this system ourselves instead of borrowing software, and we, therefore, set to investigate the idea further. This would also allow Morten to present more opportunities for projects to other students through the system which they could then modify and extend upon. Through this meeting, the idea of a **Virtual Internet Platform** hence known as the **VIP** was formed.

This project could be thought of as the first phase of a larger project. The whole idea revolved around experimenting and seeing if creating such a platform was doable and would make sense. The goal in other words was not to have a perfectly optimized and working platform by the end of the project, but more so to have some kind of proof that we were heading in the right direction. Personally, this was great. I had recently piqued an interest in the Java framework Spring Boot and had found the framework quite exciting. I wanted to put it into use, and that is best done through creating something new such as a standalone platform. The project idea also held the promise of allowing me to delve deeper into networking which I find very appealing. An exciting project, but also a daunting task.

A.2 Project Background

This section describes the motivation and reasoning behind creating a networking platform and what problems the platform tries to solve.

A.2.1 Motivation

The primary personal reason for working on such a project has been to learn more about networking. It was also one of the reasons I chose to study the course Data Communication. Through the project, I hope to gain a better understanding of how complicated the Internet is and hopefully some kind of experience that will make me attractive to the job market going forward.

I also hope to gain some experience in working with modifiability and extendibility as these are elements, I have often skimped upon due to time constraints. Finally, I want to try and put the Java framework, Spring Boot, into use, and it is hard to find a better way to do that than to start all over with a new platform architecture.

A.2.2 A Solution to Education in the Network Lab

One of the things that the project tries to solve, is that the network lab at SDU wants a software that makes it easier to explore how networking works. This is best done in a setting where the students have complete control over the Internet. This is of course something that then cannot be done on the Internet as there will be a lot of factors such as traffic, network stability, router setup, etc., that will manipulate the data. A

platform is needed that can run locally on a server or cluster and where the node setup and traffic can be managed completely isolated from the real world. Additional monitoring options are also needed which, often due to security, are not available on the Internet. In short, creating our own VIP, allows us to control every variable and thereby draw proper conclusions in any test we make (be it a load, stability test, etc.).

A.2.3 A Solution to Health Data Communication Lab (HDCL)

Morten is also in charge of another project that goes by the name Health Data Communication Lab (HDCL). In this project, they are going to manage a large-scale database server full of patient information. As the database is on a large scale, load testing is going to be important. Additionally, patient data is personally identifiable and very sensitive (GDPR). Therefore, stability and security are going to be very important for non-functional requirements. The HDCL project is going to have its requirements tested, and this could potentially be done through the VIP software when it has matured. Similar projects could also arise asking for the same kind of testing which the VIP could then offer.

A.2.4 Project Exploration

The VIP also opens up the possibility for students to do projects. When the platform has been specified it will be possible for students to create and add protocols, monitoring, and other subsystems to the VIP. For that to be possible, or at least more efficient, it requires that the VIP is designed to be very modifiable and extendible.

A.3 Verifying the Problem

Through A.2 Project Background it has been established that there are problems that can be solved with the VIP. The question now becomes, is creating a new platform the best way to solve it? Or are there already existing alternatives?

If the goal is just to present a system that can be used by students to produce and track networking traffic and vulnerabilities, then such platforms exist such as Cisco Packet Tracer. The software is very mature and is used much throughout the industry. A couple of issues arise when using vendor software. For one, you only have the features that the software is released with, you cannot necessarily extend it yourself. It will also not be very specific to the software at hand such as the HDCL project. Software like Packet Tracer also does not support multiple users pinging each other and “node takeovers” which can be interesting and a fun experience for students exploring networking. These softwares are more for creating network topologies and testing system-wide traffic without user control.

Secondly, vendor products do not provide the desired tools for exploration as well as the possibility of doing projects together with students. There is a lot of potential experience to be had through creating the system instead of using finished products.

One could argue that there is space for both solutions in each case but that a self-made system could work with both the HDCL and student projects though far more time is required because of the steep curve in the development of such a system before it has matured. Depending on the urgency of the HDCL this could be deemed acceptable, or Packet Tracer or other alternatives could be used until the VIP has matured.

A.4 Problem Statement

The final section of the introduction contains a problem statement that aims to specify to the reader what problem the project tries to solve. The problem comprises of the main problem further specified through additional subproblems.

The **main problem** is as follows:

- How can we design a platform architecture that allows for teaching students networking principles effectively?

The **subproblems** are as follows:

- How can we make the platform architecture maintainable and modifiable so that it is easy to hand off the project to coming students that wish to further expand on the system's capabilities?
- How can we design monitoring systems that allow students and professors to oversee network traffic as well as improve the ability to debug system errors?

A.5 Scope

This section will shed some light on elements that are not going to be included, this time around.

A.5.1 Scale and the Future

The expectation is that the final software will be quite big stretching over a development period of several years. This specific project will maintain focus on finding and creating a strong architecture and less so on the implementation of protocols and the management of nodes (routers, switches, hosts, etc.). For the protocol stack, the layers below the application layer will be ignored as well. The previous points make it clear that there is an expectation that this specific project will be finished somewhat open-ended. And even though the focus will be on creating an architecture it will still be necessary to look into all the requirements that will shape the final system at least to some degree as to not oversee potential architectural design problems. To help further develop the software, manuals will be available to make the transition easier for developers.

A.5.2 Tool Complexity

Early on it was decided that the software must be easy to hand off to new developers. Using a lot of tools is a double-edged sword as it makes the software more complex. In this project, the tools selected should primarily lower the barrier of entry for other developers to work on the project. Some wanted characteristics would be code language independence, automation, and a strong architecture that allows for easy expansion.

A.5.3 Command-line interface (CLI)

For this project, there will not be a user interface as it would be too time-consuming to also implement for one person. The original idea of the final product would be that as more methods and protocols are implemented, then these will use a (potentially) React-based user interface that is easily coupled up to the CLI API. Users will then be able to choose which interface they will use.

A.6 Proposed Solution

With this chapter in mind, imagining what a solution might be like (a rough sketch), is a good idea before looking at requirements. This section includes a description of the final product and this project's product.

A.6.1 The Final Product

The general idea is that the final system will be a simple, virtual copy of the Internet for students to use and learn from. It is going to be mimicking a lot of the functionality that the Internet has to offer, but primarily for educational and research-related purposes instead of functional purposes. The overall system is going to comprise of several components including microservices, that combined satisfy the problem statement. Some of these components are going to be related to real devices such as routers, hosts, switches, etc. Componentizing the system is going to make it less of a hassle to work on as microservices

can be exchanged, changed, debugged, and iterated on independently of other parts of the system. Componentization is not going to directly change the functionality of the system, but rather simplify the development of the system and allow developers to develop their own parts of the system without affecting other developers. The product is going to include most if not all protocols that exist in the protocol stack which are going to exist as independent microservices just like nodes in the network allowing for a familiar architecture. This means that a user will be able to send a message to a host in the network through HTTP, which will then go down through the stack and up and deliver a message to another host. Finally, nodes in the network will be alive as devices on the Internet. Routers will be able to update their routing tables depending on other devices, load balancing, and if devices shut down unexpectedly.

A.6.2 This Product

With the final goals out of the way, this project more specifically tries to find and create the said architecture that the final system will be based upon. The plan is to reach a point where the product can be handed off to other developers to develop protocols, commands, and additional microservices. To get to this point, a client service will have to be developed that can be used to interact with a server containing the virtual network. To get all the additional data that is necessary for monitoring the virtual network, there will have to exist some kind of additional microservices in between the client and the network which are going to be managing the client data as well as having access to the virtual network. Finally, it would be great if the system could be transferred to Docker containers to make the software independent of both platforms and machines so ex. students could access the virtual network from other machines. It is worth remembering that it is not clear how far this project will go, as it will try to go as far as possible, but as of now, this is a good estimate of what the current goal is.

A.7 Software Process: Unified Process

This project follows the methodology of the software engineering process Unified Process (UP) based on Arlow's interpretation. UP is a process known to be use case and UML focused, and "defines the who, what, when, and how of developing software" (Jim Arlow, 2009, s. 28). UP consists of 4 phases through any given project: **Inception**, **Elaboration**, **Construction,** and **Transition**. UP is iterative in the sense that every phase repeats certain workflows multiple times, also known as sprints in Scrum. These workflows consist of reiterations of **Requirements**, **Analysis**, **Design**, **Implementation**, **Test,** and **Integrate** (Figure 1). The diagram below reflects the expected workload in each activity changing through the phases. E.g., UP projects focus requirements and move the focus to implementation as they become more tangible. Finally, to conform with the standard of UML, every diagram, except this figure below, has been made in DrawIO.

It should be noted that the iterative approach of UP makes it quite difficult to write a well-structured report given that a report by its nature is chronological.

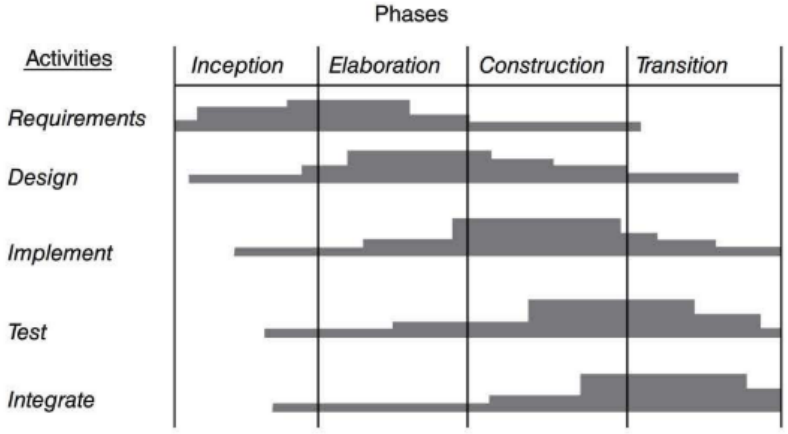


FIGURE 1 - A DIAGRAM OF UNIFIED PROCESS, ITS PHASES, AND ITS ACTIVITIES. (FRANK TSUI, 2018)

A.8 Timeline

The timeline diagram (Figure 2) contains important milestones found throughout the project and is based on UP. Milestones are found early in the Inception phase and are used to create a sense of urgency and deadlines which can enhance productivity in some cases. It also creates a stronger sense of direction.

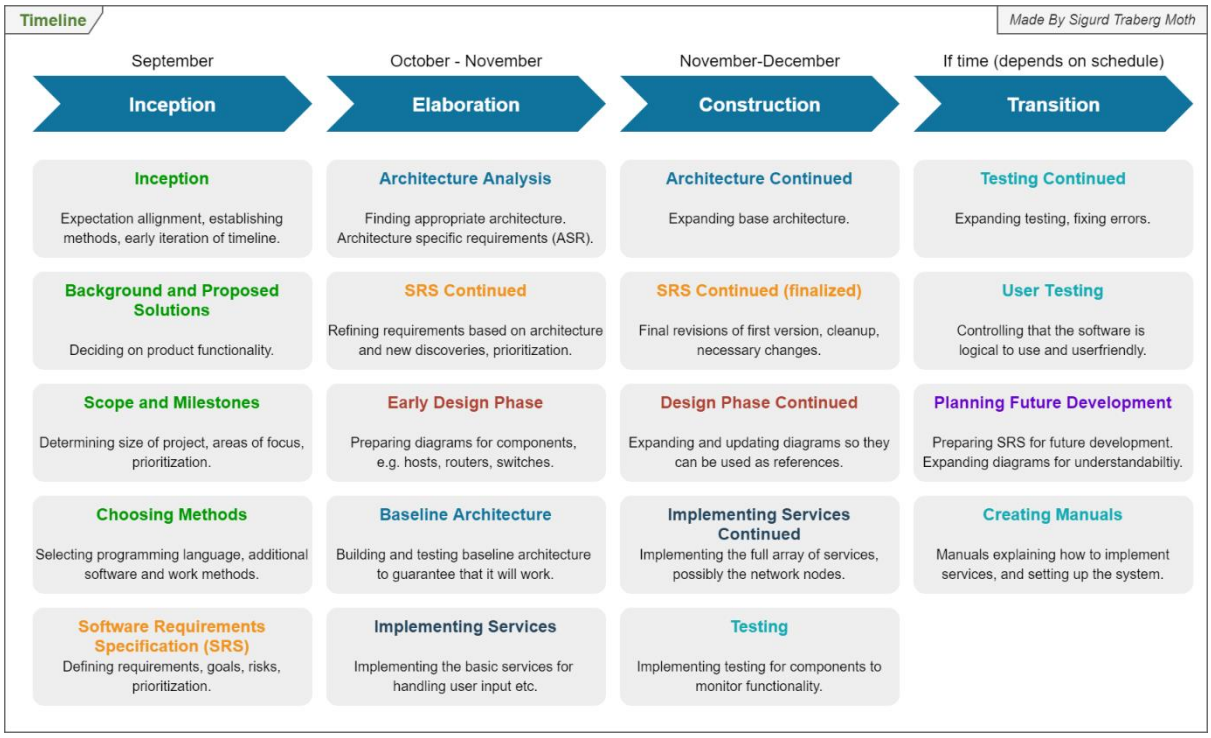


FIGURE 2 - TIMELINE DIAGRAM

As with any plan, they do change as time goes on. The above diagram reflects the project very well except that testing for the most part had to be left out. Services are also not very refined.

The primary artifacts produced are as follows:

- INCEPTION DOCUMENT
- PROJECT DESCRIPTION HANDIN TO SDU
- SOFTWARE REQUIREMENTS SPECIFICATION (SRS)
- SOFTWARE ARCHITECTURE DOCUMENT (SAD)
- REPORT (MAIN DOCUMENT)
- MANUALS
- SOURCE CODE IN GITLAB REPOSITORY.

B. REQUIREMENTS

Properly specifying requirements is considered one of the most important parts of creating good software. Properly specifying requirements ahead of building a software is time-consuming but it is also often worth it in the long run as architectural changes become exponentially more expensive to perform.

B.1 Software Requirements Specification

In this project, all requirements have been documented through the Software Requirements Specification (SRS) based on a mix of UML 2 and Unified Processes (Jim Arlow, 2009) and Jan Corfixen's SRS (Nikolaidis Pavlos, 2011). It is expected that the SRS will change and become more refined as the project moves forward but version 1.0.0 is referenced here in this report.

The SRS contains 3 primary elements. The functional requirements, non-functional requirements, and a domain diagram. The functional requirements will be found through the use case model comprising of a list of actors, use cases, and a use case diagram.

Additionally, the nonfunctional requirements will be found through business goals leading to Architecture Specific Requirements (ASRs) and related tactics. Nonfunctional requirements follow the book Software Architecture in Practice (Len Bass, 2013).

Finally, a domain model is made through a domain diagram based on the domain dictionary found in the SRS. It is important to understand that the workflows of UP are repeated several times, and therefore some requirements will be dependent on the domain model and vice versa.

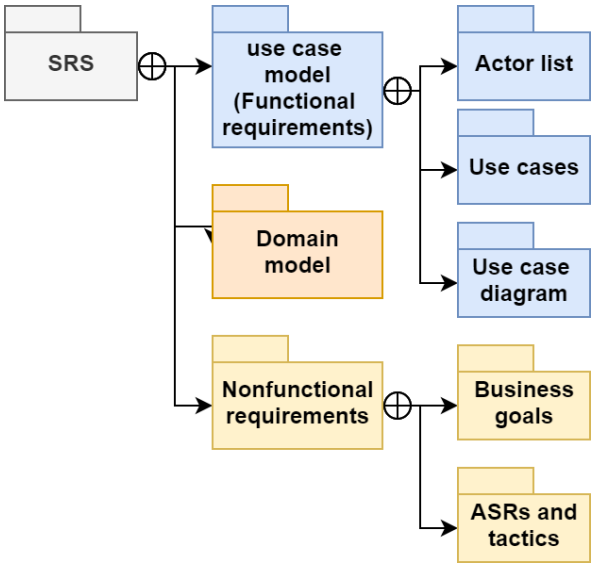


FIGURE 3 - SRS STRUCTURE

B.1.1 Use Case Model

Before diving into how the system benefitted from the use case model, it should be noted that the focus was primarily on the nonfunctional requirements due to the fact that the platform architecture was the primary goal. This also changed up the priorities, as usually use cases help form mostly the functional side of requirements and less so the architecture. That being said, the groundwork for the functional requirements has been made as functional requirements become a lot more important in future development and still to some degree affect the nonfunctional requirements.

B.1.1.1 Actor list

The actor list contains the type of users that are going to be using the system as well as descriptions.

A. **Teachers** are going to be using, monitoring, manipulating, and setting up the VIP for everyone else to use. They want full access to the VIP and the ability to allow others to further develop the system.
B. **Students** are going to be using, monitoring, manipulating, and setting up the VIP as well as part of their education. This will be done through exercises where a path has been laid down with hints for students.

C. **Projects** are usually groups of students that are developing a product. They might use the VIP to test or prove a hypothesis in a more controlled environment. They want to be able to create their own environment wherein they can test their ideas in isolation.
D. **Developers** are those interested in developing features for the VIP. Developers can be and do all of the above.
E. **Virtual Hosts** are bots imitating real users sending and receiving data on the network. As such they are not real users with many use cases, but rather shallow imitations.
F. **Common Users** are a generalization of teachers, students, and project users. It contains many of the use cases such as the ability to send and receive data as this is a feature these users need.
G. **Time** is an actor as several things are going to be happening on a scheduled basis.

Going forward, it is expected that most use cases are going to reside in the "Common Users" profile, as with many kinds of software, a lot of functionality is usually shared among most if not all users and this can be generalized.

B1.1.2 Use cases

A shorter list of prioritized use cases has been elicited with the help of a quick MoSCoW analysis[1] (Jim Arlow, 2009, s. 59), as listing all use cases would take up far too much space. The full list of use cases for each actor can be found in the SRS. A better way to prioritized use cases is through an Analytical Hierarchy Process (AHP), but as use cases are not as important on an architectural level, the analysis is skipped in favor of MoSCoW for now, as it is also very time consuming and often done for individual parts of a system. The use case prioritization is done to find use cases that should be detailed. Also do note that some use cases have already been separated into several use cases already indented by an additional number and not in bold (see table).

M/S/C/W	ID	Use case	
M	**A.1**	**Send data**	Common users want to send data across the VIP.
	A.1.1	Send big files	Common users want to send big files across the VIP.
	A.1.2	Send several files	Common users want to send several files across the VIP.
M	**A.2**	**Receive data**	Common users want to receive data across the VIP.
	A.2.1	Receive big files	Common users want to receive big files across the VIP.
	A.2.2	Receive several files	Common users want to receive several files across the VIP.
M	**A.9**	**Joining network**	Common users want to be able to join and interact with the networks on the VIP.
	A.9.1	Joining network externally	Common users want to be able to interact with the network as their own external network.
	A.9.2	Joining subnet	Common users want to join specific networks and to live as a part of the internal network on a subnet.
M	**A.3**	**Use protocols**	Common users want to use different protocols for different applications. (Contains many sub use cases)
	A.3.1	Use HTTP	Common users want to use HTTP to send and receive messages across the VIP.
S	**A.5**	**Displaying data**	Common users want to select what data is being logged and displayed
	A.5.1	Display protocols	Common users want to select which protocol information is being logged and displayed.
	A.5.2	Display packet loss	Common users want to select displaying the packet loss of a device.

[1] The MoSCoW analysis prioritizes use cases by importance of Must, Should, Could, Will not. It is very simplistic yet effective for creating a quick overview in a table.

	A.5.3	Filter layer information	Common users want to select to filter by layers of the internet model.
	A.5.4	Display routing network	Common users want to display the routing network so they can identify the setup and possibly join a subnet.
S	**A.6**	**Logging information**	Common users want logged information for traversing history.
	A.6.1	Logging data	Common users want to log data so they can traverse previous messages.
	A.6.2	Logging timestamps	Common users want timestamps to be paired to data.
S	**A.7**	**Getting help**	Common users want to access help where they can learn the ropes of the system.
	A.7.1	Explaining functionality	Common users want to access help and get an understanding of the system through explanations of functionality.
	A.7.2	Explaining keywords	Common users want to access help, where they can get an explanation of keywords (parameters -x) and their actions.
C	**A.4**	**Monitor system**	Common users want to monitor the VIP for various information.
	A.4.1	Monitor throughput	Common users want to monitor throughput or traffic across a given device.
	A.4.2	Monitor bandwidth	Common users want to monitor the level of bandwidth is across a given device.
	A.4.3	Monitor packet loss	Common users want to monitor packet loss.
	A.4.4	Monitor device uptime	Common users want to monitor the uptime of running (and shut down) devices.
	A.4.5	Detecting defects	Common users want to detect defects (error handling) when devices are not functioning as expected.
	A.4.6	Detecting heartbeat	Common users want to know if devices are continuously running by detecting heartbeat on devices.
C	**B.1**	**Create standard setups**	Teachers want to create standard setups for exercises and save them through configuration files. *(sub use cases in SRS)*
C	**B.2**	**Change standard setups**	Teachers want to change standard configuration setups for exercises. *(sub use cases in SRS)*
C	**E.1**	**Hook up to API**	Developers want an API to hook up new components to.

As one of the main functionalities of the VIP is for a client to interact with the Internal Network and send data, we prioritize and analyze use case A.1 “Send data” further in the next section. This is done by *detailing* the use case A.1.

B.1.1.3 Detailed use case

Having first prioritized the use case A.1, it is now detailed. Here the actor (user), pre- and post-conditions, and the flow and alternate flows are found. Other nested use cases might also be found during detailing. As development continues and more use cases are analyzed a more complete picture of requirements is formed. The use case A.1 is detailed as it has been deemed essential to the VIP and serves as an example of why detailing can be a benefactor. The activity of detailing a use case is something that is often done in parallel to doing other tasks in a project. This includes updating the domain glossary[2] and requirements.

[2] The domain glossary is essentially a glossary for words used in context with the system. The glossary is made to avoid confusing definitions of words. The full glossary is located in the SRS.

ID:	**A.1**
Use case	Send data (For example a message)
Primary Actor	Common users (a generalization of most users)
Description	Common users want to send data across the VIP.
Preconditions	1. A client must be connected to the Internal Network. 2. A client must have an address in the Internal Network. 3. A client must have a target in the Internal Network. 4. A client must have acceptable data to send.
Main flow	1. The use case starts when a user types a command in the CLI that has the functionality of sending data. 2. The client service handles the input encapsulating: a. The protocol e.g., “HTTP”. b. The command e.g., “get”. c. Optionally parameters e.g., “-t <target address>” “-m <message>”. 3. The VIP interprets the input, selecting the correct protocol. 4. The VIP sends a message from the client's address to the target address in accordance with the HTTP protocol. 5. (Optional)The client receives feedback that the message has been delivered.
Postconditions	A user has sent a message between two virtual hosts in the Internal Network. The user has received feedback that the message was sent successfully.
Alternate flows (Potential new use cases)	Wrong input: 1. A user types an incorrect command. 2. The user is given the information that the input is incorrect. Nothing happens. Address not set: 1. A user sends a message, but the user’s location has not been set in the configuration file or as a parameter. 2. The error is handled, and the user is informed that no source address was specified. Nothing happens.
Additional use cases	(A.1.1 Send big files), (A.1.2 Send several files)

Having now detailed the use case A.1 Send Data, more information has come to light, and it is now a lot easier to prepare to design and implement the functionality. To a certain extent the use case A.1 also helps shape the architecture as it is core to the functionality of the VIP. Additionally, a sequence diagram can now be created for the flow of the use case as seen below (Figure 4).

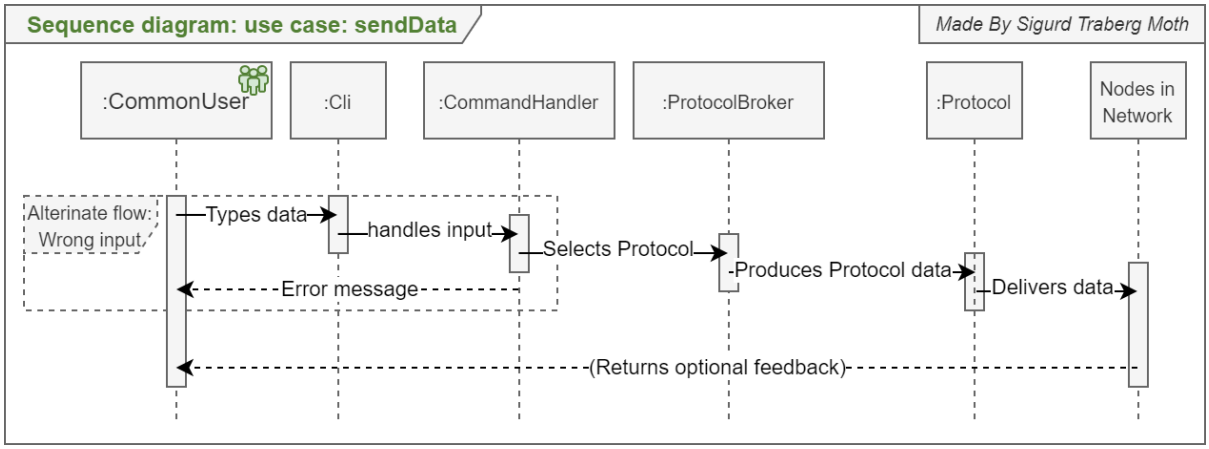


FIGURE 4 - USE CASE SEQUENCE DIAGRAM OF SEND DATA

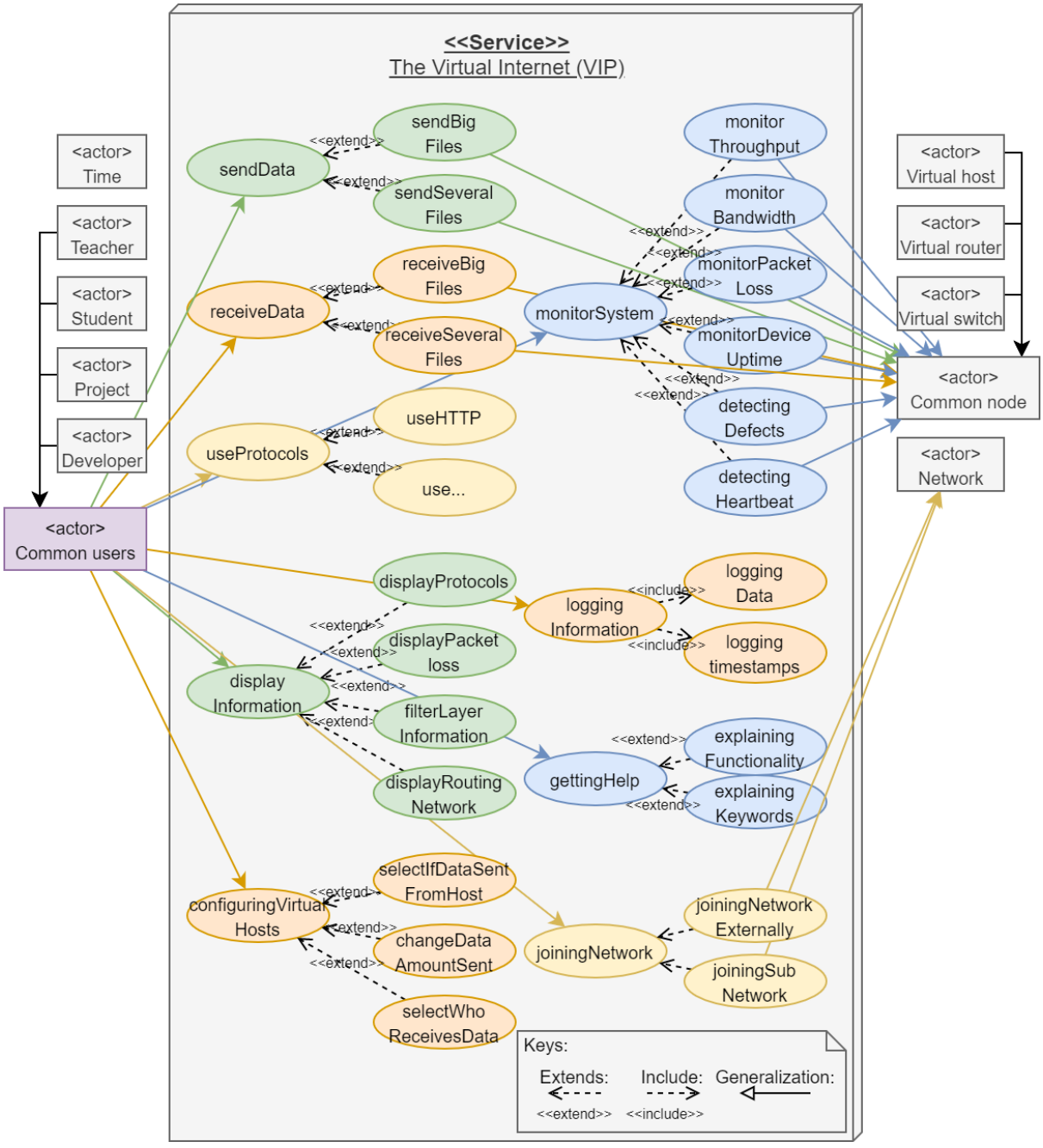
In a perfect world every possible use case for a given software is found and analyzed to the level of detailed use cases, but as with the AHP analysis time is often a limit and a deadline is always around the corner. That being said, every protocol flow should probably be detailed.

B.1.1.4 Use case diagram

After brainstorming use cases for each actor, use case diagrams can now be made. The use case diagrams are best suited for functional requirements as they are created from use cases, but they still serve to create a good viewpoint of the system and its users. Use case diagrams are continuously evolving as use cases are refined. Below, two use case diagrams have been selected to give a quick impression of how the actors "Common user" and "Teacher" interact with the VIP.

Use case diagram: Common users

The "common users" diagram is going to be the biggest diagram and will see the most development going forward. That is because most users will be generalizing functionality from this actor class. As more use cases are found and detailed, the size of the diagram increases and should be split up into separate diagrams (Figure 5).



Use case diagram: Teacher

The "Teacher" diagram is also included as this diagram show use cases related to setting up, maintaining, and adjusting the VIP. The "Teacher" diagram proves that different actors need different functionality (Figure 6).

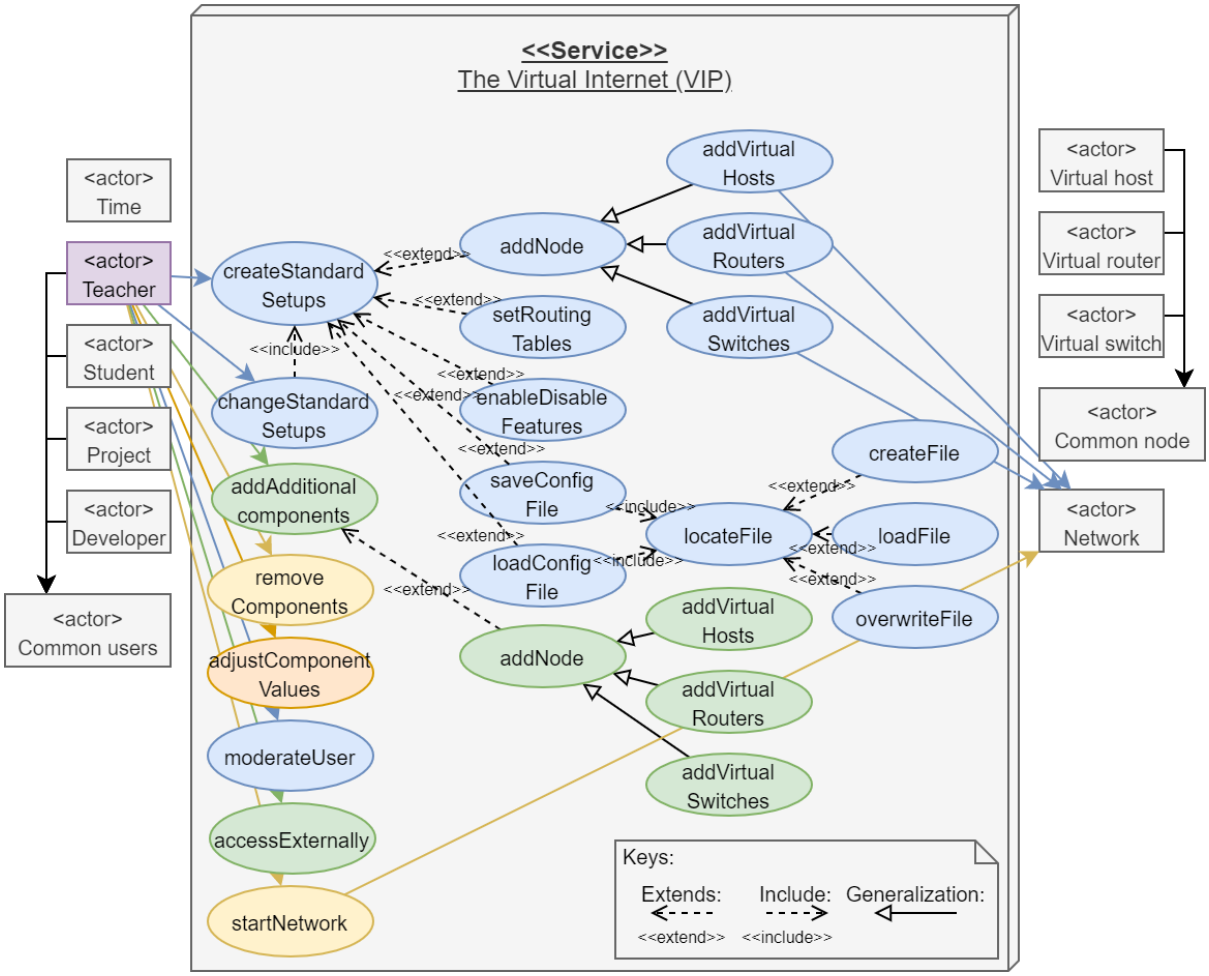


FIGURE 6 - USE CASE DIAGRAM FIGURE 5 - USE CASE DIAGRAM OF COMMON USERS

With use case diagrams in hand, it will become easier to keep functional requirements in mind. It will also become easier to discuss the functionality of actors and move the functionality between actors. Finally, it helps develop a project-specific language as sentences are used to describe functionality in conjunction with other requirement activities.

B.1.2 Nonfunctional Requirements

Having established some kind of idea of *what* functionality the VIP must have, it is now time to look into non-functional requirements to determine the best-suited architecture. Though many nonfunctional requirements are logical and are elicited directly from business goals through conversation, they are still processed and further defined with belonging Architecture Significant Requirements (ASRs) and tactics (Len Bass, 2013, s. 313). This ensures that the requirements are made explicit and that the requirements are included in the architecture when it is designed and realized and so the architecture does not become incompatible with the implementation of requirements (Figure 7).

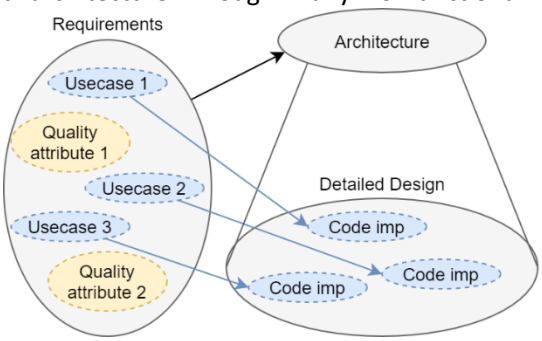


FIGURE 7 - NONFUNCTIONAL REQUIREMENTS IMPACT ON ARCHITECTURE

B.1.2.1 Business goals

Business goals are extracted from questions loosely based upon (Len Bass, 2013, s. 319) and are gathered from meetings and use cases. A business goal is defined as important if it will impact the architecture.

Personal objectives

- I want to create documentation to prove understanding of theory as the project is done through education and is rated at the examination. This will to some degree reduce my focus on the VIP implementation and SRS as making the report more easily understandable focused.
- I want to gain experience using new tools while still in a safe environment due to education. This can impact the choice of frameworks and code libraries which can increase risk.
- I want to make the VIP as easy to take over as possible, and ready for continuous development by third-party actors (developers).

Developer objectives

- (**Important**) Developers will in general have more experience with Java than other programming languages due to SDU's preference for Java in the curriculum. That does not mean students will not have their own favorite programming languages.
- (**Extra important**) It should be easy to introduce users and developers to the system. This means creating a clearly defined architecture that is easy to **modify**, **extend,** and **maintain** with an ever-switching group of developers.
- (**Important**) Developers want to have test setups of the Internal Network to easily test configurations of the VIP.
- Developers are going to need some kind of monitoring service that can track uptime, performance, and reliability of services.
- Developers are going to need a ping system that makes it possible to interact directly with services.

User objectives

- The Internal network of the VIP should be so fast that a class can be using the VIP in a networking course. Courses could contain assignments such as sending lots of data between hosts, creating network congestion, etc. These concepts should act accordingly to the Internet.
- Users want to have configurations of the Internal Network available for courses. In this way, they will be able to save, restore to a previous state, or quickly start a new assignment.
- (**Important**) Users want to be able to use their own pcs to act as clients for the whole system.

- Users need to able to check if nodes they create are available in the Internal Network.

Product quality

- (**Important**) The VIP should be reliable enough to resemble the Internet which is very reliable. The Internal Network is going to have other requirements for reliability than other services in the VIP system as the Internal Network must have the same error handling as the Internet.
- When many users are using the VIP, the system should not bug down due to repeated errors, stalling service, dying services, etc.

Based on the business goals, it is now possible to derive *utility trees* (solutions).

B.1.2.2 Utility Trees

Given there are many utility trees, the most important ones have been picked for the report, the remaining utility trees can be found in the SRS.

Utility Tree: Important

The "important" utility tree contains ASRs and tactics for the most important business goals. Quality attributes work as a way to quickly inform what area it will impact of the project and are used when picking tactics (Len Bass, 2013, s. 97).

ID	Attribute refinement	Architectural Significant Requirements (ASR) and Tactics
1	**Java programming language**	The core microservices should be developed in Java as Java is the primary programming language taught at SDU, at least currently. This will allow for a common ground for most students possible. Java also supports many useful frameworks. Quality attributes: **Modifiability**, **Usability, Compatibility**
2	**Spring Boot framework**	The core microservices will be using a combination of Spring Boot and Maven through the build tool Gradle to realize the microservice architecture. This is a choice by developers early on as the framework should reduce complexity with interfaces within each microservice. Additionally, Spring Boot provides great implementations of the RESTful services. Quality attributes: **Modifiability**, **Extendibility, Maintainability**
3	**Programming language independence**	JSON will be used to communicate between microservices. This is done to keep every microservice independent of a programming language and every input/output independent of the service itself. This means that any developer can pick up the project with their own favorite programming language and add additional microservices. Primarily protocol microservices. Quality attributes: **Extendibility**
4	**Layered Architecture**	Using a standardized way of organizing projects can improve uniformity and reduce complexity when the standard is understood by the developer. A 3-layer architecture is a common way to do this where each service is built out of a presentation, domain, and persistence layer. Presentation will handle input from other services, domain will transform data, and persistence will handle output as well as configuration data (databases or locally stored files. Quality attributes: **Modifiability**, **Extendibility**, **Maintainability**
5	**Microservice responsibility**	Every microservice will only have a single responsibility. That is, a service will be taking input, a service will be translating input, a service will be handling nodes, etc. This is done to reduce complexity and increase uniformity. Quality attributes: **Modifiability, Extendibility, Maintainability**
6	**REST / CRUD**	Microservices will benefit from using the REST architecture through the Springboot framework and JSON standard. REST will uniformize microservices and reduce complexity. Quality attributes: **Modifiability, Simplicity**, **Operability**

7	**Configuration file responsibility**	It is important that configurations can be saved so clients can maintain correct settings across multiple sessions (sittings). That does mean that these configuration files should also either be encrypted or (preferably) the code should have limits to configuration values so that users do not manually set them too high or low and crash the system. Quality attributes: **Maintainability**
8	**Device-independent**	The VIP should have the potential to be set up on different devices concurrently. This means that the workload can be spread between devices. As the VIP will be based on Spring Boot and REST this requirement will be easier to fulfill. Quality attributes: **Modifiability**, **Operability, Extendibility**

With utility trees in hand, a proper definition of nonfunctional requirements now exists instead of a vague definition that the system should be fast and reliable. There are actual goals to implement to achieve these qualities of a system and ways to do it.

B.1.3 Domain Model

Having formed the requirements, it is now possible to create a domain model of the VIP (Figure 8). Domain models are used to present systems to business relations in order to prove concepts and are great for explaining features. Words representing different parts of the system in the diagram are based on requirements, primarily through use cases, mixed with common sense.

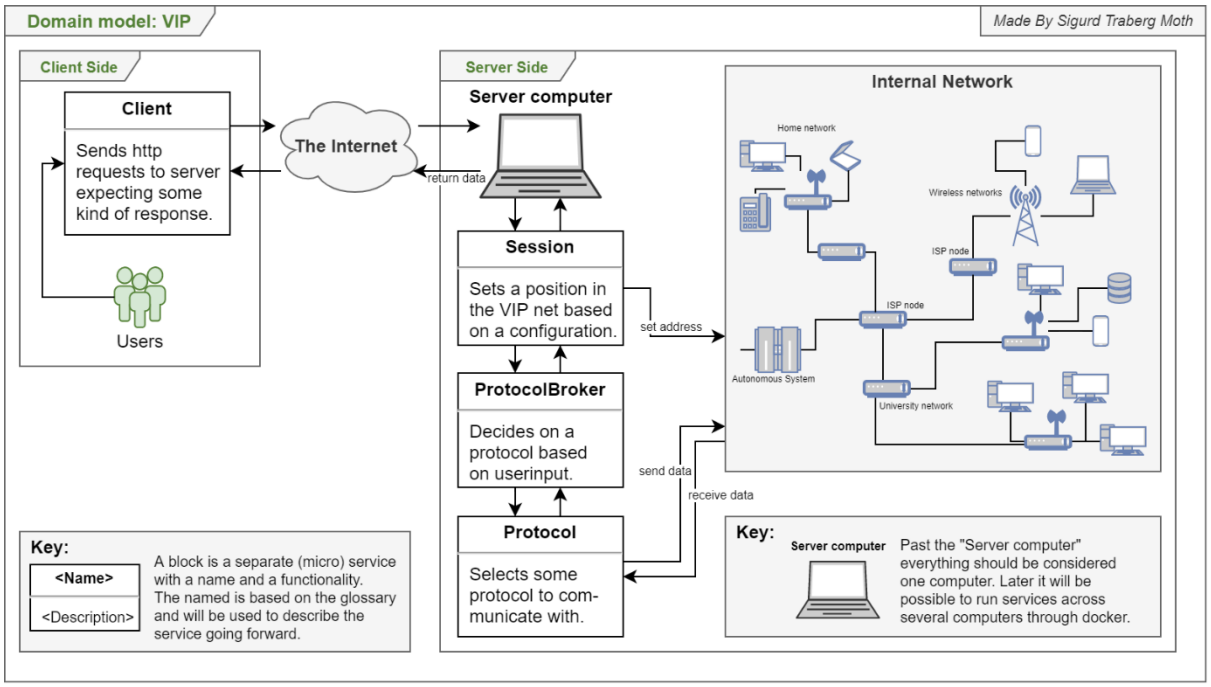


FIGURE 8 - DOMAIN MODEL

Now with the domain model in hand, most parts of the system can be described. For the sake of clarity, it should be understood that the whole system is considered the **VIP**. It consists of two entities, the client-side, and server-side entities. And finally, the Internal Network resides inside the server-side entity but should be considered a system on its own.

B.1.3.1 Client

The client service (Client) is utilized by users (clients) who want to interact with the VIP. Through the client service, users can type CLI commands which are then translated and handled locally or send across the Internet to a receiving side. Ideally, the Client service will be independent of other parts of the system so

multiple users can use their own computers as clients to access the VIP server-side. That will make it possible to establish multiple connections allowing the clients to communicate by sending and receiving data as long as the Client services are connected to the server-side system.

B.1.3.2 Session

After data has been sent from the Client service, it arrives over the Internet through an access point in the session service (Session). The session service will then, based on the data, inform the Internal Network about how the server-side should treat the client and how it should affect the Internal Network service such as adding the client's address to the Internal Network if the client wants to become a host in the Internal Network. The session service could potentially also manage client-related information that still should be located on the server such as user registration and server-related settings. The session finally decides whether the data is considered a protocol and should be handled in the ProtocolBroker.

B.1.3.3 ProtocolBroker

The ProtocolBroker service checks the data sent from the client for what protocol has been requested. Based on the protocol the ProtocolBroker service forwards to the correct Protocol service.

B.1.3.4 Protocol

The Protocol service covers a lot of different microservices, one for each protocol available in the VIP. Each Protocol service will be based on a generic Protocol architecture but besides that functionality (input/output) will be completely different depending on Protocol. The Protocol services should ideally grant the functionality that the identically named protocols on the real Internet have. As an example, the HTTP protocol command should access the HTTP Protocol service and that service should access the Internal Network service with the functionality of HTTP.

B.1.3.5 Internal Network

Finally, the Internal Network service (Internal Network) is where all the network nodes are going to be located and what the preparation through the previous services has been for. It is going to consist of all the nodes that exist on the Internet including everything from hosts, routers, and switches to Autonomous Systems (AS), SVNs, etc. The nodes are all going to be using all the protocols that keep the Internet running and are going to act like on the Internet. Nodes are also going to be susceptible to crashing, congestion, queue overflow, and all the common events that can happen to such devices in real life. The Internal Network service is going to be modular and components are going to be hot-swappable under runtime. As should be clear, the Internal Network service is extremely complex, and this is where there is the most potential for new projects as nodes and protocols need to be implemented and cannot be done in just one project. For this project, a very shallow version is going to exist, which handles a simplified communication between two hosts and nothing more. It serves its purpose for testing and should be considered a dummy implementation to be replaced at some point.

C. ANALYSIS

Having now established what the functional and non-functional requirements are as well as having gotten some idea of how it can all work together, it is time to try and realize how the nonfunctional requirements can be fulfilled. Through analysis, the choice of framework and tools are selected based on requirements, research, and evaluation. It is not necessarily the best tools that are chosen, but rather the best tools based on expectations and experience. In other words, bad choices can have been made because of inferior experience or understanding. The analysis focuses on some of the more interesting choices.

C.1 Spring Boot

Early on it was decided that the Spring-based framework Spring Boot was going to be the framework of choice. Spring Boot brings several useful tools and design strategies to the table which comply with many of the ASRs found earlier. The most used features are explained below.

Spring Boot delivers easier to understand dependency injection

With the use of Spring Boot, it becomes possible to manage dependencies in class files through annotations (@autowired, @service, @component) which results in code that is a lot easier to read, though it shares the exact same functionality. An example can be seen below:

```
public static void main(String[] args) {
        ITranslator translator = new SimpleTranslator();
        IInterpreter interpreter = new SimpleInterpreter();
        IClientWrapConverter clientWrapConverter = new ClientWrapConverterJSON();
        HeadTransmissionHandler transmissionHandler = new TailTransmissionHandler();
        Configurator.getInstance().setStrategy(new TailConfigurationFileHandler());
        Configurator.getInstance().init();
        HeadExpressionHandler expressionHandler = new TailExpressionHandler(translator,
        HeadExpressionHandler expressionHandler = new TailExpressionHandler(interpreter,
                        transmissionHandler);

        IPromptReader reader = new PromptScanner(expressionHandler);
        reader.start();
        IInputReader inputReader = new InputScanner(expressionHandler);
        inputReader.start();
}
```

FIGURE 10 - BEFORE SPRING BOOT DEPENDENCY INJECTION

```
@SpringBootApplication
public class ClientApplication implements ApplicationRunner {

    @Autowired
    private IInputReader inputReader;

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        inputReader.start();
    }
}
```

FIGURE 9 - AFTER SPRING BOOT DEPENDENCY INJECTION

Note how Figure 10 is extremely bloated and carries dependencies through every class file (a necessary evil if the functionality of being able to switch out implementations is wanted). In Figure 9 there is a much clearer separation, but it keeps the same modularity without daisy-chaining implementations of interfaces.

Spring Boot delivers easy to understand XML-less Maven integration through Gradle

Spring Boot allows the use of the Gradle build tool which automatically creates necessary Maven pom.xml files when compiling projects. The result is that it is a lot easier to understand and navigate what dependencies such as repositories and libraries that are pulled externally and where they are pulled from.

Spring Boot delivers a fast and excellent way of creating microservice web applications

Spring Boot has created their own library that handles how services publish and subscribe themselves via HTTP. It follows the rules of RESTful applications and should therefore not be too difficult to understand as the REST API is the most common web API. Additionally, using this implementation makes it easier to migrate the system to containers like Docker and therefore separation of where different services are hosted.

JSON format

For transferring data between web applications, a standard must be picked. Object streams can be used as is common within monolithic applications but that makes it difficult to later separate projects into separate systems. Alternatively, there exist several standards such as JSON, YAML, etc. which are universal, lightweight ways to send data that is *independent* of programming languages. As Spring Boots web application library has been made with JSON in mind, it becomes kind of obvious to use JSON for the VIP. JSON also has the advantage that it is the most used standard, is readable by humans, and that UI frameworks such as React use it.

So essentially Spring Boot is going to make the VIP's services a lot easier to understand and manage, though there will be a learning curve as developers are introduced to the Spring Boot framework. There are also some difficulties with understanding the Gradle build tool as it can become quite advanced. But in general Spring Boot generates a lot of qualities (quality attributes) that the VIP needs, especially concerning *Modifiability*. Spring Boot will also not be forced upon the developer unless they are developing core architecture in already created services. Protocols and new services will be entirely independent of frameworks and Java etc., other than the fact that they consume JSON.

C.2 Layered Architecture

When building larger projects, it is always a good idea to standardize how components in the software are built. One way is to follow a layered model. What this means is that code is separated into packages (or layers) depending on what goal they serve, usually about 3 layers. The layers are commonly known as presentation, domain, and persistence being one of the most used ways of separating incoming data, manipulation of data, and further transfer of data. In Figure 11 the separation can be seen. Another benefit of this architecture is that it makes it possible to exchange whole layers for every single service. This is especially useful if you want additional services in between two services or when you want to replace the user interface (for example going from CLI to UI).

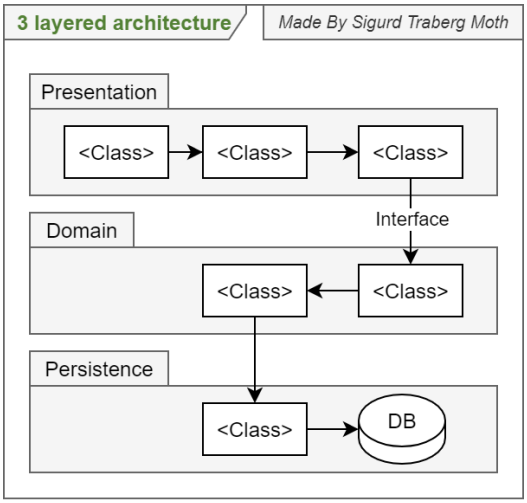


FIGURE 11 - LAYERED ARCHITECTURE

C.3 Docker virtualization

The reason for using Docker is described very well by the Docker community. Docker tries to solve the issue of "it works on my machine" and uses containers to do so. Containers are virtual machines usually running some version of Linux that can host one or more projects. Containers are isolated environments and are clean of bloat software. For this reason, every computer that runs a system using Docker will see the same result. Docker containers can also be accessed externally and by that, it is possible to connect separate Docker containers. As many frameworks and tools have been selected for this project and modifiability is very important, moving to Docker will allow developers to load each container with the specific software that the developers want to use without having to set up .gradle files or other device-specific actions.

C.4 GitLab Pipeline

Pipelines are a part of DevOps CI/CD, more specifically continuous integration (CI). Pipelines run when code is uploaded to a repository with a yml-file and does two things. Every pipeline must contain jobs and then stages for each job. That is a stage can be to build or test code, while jobs are parts of stages. Jobs are then run on runners which are virtual machines. A pipeline in programming is a general term and is not specific to GitLab. That being said, pipelines are different for each platform such as GitHub, GitLab, Azure, etc., and use each their syntax and libraries. Pipelines work well together with Docker and there exists a lot of documentation to help make the transition to using pipelines. Using GitLab's pipeline it will be possible to automate a lot of time-consuming processes such as setting up authentication files that normally require SSH key generation and correct folder setups. It will also solve the issue of having to download a lot of repositories independently and manually, and finally, it will automate self-made libraries, so libraries do not have to be manually published and versioned.

D. DESIGN

In the design phase, the focus is not on what the software should do, but rather how the software should do it. The goal is to decide on the best design patterns (microarchitecture) within the framework (macro architecture) that was decided upon. The analysis has so far already sparked a lot of good ideas in addition to looking back at requirements, and these ideas are now built upon.

D.1 Physical View: VIP

The domain diagram is further refined taking into considerations where components are going to be located, and how state is going to be stored, e.g., where configuration files are going to be necessary. The output is a physical view of the VIP (Figure 12).

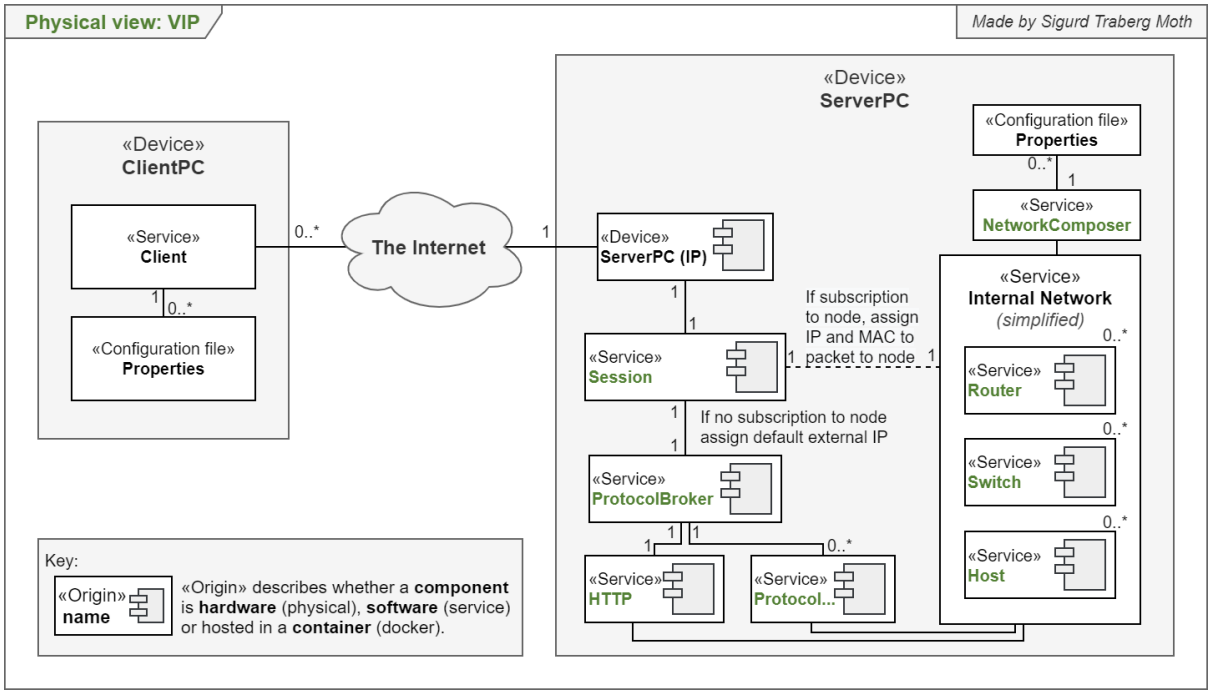


FIGURE 12 - PHYSICAL VIEW OF THE VIP

D.1.1 Configuration Files

There are generally going to be two areas where configuration files are necessary to avoid repetitive tasks. First, the user wants to set a lot of client configurations that have nothing to do with the server. That will generally be things such as their username, wanted IP, being internally or externally placed in the Internal Network, the information they want to log, and other settings. Second, the user wants some settings connected to the session such as the client's actual address and if they want data monitoring. These settings might require access to the Internal Network and needs to be reported back to the client.

Additionally, as stated in requirements, it is going to be useful for teachers, students, and project groups to have some way of saving, resetting, and changing default configurations of the Internal Network.

D.1.2 Docker

There are also considerations to be made, whether it makes sense to run every service and Internal Network component in separate Docker containers, if only services should be in containers, or if only client and server-side should run in containers. This is going to depend on performance targets versus scalability and device independence. For now, as Docker has not yet been successfully implemented, services will be designed with the Spring Boot REST implementation so that they are ready to be implemented with Docker.

D.2 Overview: VIP

Having decided upon most of the core architecture, it is now possible to create an overview of the VIP (Figure 13). The overview shows how each service is separated into layers with the layered architecture in mind creating a simple, recurring design. With this diagram, it is now much easier to detail each service individually implementing the Spring Boot framework and layers.

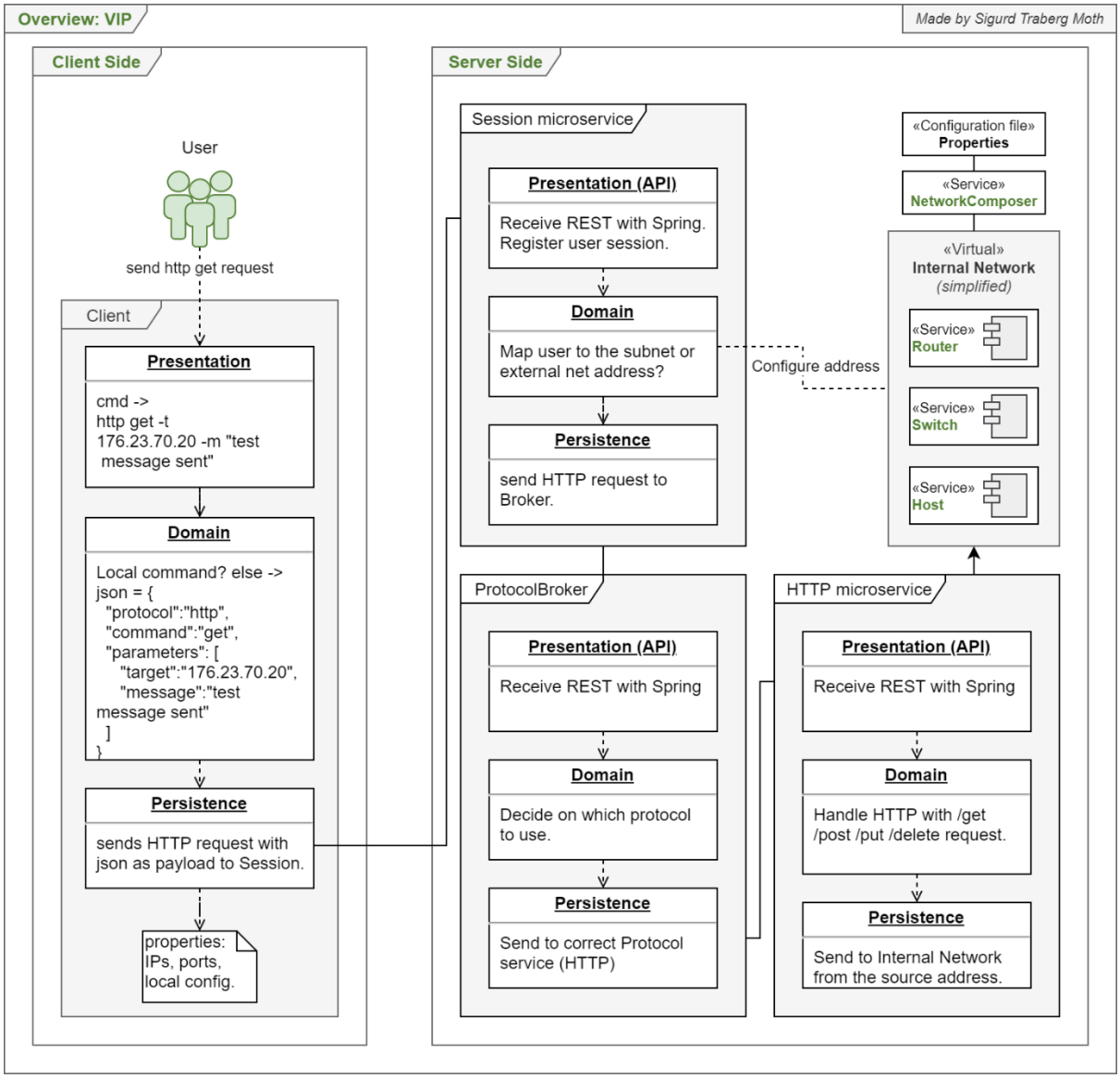


FIGURE 13 - OVERVIEW DIAGRAM OF THE VIP

D.3 Libraries

Most if not all services are going to use self-made libraries. Libraries are online available Java .jar files on GitLab and are made so that shared functionality through codebases does not have to be manually copied to every service's codebase. The current libraries consist of the following:

- (Important) A library handling the user input in a Java object format that can be used in every service referred to as an **Expression** consisting of classes in a package.
- (Important) A library that creates a **Wrap** containing both the Expression and additional metadata. The wrap is used for wrapping and unwrapping all user input and metadata in an object format

between transmission to other services so data can be used by the VIP. Wrap also contains the interface bundlable which services can implement to create new bundles which can then be added to the metadata.

- A library **Transmit** that transmits the Wrap to other services. It is a simple library, but it makes it easier to change the implementation of transforming to and from JSON.
- A library **Network** that contains data models for nodes in the network which every service can then reference and build.

The libraries are made as generic as possible so they can be reused in most cases. Each library will have its own project in a subgroup on GitLab which can be changed if necessary. Since libraries use versioning, every service having a dependency on a library must have its dependency version updated before using newer versions. It is therefore possible to update the VIP service by service instead of all at once which would result in compile errors across every service.

D.4 Handling User Input (Expression and Wrap)

The **Wrap** carrying an **Expression** mentioned in D.3 Libraries is a very essential part of the VIP as it contains all user input. It is therefore elaborated further here.

A user writing a string of text and the way the VIP handles the input as an Expression resembles a standard terminal very well. The input is separated into three elements, and they are as follows:

- A type of primary functionality (referenced as a **protocol** in code) that the user wants to use, the ongoing example here is the HTTP protocol.
- The user will want to use some form of primary **command** within the protocol, E.g., get, post, etc.
- Every Expression can have optional **parameters** further specifying how the input is handled by services.

A diagram of the Wrap and Expression can be seen in Figure 14.
An example of an HTTP post can be seen in Figure 25.

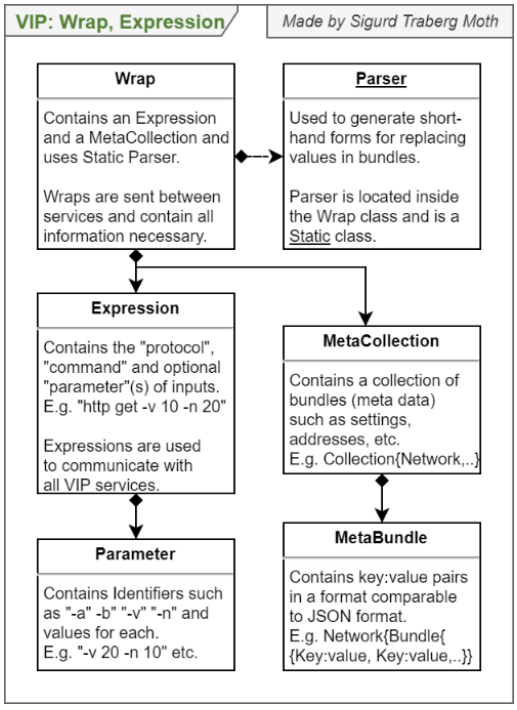


FIGURE 14 - DESIGN OF WRAP AND EXPRESSION

Besides the text string, additional information might be required. In HTTP for example it is necessary to have an address for the receiving node. This address can be set in local configurations by another "protocol" and command with parameters. The additional information named metadata is located in a MetaCollection separated into bundles such as a network or user bundle depending on content. The Expression and MetaCollection are both saved in the Wrap making for a single object that can be moved around which is easily translatable to JSON.

D.5 Internal Network Design

The Internal Network also needs to be planned out. There are a few choices to be made as to how nodes inside the Internal Network are hosted and handled. Given that only the application layer will be explored in this project, it is hard to create a proper architecture here as the protocol stack requires lower layers for it to transfer data properly. Therefore, an alternate implementation for lower layers will have to do until nodes and the stack has been designed. To avoid this issue and to have something to show, the nodes are created directly onto the Session service so that the Session will have direct control over all nodes in the Internal Network (Figure 15). Doing so will make Session which contains address metadata have access to both source and destination addresses and the trouble of navigating through nodes can hence be skipped.

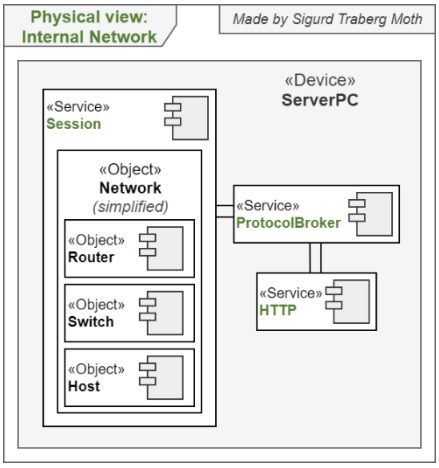


FIGURE 15 - PHYSICAL VIEW: INTERNAL NETWORK

D.5.1 Internal Network Nodes

Having decided on how nodes are hosted, nodes must now be modeled. Since the host nodes will handle barebones communication (for a start) through HTTP (Get, Post), it is necessary to have at least 2 hosts in the system to be able to test the HTTP protocol. The hosts are close to identical except for the fact that they do not share identical networks, addresses, and some dummy values. The design of each node can be seen in Figure 16.

As the Internal Network is placed on the Session service and layers beyond the application layer are omitted, Nodes too will need a different design. The decision, for now, becomes to design nodes with values that might be expected from different layers such as IP, ports, MAC addresses, etc., this makes it easier to showcase that Nodes are different in the VIP. Additionally, the Nodes will need an ID for the client to find, as IP's can be different (or the same!) depending on if they are being masked behind a subnet, and as the network layer is not implemented, it is better to avoid entirely to interact with the IP as to not obfuscate its purpose. Finally, nodes also need some form of protocol container for Nodes to be able to interact with different protocols. E.g., a host needs an access point for HTTP get and post if they need to be able to receive those HTTP methods.

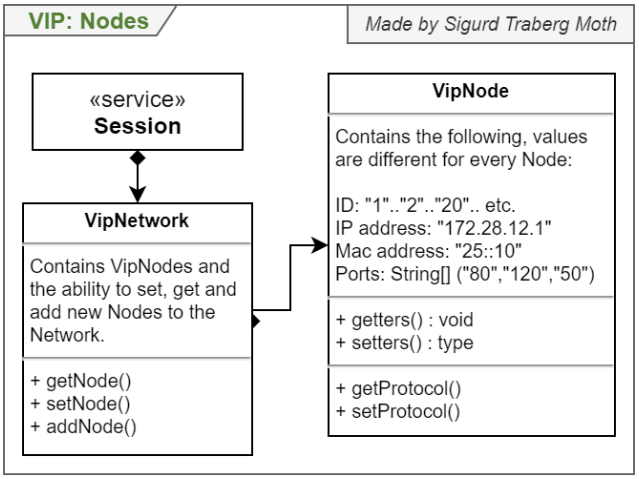


FIGURE 16 - VIP NODES DESIGN

Quick note: The design of the Internal Network service itself deserves a much more in-depth analysis and should follow RFCs[3].

[3] https://www.ietf.org/standards/rfcs/

D.6 HTTP Design

A basic design of the HTTP protocol has been created to imitate its functionality. It is straight forward and works as one would expect with HTTP get and post. A user can ask (get) for content located at a given URL and will receive the content given that there is content saved at the location. Alternatively, a 404 or 400 error is returned. HTTP post works as well and can save content to a URL responding with 200 OK if successful. The correct HTTP command is chosen based on the input saved in the Expression and is used in the Unwrapper (Figure 17).

The design is simple and easily extendible, allowing for the implementation of all error types as well as HTTP put and delete. The only reason they are not currently added is that they do not provide much more in terms of showcasing the functionality of the system.

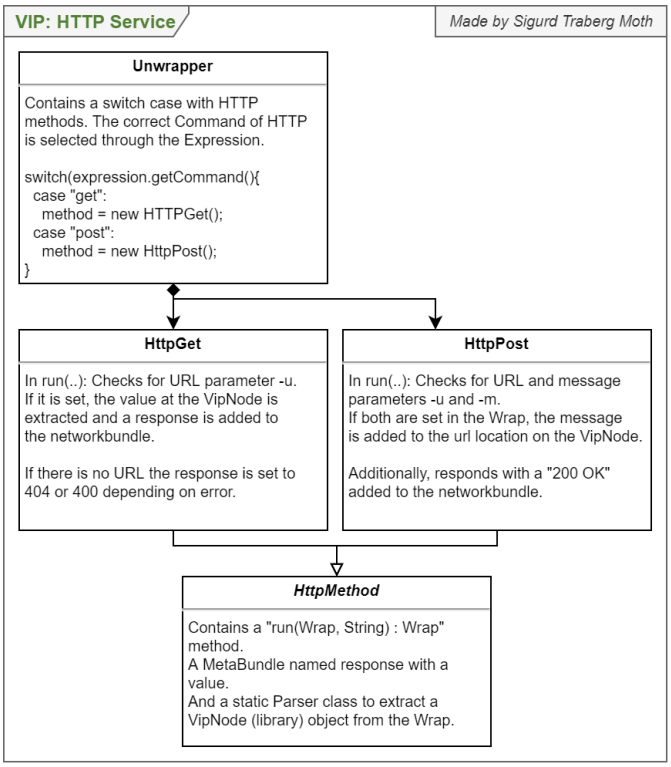


FIGURE 17 - DIAGRAM OF THE HTTP SERVICE

D.7 Monitoring the Internal Network

For monitoring the Internal Network several types of monitoring services have to be considered and preferably, most of them should share the same design. In the requirements, it was deemed critical that the system can determine if nodes become unresponsive (among other things). It also made sense that nodes should be pingable. These two designs require different approaches as they do not share the same direction of dataflow. A solution that can handle both has been designed with these two cases in mind (Figure 18).

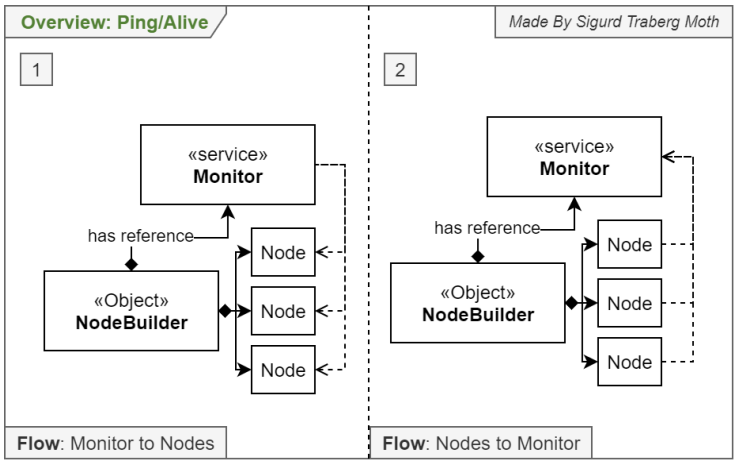


FIGURE 18 - OVERVIEW PING/ALIVE AND OTHER MONITORING SERVICES

This design will allow for subscribed clients in the Session service to ask for pings directed from the monitor to the Nodes to check if a node is alive. It will also allow for Nodes to repeatedly ping the Monitor service informing if errors are happening which clients can then also subscribe to. The design can be borrowed by sibling services which can then be used to ask for timestamps, data, etc.

E. IMPLEMENTATION

In this chapter, the focus is on the actual implementation and use of services. The chapter contains specific code snippets of interest, user input, and user output. JSON snippets will also be located in attachments with a more easily readable format.

E.1 Expression and Wrap

The code for the Wrap and Expression is shown here as its functionality is vital for every service in the VIP (Figure 19).

A user can through the client write an input that is transformed into an **Expression** which is wrapped with metadata in a **Wrap** and is then processed by the VIP.

Every Expression consists of a <protocol> <command <parameter..> <value..> design with an optional number of parameters and belonging values. The Expression itself is never manipulated.

The MetaCollection contains the metadata which on the other hand can be manipulated and where new bundles can be added with additional data of interest along the way across services.

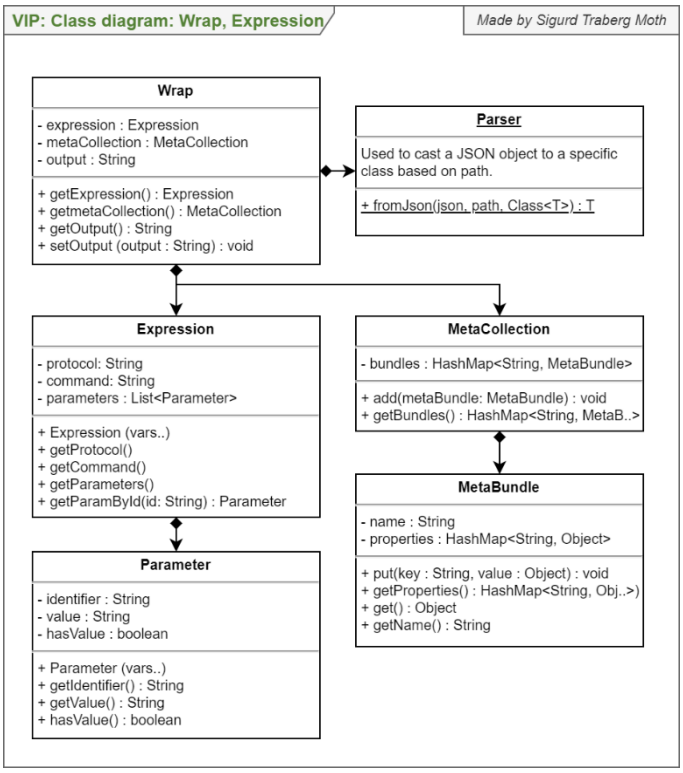


FIGURE 19 - CLASSDIAGRAM OF WRAP AND EXPRESSION

E.2 Configuration Models and MetaBundles

To make it easy to store data and separate it in the future, a model for configuration files has been made (Figure 20). There is one Configuration for every file and the configuration files are stored locally as .json files.

The Configurator works by initializing all Configurations at the startup of the client. If files do not already exist, they are created with empty values and must be set.

Every Configuration is inhabited by a MetaBundle that contains key-value pairs and these bundles can then be added to the Wrap by the Client service.

To add a new Configuration to the code, all that is needed is to create a file like the NetworkConfiguration which for the most part is just getters, setters, and an implementation of the MetaBundle interface. It must then be accessed through the Configurator in the main code. Other services can also add MetaBundles by implementing the interface Bundlable from the library Wrap.

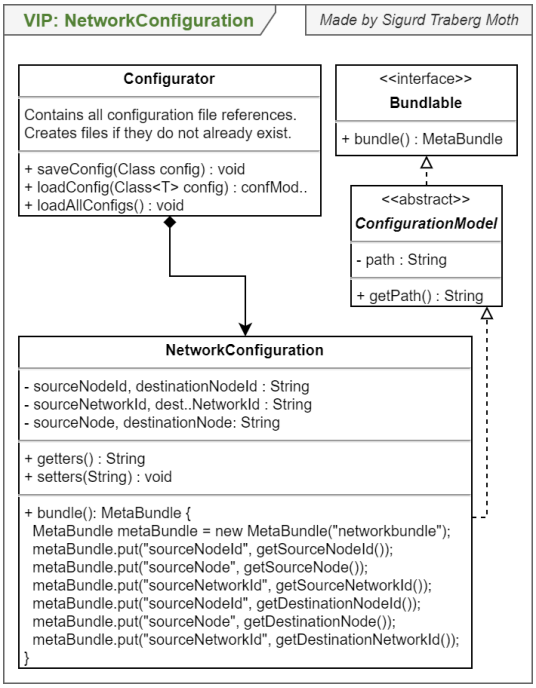


FIGURE 20 - CLASSDIAGRAM NETWORKCONFIGURATION

E.3 Local Configuration Example

Adjusting configurations work exactly like other Expressions. A user can currently change their source and destination address as well as their username. The features are barebones but prove the architecture which allows for easy extension and for adding more features with little to no trouble.

As an example, a user can set his or her username by typing the following (Figure 21):

```
<<====<==<<=<========<==<=
> :bootRun
set user -n newUsername
```

FIGURE 21 - SETTING AND SAVING A NEW USERNAME

The new username is then overwritten in a local configuration file saved to the PC (Figure 22):

```
Protocol:   set, Command:  user
         -n newUsername
UserConfiguration:
        + added parameter n of value:newUsername
```

FIGURE 22 - A NEW USERNAME IS SAVED

The code below represents some snippets of how a "command" is added to a map and identified with name, class, and parameter (Figure 24 and Figure 23).

```
Parameter parameter = expression.getParameterByIdentifier("n");
if (parameter != null){
    userConf.setName(parameter.getValue());
```

FIGURE 24 - A SNIPPET OF SETUSER.JAVA (CLIENT)

```
if (expression.getProtocol().equals("set")) {
    Map<String, IExecuteExpression> setCommands = new HashMap<>();
    setCommands.put("source", setSource);
    setCommands.put("destination", setDestination);
    setCommands.put("user", setUser);
    ProtocolHandler setProtocolHandler = new ProtocolHandler("set", setCommands);
```

FIGURE 23 - A SNIPPET OF EXPRESSIONHANDLERIMP (CLIENT)

E.4 HTTP Post Example

A user can through the CLI in the client write an HTTP post request. Requirements for this protocol command is that the protocol, the command, a URL, and a message is specified and converted to an **Expression** (Figure 25).

```
<=<====<==<==<=<==========----> <=<=<<=<==========----> 75% EXECUTI<=
> :bootRun
http post -u http://www.url.com -m this is a message to be saved.
```

FIGURE 25 - USER INPUT OF AN HTTP POST REQUEST

E.4.1 Client sends an HTTP post Wrap

After writing the post request the client attaches additional metadata to the Wrap from local configuration files. Among these are already set the source and destination addresses saved in a networkbundle, and user information set in a userbundle (Figure 26). The Wrap is then sent to the Session service.


```
{"expression":{"protocol":"http","command":"post","parameters":[{"identifier":"u","value":"http://www.url.com","hasValue":true}
,{"identifier":"m","value":"this is a message to be saved.","hasValue":true}]},"metaCollection":{"bundles":{"networkbundle":{"n
ame":"networkbundle","properties":{"sourceNode":"1","destinationNodeId":"1","sourceNodeId":"0","sourceNetworkId":"0","destinati
onNetworkId":"0","destinationNode":"0"}},"userbundle":{"name":"userbundle","properties":{"name":"SomeUser"}}}}}
```


FIGURE 26 - JSON EXPORT FROM HTTP POST

See Figure 33 - Export in JSON in attachments for a proper JSON format.

E.4.2 Session attaches data and forwards

Arriving at the Session service, source and destination nodes are found by ID and necessary metadata is added to the Wrap. The Wrap is then forwarded to the ProtocolBroker which selects the correct protocol service by looking at its metadata. The Wrap is then forwarded to the HTTP service.

E.4.3 HTTP protocol service selects a method

The HTTP service contains the get and post methods of HTTP and again looking into the Wrap's metadata the Post method is selected. As the Internal Network is currently located on the Session service, the Wrap is returned to the Session.

E.4.4 Session performs an action

Back at the Session service, the destination node is updated with the new information which is saved to the URL with the corresponding message "this is a message to be saved.".

E.4.5 Client receives confirmation

The client finally receives the Wrap with all the newly added data including a confirmation such as "200 OK" if the HTTP post was successful (Figure 27).


```
{"expression":{"protocol":"http","command":"post","parameters":[{"identifier":"u","value":"http://www.url.com","hasValue":true}
,{"identifier":"m","value":"this is a message to be saved.","hasValue":true}]},"metaCollection":{"bundles":{"networkbundle":{"n
ame":"networkbundle","properties":{"sourceNode":{"id":0.0,"ip":"198.168.1.2","mac":"25::10","ports":[80.0,8080.0,443.0],"protoc
ols":{"init":{"name":"init","properties":{}}}},"destinationNodeId":"1","sourceNodeId":"0","sourceNetworkId":"0","destinationNet
workId":"0","destinationNode":{"id":1,"ip":"198.168.1.3","mac":"26::11","ports":[200,614,20],"protocols":{"init":{"name":"init"
,"properties":{}},"http":{"name":"http","properties":{"http://www.url.com":"this is a message to be saved."}}}}},"userbundle":
{"name":"userbundle","properties":{"name":"SomeUser"}},"response":{"name":"response","properties":{"value":"200 OK"}}}}}
```


FIGURE 27 - JSON IMPORT FROM HTTP POST

See Figure 34 - Import in JSON in attachments for a proper JSON format.

E.5 HTTP Get Example

After performing an HTTP post method, the newly stored information can then be found using the HTTP get method. This is done the same way by writing an HTTP get request in the CLI (Figure 28).

FIGURE 28 - USER INPUT OF AN HTTP GET REQUEST

E.5.1 Client sends an HTTP get Wrap

Just like the HTTP post, the input in the CLI is converted to an Expression. Local configurations are added to the MetaCollection and wrapped in a Wrap. The Wrap is sent through the Session service where additional metadata is added including addresses of nodes and it is then sent to the correct protocol service through the ProtocolBroker (Figure 29).


```
{"expression":{"protocol":"http","command":"get","parameters":[{"identifier":"u","value":"http://www.url.com","hasValue":true}]
},"metaCollection":{"bundles":{"networkbundle":{"name":"networkbundle","properties":{"sourceNode":"1","destinationNodeId":"1","
sourceNodeId":"0","sourceNetworkId":"0","destinationNetworkId":"0","destinationNode":"0"}},"userbundle":{"name":"userbundle","p
roperties":{"name":"SomeUser"}}}}}
```


FIGURE 29 - JSON EXPORT FROM HTTP GET

E.5.2 HTTP protocol service selects a method

In the HTTP protocol service, the Wrap is unwrapped, and the correct HTTP method (get) is selected through a switch case and saved to the Wrap's metadata.

E.5.3 Session performs an action

Back at the Session service, the HTTP get method is performed on the destination node and if the URL is correct, the saved content of the URL is extracted from the node. If there is no data at the URL, the get method responds with “404 File Not Found” if there is no saved data at the URL.

E.5.4 Client receives information

Finally, the client receives the full Wrap with Expression and metadata. This includes the message “this is a message to be saved.” which was saved in the HTTP post example. The response is seen below Figure 30.


```
{"expression":{"protocol":"http","command":"get","parameters":[{"identifier":"u","value":"http://www.url.com","hasValue":true}]
},"metaCollection":{"bundles":{"networkbundle":{"name":"networkbundle","properties":{"sourceNode":{"id":0.0,"ip":"198.168.1.2",
"mac":"25::10","ports":[80.0,8080.0,443.0],"protocols":{"init":{"name":"init","properties":{}}}},"destinationNodeId":"1","sourc
eNodeId":"0","sourceNetworkId":"0","destinationNetworkId":"0","destinationNode":{"id":1.0,"ip":"198.168.1.3","mac":"26::11","po
rts":[200.0,614.0,20.0],"protocols":{"init":{"name":"init","properties":{}},"http":{"name":"http","properties":{"http://www.url
.com":"this is a message to be saved."}}}}},"userbundle":{"name":"userbundle","properties":{"name":"SomeUser"}},"response":{"n
ame":"response","properties":{"value":"this is a message to be saved."}}}}}
```


FIGURE 30 - JSON IMPORT FROM HTTP GET

E.6 Internal Network Nodes

The current implementation of Internal Network Nodes is hosted on the Session Service as described in D.5 Internal Network Design. The implementation consists of Lists of VipNetwork objects which can contain, add, delete, and reference VipNode objects. A VipNetwork is synonymous with a subnet while a VipNode can be considered a host, at least for now. A simple implementation is seen below (Figure 31).

```
VipNetwork vipNetwork1 = new VipNetwork(0);
vipNetwork1.addNode(new VipNode(0, "198.168.1.2", "25::10", Arrays.asList(80, 8080, 443)));
vipNetwork1.addNode(new VipNode(1, "198.168.1.3", "26::11", Arrays.asList(200, 614, 20)));
VipNetwork vipNetwork2 = new VipNetwork(1);
vipNetwork2.addNode(new VipNode(0, "198.168.1.4", "15::2", Arrays.asList(100, 314, 10)));
vipNetwork2.addNode(new VipNode(1, "198.168.1.5", "16::4", Arrays.asList(200, 614, 20)));
```

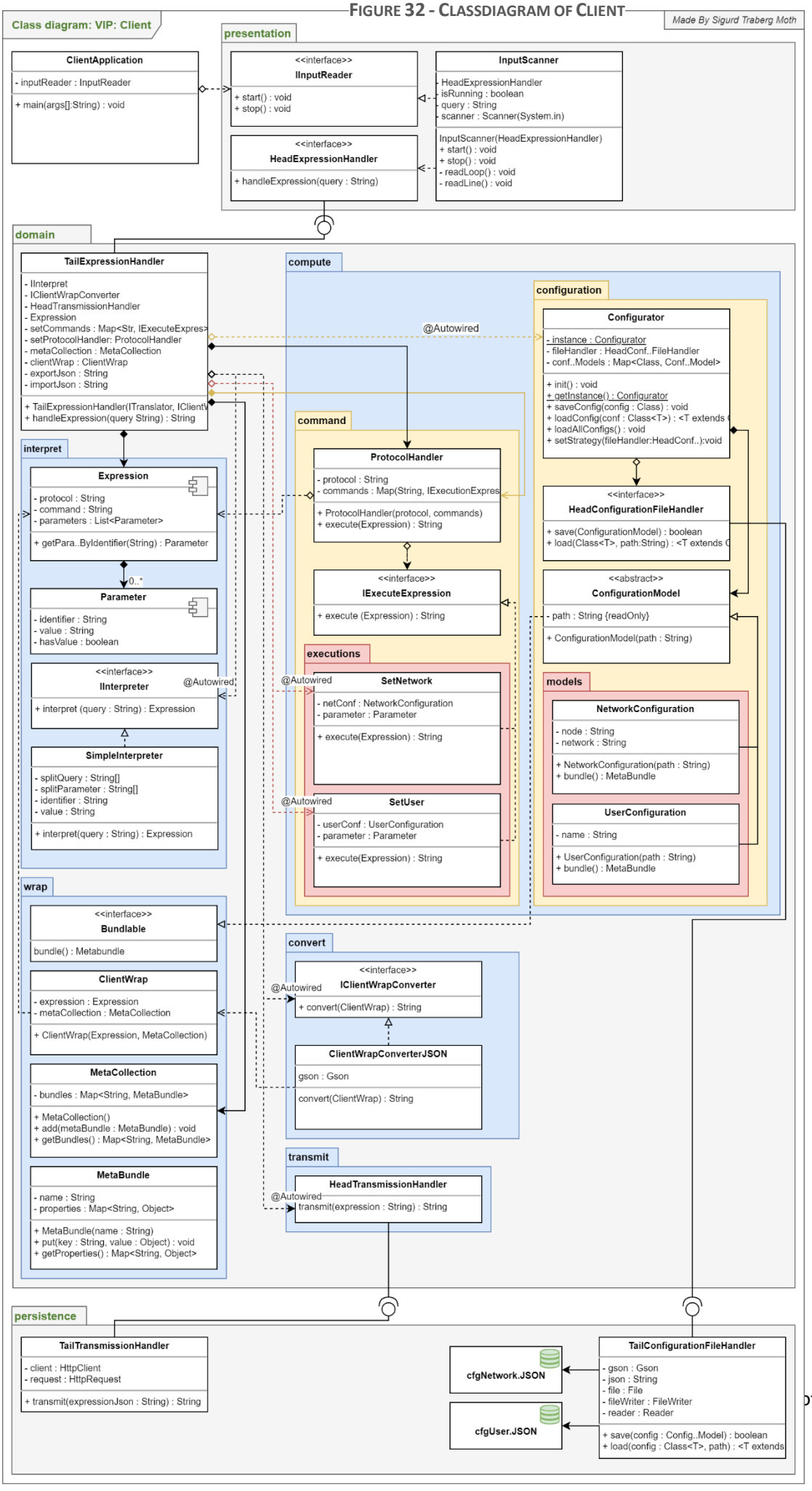
FIGURE 31 - CODE FROM SESSION SERVICE OF VIPNETWORKS AND VIPNODES

Currently, a VipNode contains an Id, IP, MAC address, and ports. There are two unusual things here, one is that every Node also has an Id which is currently used to find Nodes from the Client service. The other is that when a VipNode is to be interacted with, it is saved into the MetaCollection of the Wrap which is then processed in a protocol service. If the protocol used is an HTTP get method, then the protocol is also added to the MetaCollection which then replaces the VipNode when it arrives at the Session service again. This currently can cause some inconsistencies if two users hypothetically changed the same Node at the same time, and it is something that should be checked for in the future. Possibly through a monitoring service.

E.7 Client Class Diagram

Finally, this chapter ends with a class diagram of the whole Client service. The diagram is left on the next page as it is difficult to display due to its size. But it serves as a good example of how the transition from the design to the implementation was realized. It also makes it easier to visualize how every layer and package interact with each other. The color of each package represents package depth, which means that gray is level 1, blue is level 2, yellow is level 3, and red is level 4 (Figure 32).

FIGURE 32 - CLASSDIAGRAM OF CLIENT



Wrapping it all up, the final product of this project is delivered. The solution is not a finished product, but it has come quite a long way. There is a clear design strategy for the platform with not too many bigger issues to be solved. But make no mistake that there is a lot of implementation left to be done for the VIP to be a useable network platform for teaching students and testing software.

F. DISCUSSION

In this section, a few discussions have been handpicked to try and explain why some choices or mistakes were made or were not made, which did not fit in elsewhere in the report.

F.1 Monolithic Architecture

Going into this project there was not a lot of consideration for making a monolithic architecture which is essentially a single program, unlike microservice architectures. The reasoning was that monolithic architectures become very coupled and incohesive over time as there are no boundaries to prevent bad designs by programmers. As various types of modularity from the get-go was a top priority, the idea of a monolithic architecture was quickly abandoned. That said a monolithic architecture could have been faster to build, and it would probably have been better performing if well built.

F.2 REST Performance

The VIP had several nonfunctional requirements related to the performance and speed of the platform. They were not considered too heavily as it was agreed upon that performance would be something to look into if necessary and therefore not a priority. In hindsight, it has become clear that the Spring Boot web implementation using HTTP does sacrifice a lot of performance between services, possibly because of the JSON conversions. A potential change going forward could be to decide on which server-side services could be grouped together locking them to a single hardware platform (e.g., one computer), and then reduce the number of HTTP transmissions and JSON conversions required.

F.3 Tool Complexity

One of the fears from back when requirements were being established, was that maybe too many new tools were being introduced to the project. Having no experience with neither Springboot, Gradle, self-made libraries, JSON applications, and Docker to some extent, pipelines and microservice architecture could prove to be quite a challenge. And so, it was. One could argue that there is a fine line between improving the project by the use of tools and using tools at the expense of the project, which this at some points felt like. It is important to understand that many of the decisions above were dependent on one another, but looking back, excluding the pipeline would have been a good idea as it would only contribute to easier management of Git and deployment with Docker. Then more time could have been put into the implementation of either the Internal Network or testing which would have resulted in a stronger product with just a bit more effort. Trying to implement Docker also ended up being a waste of time as it was not manageable before the deadline. Usually implementing Docker is not too difficult, but it proved to be hard as the Docker containers would require Springboot, and due to limited experience with Docker, creating a container that pulls dependencies on Springboot through Gradle was a problem. Docker would have made the VIP a lot nicer as it would have been easier to make it platform-independent and host independent, but as the pipeline also failed, it would not be as big an impact. Fixing both the pipeline and dockerizing the VIP should be done going forward.

F.4 Mono-repositories and Multi-repositories

An issue that came up a couple of times was the advantages and disadvantages of having one repository containing every microservice vs a repository for each microservice. There has always been a kind of

religious fight going on in the online community of what is the best way to set up repositories. At first, the project was set up with just a single repository. It worked just as one would expect, and with the then-existing pipeline, Git and branches were used to handle the separation of code. This strategy made it quick to download the repository and set it up on pcs as only one repository had to be downloaded. On the negative side, if only one microservice was needed, every other microservice would still be downloaded. Additionally, Git commits could contain changes for other services that they should not.

At a later point, the project was switched to a multi-repository, lured by suggestions that when working with microservices, having one repository per microservice would improve workflow. In general, the experience was that it made it a lot slower to set up projects as every service had to be downloaded independently. A thing it did achieve, was to truly separate the work in each microservice. That is, one could not work on a branch and change the ProtocolBroker service while simultaneously changing the client application. This was less of an issue when only one person is working on the project, but it is possible to imagine if several people or even groups are working on the system, conflicts could arise more easily in a Monorepository. Finally, there are tools for Multirepositories that make it simple to download every repository at the same time, so this is more of a temporary annoyance.

F.5 Failed Pipeline

A choice that was made early on was to try and make room for setting up a pipeline using GitHub as it is considered the default Git repository service. This choice was made before the realization that GitLab was a requirement. The project was therefore set up with a pipeline based on a combination of GitHub actions and Azure DevOps. It worked, but when the pipeline inevitably had to be moved, it required a total rewriting of the pipeline code. Moving the pipeline was unsuccessful as GitLab would not build the projects using its default before pushing the software to GitLab, and so it was necessary to write a Runner, this proved to be too time-consuming, and resulted in having to remove the pipeline which was a huge pain and a blow to the project. This once again proved that properly finding and stating requirements ahead of implementation is important, as assumptions cannot be made, and reworking architecture and reverting implementation is a lot more expensive than writing proper code after designing it. Recreating a pipeline should not prove too difficult and should be an objective for when the VIP is moved to Docker.

G. CONCLUSION

In conclusion, a system has been designed that has the ability to transfer data across nodes in a network. The system is simple, extremely modular, easy to extend, and also not far away from being both platform-independent and running on a pipeline for increased usability for both users and developers. The system's simplicity and modularity are powerful properties, and it is clear that Spring Boot has contributed to this. Spring Boot's REST implementation has made it possible in a very developer-friendly way to separate codebases into microservices with HTTP. Spring Boot has also made the strategy pattern much less code-heavy and confusing, which is normally a necessary evil when designing layered services where layers can be exchanged. Spring Boot naturally benefitting from JSON has also allowed for a code language-independent, human-readable, form of communication between services for even more freedom to developers.

It is not all sunshine and roses as simplicity in other areas had to be sacrificed in the use of tools such as Spring Boot and pipelines. Both in the case of time spent setting it up, but also because a lot of setup and understanding is required to install and continue working on the platform. The great thing is that when first these relatively basic things are understood, the whole system will be much easier to understand as

most designs are very healthy patterns and are repeated across all of the VIP. Due to the amount of modularity introduced, latency in the system has increased. The system, therefore, suffers a bit performance-wise, and there are cases of data becoming outdated in transit, but several ways of optimizing these issues have been figured out. Both through reducing the number of HTTP requests between services, but also by implementing the proper protocol stack as it is going to handle inconsistencies. Applications themselves also usually have a hand in this as they have to correct for outdated information.

Finally, an incomplete design has been created for the monitoring services to come. The way that user input is handled has made it easy to configure and limit what kinds of data that is wanted by the user in every service. Because of this, it will be harder to define the specific requirements for what data is wanted rather than how to actually code it and retrieve the data.

With this system, an idea of how the final product should look like has become much more refined and it should not be too difficult to see how this product could potentially be shaped into a fully-fledged working virtual network platform. There is still a long way, but hopefully, the VIP serves as a good platform with a solid architecture for exploring such a platform further, and hopefully, the documentation will provide the necessary information for quickly establishing goals and getting new developers up to speed fast and efficient.

H. PERSPECTIVE

The VIP attempts to create an educated guess of what and how a network education-focused platform might be done. Looking forward, time will tell whether the architecture actually makes sense, holds up, and is fast enough, or if it has critical oversights.

H.1 Features to be Developed

It is going to be exciting to see if and how the Internal Network can be implemented as a standalone service and how it will evolve with different kinds of Nodes having access to different protocols. It is also going to be exciting to see if the architecture can handle the implementation of lower layers of the protocol stack and serve multiple users without it being dragged down by latency due to processing power shortage.

The platform is also not a long way off from running on Docker, and that could solve a lot of setup issues that currently take up time for any new developer involved.

For the UI it should be fairly simple to add as every button and input would basically just be the same strings of code as written in the CLI just more automated. With UI a lot of “continuous updates” are going to be more important, as a window can hold a lot more information visually than with a CLI. It would be great to see a kind of monitoring service that can be toggled on and off, and a visual representation of all Nodes, their uptime, load, etc., for a more user-friendly and complete experience.

Finally, the platform has a lot of bad naming conventions (the expression’s use of the word protocol comes to mind), it would be great to clean it up with a stronger more identifiable naming scheme that cannot be so easily confused with real terms. The VIP would also benefit from implementing some testing and error handling, as it could quickly become confusing if new functionality does not work, and so, where it fails. Fixing both naming conventions and adding some testing would increase the platform's reliability by a lot though it, in its current form, is quite stable.

H.2 The Architecture's Impact on Future Development

For new potential developers, there is going to be a learning curve as there is with any software in development. Given the architecture, it should be possible to get into the VIP's design structure, and lots of manuals, diagrams, and the SRS, has been provided for this purpose. As most design patterns have already been explored for input, services, protocols, Nodes, Configurations, and even to some extent the monitoring, it should be easy to continue and add the much-needed functionality that the VIP needs.

In the end, there is only one thing to say. If development is continued, then I wish good luck to those that get to continue working on the VIP, and I hope that they will enjoy the design of the architecture.

I. BIBLIOGRAPHY

Anders Dahl, T. D. (2016). *Styrk Projektarbejdet, 3. udgave.*

Frank Tsui, O. K. (2018). *Essentials of Software Engineering, Fourth Edition.* Jones & Bartlett Learning.

Jim Arlow, I. N. (2009). *UML 2 and the Unified Process, Second Edition.* Pearson Education Inc.

Kurose, J. F., & Ross, K. W. (2017). *Computer Networking A Top-Down Approach.* Pearson.

Len Bass, P. C. (2013). *Software Architecture in Practice.* Pearson Education Inc.

Nikolaidis Pavlos, J. C. (2011). *Software Requirements Specification for JHotDraw.*

J. ATTACHMENTS

J.1 HTTP Post Wrap

User input that is transformed into a Java Wrap and Expression object and then sends through the VIP returning with added information using HTTP post.

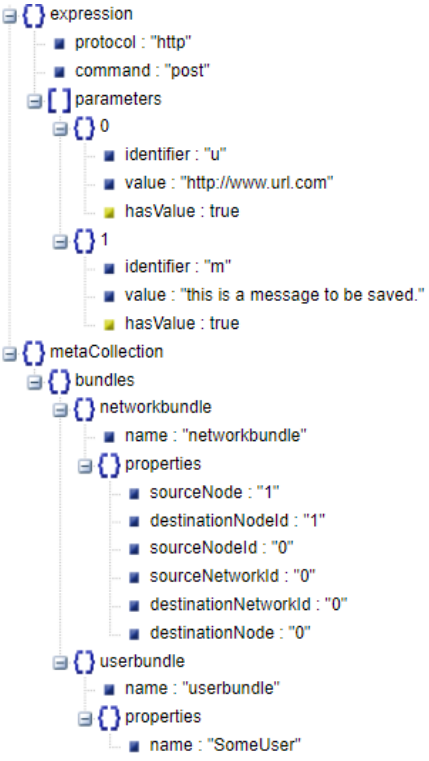


FIGURE 33 - EXPORT IN JSON

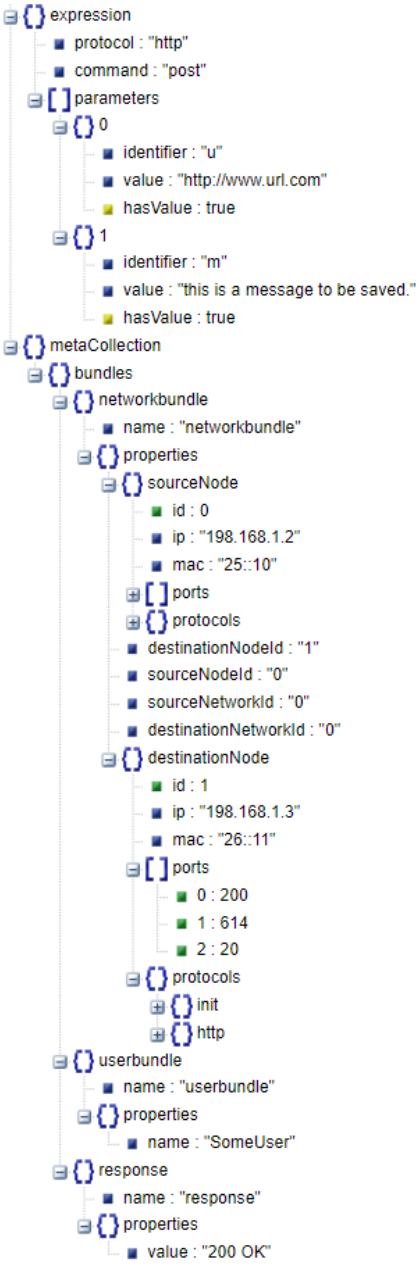


FIGURE 34 - IMPORT IN JSON

J.2 HTTP Get Wrap

User input that is transformed into a Java Wrap and Expression object and then send through the VIP returning with added information using HTTP get.

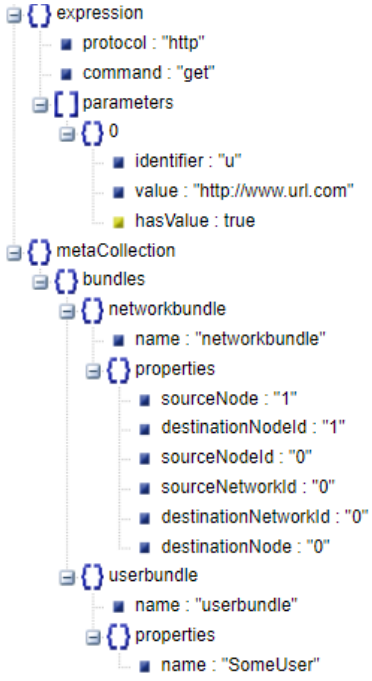


FIGURE 35 - EXPORT IN JSON

FIGURE 36 - IMPORT IN JSON