Global Monitoring and Visualization of the Internet

Bachelor End Project - Final Report

R. Huisman
A. Jeleniewski
S. Kaptein
T. Saveur
S. Weegink

Delft University of Technology

Global Monitoring and Visualization of the Internet

Bachelor End Project - Final Report

by

R. Huisman
A. Jeleniewski
S. Kaptein
T. Saveur
S. Weegink

to obtain the degree of

Bachelor of Science
in Computer Science and Engineering,

at the Delft University of Technology.

Project duration:	April 20, 2020 – July 10, 2020	
Client:	C. Doerr,	Cyber Security and Enterprise Security
Coach:	S. Picek,	Cyber Security Research Group
Bachelor Project Coordinators:	O. Visser,	Distributed Systems Group
	H. Wang,	Multimedia Computing Group

TU Delft

Preface

We chose this project for our Bachelor End Project since we wanted to work on a challenging problem that interests us. An important motivator was also that this is a product which is going to be used after it is finished. We are happy with the result of our project, as it can handle the number of clients that our client had aimed for, and contains the features that he wanted to have. Our product can be used for researching purposes in the Cyber Security field and we are hoping that it achieves the desired goals our client has.

We, as a team, had a great time and found it interesting to work on this BEP, especially since we also had to use technologies which were new to us and work in an area that was unknown to us before this project. It required quite some research to understand how everything works and what the project can be used for, since the Bachelor did not cover all knowledge that was necessary. This, however, made it both more challenging and fun. We learned to work with a lot of new useful tools and gained a lot of practical experience that will definitely help us in future projects.

We want to thank our client, Christian Doerr, for his help during the project on topics that we did not understand yet and his suggestions on how to solve or deal with any problems we encountered during the development. His feedback on our implementation and design decisions were also very useful.

We also want to thank our coach, Stjepan Picek, for his useful feedback on how to deal with the problems that we encountered during the development and on how to communicate this with our client. Furthermore, his feedback on the report, and our progress overall, helped us to deliver a good product that we are proud of.

While there is still room for improvement and more features, we are satisfied with the result that we delivered, how we developed it with each other and the experience overall.

R. Huisman

A. Jeleniewski

S. Kaptein

T. Saveur

S. Weegink
Delft, July 10, 2020

Summary

The internet consists of many networks connected by the BGP protocol and can easily be manipulated by a hacker. Every day, hackers reroute internet traffic and use that to impersonate entities such as companies, devices and humans. To detect this rerouting, which is also know as a route leak, a BGP monitor can be used. A BGP monitor checks whether the local route is correct by comparing the local route to the route from hundreds of devices. The lower the percentage of the routes that are the same, the bigger the chance there is a route leak. Hence, a user can tell whether a hacker manipulated the route. This project aims to build a BGP monitor application, that will ease the process of finding whether a hacker rerouted a part of the internet. To achieve this, the Internet Monitor application has been developed.

In the Internet Monitor, an admin can add different types of tasks that will be run by the devices. This input will then be transmitted to the backend and to the database, where they will be stored. After some time, the backend will send out the tasks to the nodes. The nodes will execute the task and send back the results to the backend. These results can then be used to check whether the information from BGP servers or the local result is the same, and thus not manipulated. Apart from this, Internet Monitor also offers the possibility to see the map of all the real-time traffic between devices.

Contents

1 Introduction

The internet has been around for a long time and has been of great importance to humanity over the past few decades. However, as with any system of age, the internet also has its problems. One of these problems is in the way networks are connected, which is done using the Border Gateway Protocol (BGP). “BGP contains no security protections, and essentially every network owner in the world can impersonate other companies and reroute their network traffic” [5]. Of course, any victim of such an impersonation wants to be notified as soon as possible. In the past, one could use a tool called BGPMon[1] to detect a change of route. However, this tool is scheduled to be shut down, so the Internet Community will lose an important tool to monitor the health of the Internet and detect incidents.

From our research report, it became clear that at the moment there exists no tool that can replace all the features that BGPMon offered. However, Internet Monitor has the potential to be a worthy replacement, as it encapsulates the functionality of BGPMon with many extra features requested by our client. Internet Monitor does not only have a real-time map of the traffic from the devices but also allows for running many different tasks on devices. To implement all of this behaviour efficiently, we divided the program into four parts:

- Measurement Endpoint (ME)
- Port Forward Server (PFS)
- API Backend (API)
- Visualisation

These four components combined will create a program, which is capable of doing the following:

1. Admins are capable of creating assignments
2. These assignments will be sent to the backend, where they are stored in a database
3. The ME receives tasks from the backend, executes the task and sends back the results
4. These results will be shown in the visualisation

To run this program, the visualisation and API backend needs to run on a server. Optionally a user can install our ME program on any Linux- or Windows-based device they want. Having the ME installed, allows this device to exchange information with the API. Once the ME has been started, it will connect to the API and start running assignments as provided by the API. When the assignment is done, the result is sent back to the API and the ME will continue with the next assignment.

This report will give a complete overview of the Internet Monitor program created during the Bachelor End Project. In chapter 2, some basic terminology and background information will be discussed. This terminology will be important when reading through the rest of the report. After having discussed this, the design of the different components will be displayed in chapter 3. This design consists of both language and design decisions. To give more insight into these decisions and how the program will

[1] `https://www.bgpstream.com`

function, we have created a set of architecture and sequence diagrams. Of course, these diagrams can never cover how exactly a component works. That is why, in chapters 4 to 7, each of the individual components will be elaborated even more. In each chapter, the component will be split into several sub-components, which will all be explained in detail. Also, we will evaluate and discuss the SIG feedback for each of the four components. After all the main components have been discussed, we will go on with explaining the port forward protocol design in chapter 8. This protocol design ensures that communication is secure with the Port Forward Server while performing the Port Forward assignment. After the protocol design, we will discuss our process in chapter 9, where we will go into further detail on the development strategies, communication and problems we encountered. Next, in chapter 10, we will discuss the ethical implications of the product. Lastly, in chapter 11, we end with a conclusion.

2 Background Information

In this chapter, we will briefly discuss some of the terminology and background information needed to understand this report. First, we will discuss some of the research we have done before we started working on the actual product. This consists of an explanation of the Border Gateway Protocol (BGP) and a competitor analysis. After that, we define some terminology, which is useful to understand this report. Finally, we will end with some information about the different types of assignments we have implemented and what they do.

2.1. BGP

"The Border Gateway Protocol (BGP), which is used to distribute routing information between autonomous systems, is a critical component of the Internet's routing infrastructure" [8]. This critical component will form the foundation of this project, hence some research must be done on the subject. Questions as to how it works, how one can use it to monitor the internet and many more will be answered in the coming sections. After these questions have been answered we will end with the questions as to why BGP is a fundamental component for this project.

2.1.1. Autonomous Systems

Before discussing the protocol, we need to define what an autonomous system (AS) exactly is. We use the definition of an AS from Hawkinson and Bates: "An AS is a connected group of one or more IP prefixes run by one or more network operators which has a **SINGLE** and **CLEARLY DEFINED** routing policy" [6]. The term prefix can be thought of as a group of one or more networks. This is since the IP prefix might be `100.255.159.3/32`, which is only one address, while `100.255.0.0/16` is a group prefix with members `100.255.0.0` until `100.255.255.255`. Routing policy is defined as the way that routing decisions are made on the internet. ASes exchange routing information that is subject to this policy [6].

A prefix belongs to only one AS, which is due to the fact that there can only be one routing policy for traffic directed to each prefix [6]. Each AS has a unique number of the form ASX, with X a 32-bit integer introduced by Vohra and Chen [16].

2.1.2. Protocol

BGP is designed to exchange routing information among ASes on the internet and it is also able to make routing decisions. Hence, it can be classified as a path vector protocol; a protocol that maintains the path information that gets updated dynamically. In other words, the path-vector protocol is a significant factor in determining the routing table. Therefore, the path-vector protocol plays a big role in how BGP operates, which will be discussed in section 2.1.3.

BGP may be used in two ways: internal and external, which are called iBGP and eBGP respectively. iBGP means that the "BGP connection is between internal peers" [13]. Internal peers are two BGP server which make a connection between them and are in the same AS. On the other hand, eBGP is the connection between servers which are not in the same AS [13]. This external connection is used to create and update the paths in the routing tables. So eBGP is mostly used when determining the routing for the internet and it can, therefore, also be used to monitoring the state of the internet. Contrary to this, iBGP is only used for routing within an AS, which is not important for our project.

2.1.3. Operation

Two BGP servers establish a Transmission Control Protocol (TCP) connection between them. TCP is a highly reliable host-to-host protocol that enables communication in a network of computers via so-called packets. These packets contain the data that a user or an application transmits and some other information needed for the protocol to work [12].

The BGP servers exchange information to open and confirm the connection, after this the entire BGP routing table will be shared amongst each other. If the routing tables change later on, incremental updates are sent. This causes all BGP servers to save the entire BGP routing table of all its peers for as long as the connection is open. There is also an exchange of notification messages when errors occur or keep-alive messages to keep the connection open [13].

A route is defined as “a unit of information that pairs a destination with the attributes of a path to that destination” [13]. BGP servers advertise routes to each other via **UPDATE** messages. Each server is allowed to add or modify path attributes of the route before advertising it to its peers. There is also a mechanism for announcing that a route is no longer available. This can be done via advertising the IP prefix in the **WITHDRAWN ROUTES** field in the **UPDATE** message, advertising a replacement route or closing the BGP connection. The last one removes all routes that the pair of servers advertised to each other [13].

An example can be that the IP prefix 100.255.1.0/24 is inside ASX. This AS announces via BGP to its peers that they can reach the range of IP addresses via this AS. Any traffic outside of ASX is thus routed via BGP to ASX.

2.1.4. Attacks on BGP

With the general information about how BGP operates, we can discuss several possible attacks, also called BGP Hijacks. The scope for this is just to discuss a few possible attacks, not all of them and not going into a lot of details. The purpose is to show that these attacks are possible and that they actually still happen, which is important for the project. *BGPsec*, the protocol that is supposed to fix the vulnerability issues from BGP [9], is out of the scope of this project. It can only fix some of the security issues and not all of them: “even secured with *BGPsec*, BGP still has inherent security vulnerabilities. In particular, traffic can still be hijacked” [9].

The risks in BGP come from the fact that BGP was designed with no security or data protection in mind. There are no mechanisms in BGP that protect against attacks that modify, delete, forge, or replay data [10]. All of these attacks can disrupt network routing behaviour, which can have severe consequences as we will see later on. There are three main vulnerabilities in the protocol that enable attacks, according to Murphy [10]:

1. BGP has no internal mechanism that provides the protection of the integrity of the messages. Furthermore, there is no protection mechanism for the freshness and authenticity of peer messages in communication between peers.
2. BGP has no mechanism that validates the authority of an AS to announce their IP prefixes.
3. BGP has no mechanism to ensure the authenticity of the path attributes announced by an AS.

Next, we will look at some attacks to see the severity of the insecurity of BGP. A lot of attacks have happened and are still happening, which our project can visualise.

Pakistan Telecom
Pakistan Telecom brought down YouTube[1] for two hours when it tried to restrict access to the website. They did this by advertising new routing information, which was propagated to other ASes via BGP. The intention was to not announce this new route to upstream providers. However, this did happen and because the routing information was never verified, the other ASes accepted and forwarded it. This resulted in a DDoS attack on themselves since all traffic to YouTube now had to go through Pakistan Telecom [14].

MyEtherWallet
MyEtherWallet[2] was the victim of another BGP hijack. Users that wanted to login saw an unsigned SSL certificate when connecting to the website, but many users clicked through the warnings. Doing this rerouted the traffic to a malicious server in Russia, which emptied their Ethereum wallets. The attack was done by hijacking DNS requests to the website and making the Russian server look like the legit owner. They did this by compromising an upstream ISP and used it to announce a subset of IP addresses to other networks. BGP hijacking was used to intercept those requests [4].

Other
There are lots of other examples. Another interesting case is Rostelecom, a Russian telecommunication company. They (wrongly) announced prefixes for different ASes belonging to financial institutions, telecom companies, and other organizations [15]. It is not known if this incident was intentional or not. With a bit of research, lots of other attacks can be found against BGP and since they can still happen, it is just a matter of time to wait for the next big attack.

2.1.5. Project Relevance
As mentioned before this project revolves around BGP, but *why* is this project needed in the first place? BGP is one of the critical components for routing the internet and BGP is, therefore, of interest for many. For instance, a hacker might want to manipulate the routing such that the internet traffic is forwarded to his or her location. This gives the hacker a lot of intelligence and possibilities to perform attacks. This risk of attacks leads to a demand for some sort of application to monitor BGP so that it can be seen whether the path to a domain is correct or not. When the path is incorrect we call this a mismatch. If there is a mismatch it might be a hacker who rerouted the path and using the BGP monitor this adjustment in routing can be found quickly. So, the application allows monitoring the state of the internet, since the fewer mismatches that happen the better the state of the internet is. So far, a few BGP monitors exist. However, as became clear during competitor analysis (section 2.2), not one has all the features the client wants and most are not user friendly. Hence, the end product of this project will be a free BGP monitor that aims to provide a user-friendly application that can do all features that someone could want in a BGP monitor.

2.1.6. Conclusion
In this chapter, we discussed how BGP works by answering questions related to the topic. Questions as to what the difference between iBGP and eBGP is, how BGP works and what an autonomous system is. Then we continued on about what the relevance of the project is and what goal we want to achieve. But how we will use the BGP research to achieve this goal is maybe the most important question, which has not been answered yet. In the end product, we should be able to display the mismatches, which will be done using the BGP protocol. As mentioned in section 2.1.3, the BGP servers will make a routing table, all these routes will be compared to the history and expected routes to display the mismatches. To build a system that can display these mismatches, the research done in this chapter will come in useful.

[1] `https://www.youtube.com/`
[2] `https://www.myetherwallet.com/`

2.2. Competitor analysis

To better understand the market in which the product will be deployed, we have to look at similar products on the market. In this market, there are several competitors such as: *RIPE Atlas, BGPStream (2 variants), BGPMon, and RIPEStat BGPlay*. We look at the features and the limitations for each competitor. After that, we conclude what features we will add to create an improved product compared to our competitors.

2.2.1. RIPE Atlas[3]

RIPE Atlas uses a global network of probes that measure Internet connectivity, reachability and provides a way to understand the current state of the entire internet. It allows users to continuously monitor network reachability from certain points. The probes that do this monitoring allow for the following types of commands:

- Ping
- Traceroute
- SSL/TLS
- DNS
- NTP
- HTTP

A user can investigate and troubleshoot connectivity issues by using connectivity checks. He/She can also create alarms using status checks, these can be used on their monitoring tools. This website also allows for checking the responsiveness of the DNS infrastructure and IPv6 connectivity.

The main advantage of RIPE Atlas is that there are a lot of probes around the world and the data they collect is easily accessible and easy to use.

Limitations

- **Users will need to connect a proprietary measuring device to their router.**
  A software-only solution would be ideal.
- **Users need to earn credits to perform user-defined measurements.**
  Doing user-defined measurements without spending credits would be better.

2.2.2. BGPStream[4]

BGPStream shows a stream of alerts concerning hijacks, leaks and other outages of BGP. It does this by showing a list and a map of all outages. No installation of any kind is required for this website. However, the website does not have any other features.

Limitations

- **There is no way to perform actions like traceroute, ping, etc.**
  It should be possible for a user to perform these actions since they can give great insight into the internet.
- **There is no mention of how to retrieve the data.**
  It should be clear for every user on how to use the data gathered because they can be used to further the development of the program.

2.2.3. Caida's BGPStream[5]

Caida's BGPStream is an open-source C/C++ and Python API for live and historical BGP analysis, which is still actively being worked on. The API is split up into two different programs. LibBGPStream and PyBGPStream, the first being for C while the latter is for Python. These libraries can be used to send and receive requests. LibBGPStream also includes BGPReader, which can be used in the command line to print information about the BGP data.

[3]`https://atlas.ripe.net/`
[4]`https://bgpstream.com/`
[5]`https://bgpstream.caida.org/`

Limitations

- **It is unclear how to use/see the data on the website.**
  It should be possible for a user to see some of the data on the website.
- **The data that can be obtained is not represented in a human-friendly manner.**
  Humans should be able to understand what everything means, in order to get a better understanding of what their nodes are doing.

2.2.4. BGPMon[6]

BGPMon provides information to determine the stability of the user's networks and potential risks to data. It also sends a notification in case of an interesting path change (which might mean that network security has been breached). It contains both an API and a monitoring tool. This tool can be used to check certain prefixes for alerts, and also keep an eye on the location where the alerts were coming from. The API can be used in other applications.

Limitations

- **An account is needed.**
  Someone should be able to see what and how before creating an account, an account should not be a necessity.
- **Only 5 prefixes for free.**
  The program should be free to use.

2.2.5. RIPEStat BGPlay[7]

BGPlay is an advanced RIPEStat widget that visualises BGP routing information. This routing information is shown in a detailed and large graph. This graph will show a lot of different types of data, like node announcement and withdrawals. The source code is also freely available.

Limitations

- **The widget is very slow because of the amount of information.**
  The amount of information on the screen should not overload a human and the computer.
- **The information presented is not easy to read and in an understandable format.**
  The information that is presented on the website should be easy to understand for the user.

2.2.6. Our Approach

After looking at some of our competitors we have got some ideas of what we want to improve in comparison to our competitors in terms of being user friendly.

No Proprietary Hardware Needed In order to contribute to RIPE Atlas the user will need a piece of proprietary hardware to connect to their routers, which we believe drastically lowers the number of users that will contribute. We want to ensure that as many people as possible can use this product and supply the data needed to keep our project online and updated. This is why we will allow this program to be run on as many devices as possible, like Raspberry Pi's. In contrary to Proprietary hardware, which we would have to supply and pay for.

No payment needed Because we want as many people as possible to use and contribute to our project, it will be completely free to use. In contrary to BGPMon, where a user will have to pay if they want to check more than 5 prefixes.

Fast and Easy For everyone to be able to use our product, it needs to be fast and easy to use. There is no complicated installation procedure or account creation. It is simply plug-and-play, the user downloads the software for their respective device and runs it. The program will register itself and start executing if the user wants to gather more information about their node, the user can simply log in to the website where all data is displayed in an overview for the user to read. Someone that does not have a lot of knowledge about computers should also be able to install the software onto his or her own

[6]https://bgpmon.net/
[7]https://stat.ripe.net/special/bgplay

device, as described in the new paragraph. We also believe that having a map with the BGP data as a front-page will ensure that people can understand the data and what these nodes are doing.

Good documentation A lot of websites seem to lack documentation, so gathering and contributing data is made more difficult than it might be. That is why we want to make good documentation on the different parts of our software, like the installation and the API. For this purpose, there will a comprehensive guide for use and of the application on the website. In addition, the API specification is written to the OpenAPI 3.0 standard. The backend code is all documented using XML documentation.

Limitations of our Approach
No user-defined measurements For now we will only allow admins to add new assignments.

2.3. Terminology

Nodes Nodes are created when you install the measurement endpoint, this means that one device can have multiple nodes. The measurement endpoint will connect to the backend and exchange assignments and results. To read more about what measurement endpoints do, look at chapter 4.

Restrictions Restrictions are parameters that can be set for an assignment, they indicate which nodes are allowed to do the assignment. There can be multiple, even of the same type, restrictions on a single assignment. We implemented five different types of restrictions:

- **Architecture restriction**: only nodes which match the specified architecture are allowed to do the assignment
- **Operating system restriction**: only nodes which match the specified operating system are allowed to do the assignment
- **Region restriction**: only nodes which are in the specified region are allowed to do the assignment
- **IP restriction**: only nodes which have the specified IP are allowed to do the assignment
- **Node restriction**: only nodes with the specified ID are allowed to do the assignment

Forbidden assignment Forbidden assignments are configurable for each node individually. This allows a node to block certain types of assignments from being assigned and run on the node.

IP blacklist The IP blacklist is a list of IP ranges which indicate which IP ranges are blocked from retrieving assignments and posting results. This list is defined by the admin.

2.4. Assignments

Currently, we have implemented 8 different types of assignments. In this section, we will explain each of them.

DNS This assignment will perform a Domain Name System (DNS) lookup for the domain provided. It will, for example, return a domain resource record like an A record for IPv4 or AAAA for IPv6.

WHOIS WHOIS will query a special WHOIS server associated with the domain and return various information about it. For instance, the creation date and the registrant of the domain [1].

Traceroute Traceroute determines the route to the destination specified using special packets. This can be a domain or IP address and the output gives you the path with intermediate routers and the round-trip times to each router on the path [2].

Port Forward Port Forward captures traffic on a ME and forwards the traffic to a remote Port Forwarding Server. This server determines what to reply to the packets and sends it back to the ME, which injects the replies in the network. This makes spoofed communication possible, where a potential attacker can contact a computer running an ME. For the attacker, it looks like the computer is replying, but in fact, it communicates with the server.

PCAP Capture PCAP Capture captures all incoming traffic to the computer and saves the data in a file, which is uploaded to the server.

UPnP Port Opening UPnP will open the port specified for TCP and UDP connections. Then it waits for the specified duration after which it closes the port.

Banner Grabbing Banner Grabbing means that an ME connects to a specified domain and port via TCP and reads any incoming data for a short amount of time. This information is then returned to the server.

Certificate Grabbing This assignment can only be used on domains that support HTTPS. It will download the certificate and upload it to the server.

3 Design

This chapter will go in-depth into the design of the application. This was done in the first two weeks of the project before any implementation was done. First, we will discuss the different language and design decisions. After which, we show the architecture design. Thirdly, we will show the different protocols within the application. Finally, we show several sequence diagrams displaying flows in the product.

3.1. Language/Design Decisions

In this section, we will explain our decisions regarding languages and frameworks for the backend, visualization and measurement endpoint. Some decisions were easily made, while for other decisions we had to make a consideration.

3.1.1. BackEnd

For the backend, we decided to go with an ASP.NET Core web API written in C#. We also considered Java, since the visualization is going to be written in Java. But because several members of the group had experience writing APIs in ASP.NET and .NET Core also builds for Linux, we eventually decided to go with .NET Core 3.1. Another plus of .NET is entity framework. A database framework that allows for easy access to databases via a single API, which also allows for easy use of different database providers without sacrificing code readability.

3.1.2. Database

For the database, we decided to go with two separate databases. The first database is for the node data, such as configurations and assignments. The second database is for the results of the assignments. This is because of mainly two reasons. Firstly, the node data is relational while the result data is not (at least not all of it). And secondly, for the result data, we want to be able to do large-scale processing on millions of rows.

For the node database, we looked at a couple of options. This database only had one requirement; it needs to able to quickly retrieve assignments for nodes. So, we decided to go with a database that we have experience with and one which is up-to-date. Therefore we chose PostgreSQL V 12.x. This is to prevent latency while handling requests and to prevent requests backing up. We also had a look at MongoDB, our client, however, advised against this.

For the assignment result database, we had the requirement to be able to process millions of rows. Since we had no experience beforehand with big data, we used the recommendations of the client. Therefore, we went with Hadoop. However, just Hadoop was not enough, so in addition, we went with, according to the internet the best tool for processing large-scale data, Spark. Spark replaces the map-reduce and the yarn from Hadoop.

3.1.3. Measurement Endpoint

There was little discussion about the language that should be used for the MEs, because of the requirements set by the client. The client suggested that we use some libraries and techniques that are easiest to implement in C or Go. For example, receiving, editing and forwarding raw packet data. This is best done in a low-level language like C or Go, also because these 2 languages have many libraries for the tasks we need to do. We decided against C for a couple of reasons. Firstly, because of security. Because all memory management in C needs to be done by the programmer, this can create very dangerous bugs that can lead to security flaws in the system. One of the most dangerous exploits being buffer overflows, which can easily happen when the programmer makes a small error in the code. Secondly, Go is easier to write and understand than C and therefore more maintainable. Go provides very similar functionality to C, but Go is a much more programmer-friendly while still providing comparatively fast speeds when compared to similar languages like Python.

3.1.4. Port Forward Server

The decision to write the Port Forward Server in Go was quick to make. The Port Forward Protocol was already written in Go for the ME, which means that we can reuse the same code. Furthermore, Go is excellent for handling many connections at the same time. Since the server is relatively small with only one main task, the executable is more lightweight when using Go than if we were to use the .NET core for example.

3.1.5. Visualization

For the visualization the client came with the idea to write a web-app in Java using Vaadin[1] version 14. We decided on using Vaadin because of the style of the apps made, and the ease of use. We are also able to use JavaScript libraries from the Vaadin application, which is very important when working with maps and other interactive components.

[1] `https://vaadin.com/`

3.2. Architecture Design

For developing the components of the system we started by designing the components. We will first discuss the design of the backend, after which we will discuss the design of the measurement endpoint. Finally, we will explain the design of the visualisation.

3.2.1. Backend

In figure 3.1 our component diagram for the backend is shown. The backend will consist of a lot of different services and controllers, all of which are managed by one main component; the ASP.NET Core client. The goal of this client is to handle incoming requests. Everything is also connected to a DB, this is either the Kafka cache or the general DB. The general DB is used to store data about every node in the system, the measurement endpoints.

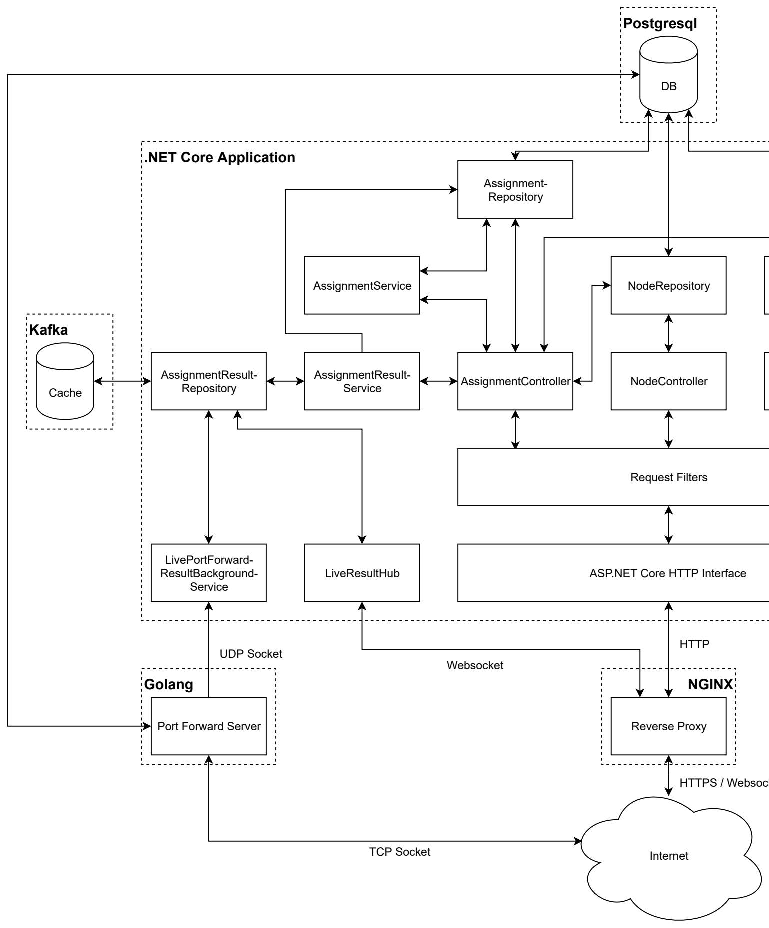


Figure 3.1: Component diagram of the Backend.

Filters

Filters used to validate incoming data, catch exceptions and wrap results.

LiveResultHub

A WebSocket for the live data. The visualisation connects to this hub to show the map.

LivePortForwardResultBackgroundService

Background service which opens a socket and allows the port forward server to send data to this socket.

Controllers

The controller looks at the incoming request from the client and uses the corresponding service to handle the request.

Services
The service will receive the requests and can then perform logic operations and interact with the database contexts. The Node Service is our main component; it will handle everything that has to do with the Nodes, it can retrieve their information from the DB, handle their assignments through the **Assignment Controller**, and handle the data from the Nodes through the **Node Data Controller**.

Contexts
The contexts are used to interact with the database.

3.2.2. Measurement endpoint

In figure 3.2, the component diagram for the measurement endpoint is shown. The measurement endpoint consists of one main controller, heartbeat, which will execute all of the assignments that it obtains from the Go HTTP Client. The HTTP Client gets these from the REST API calls to the backend.

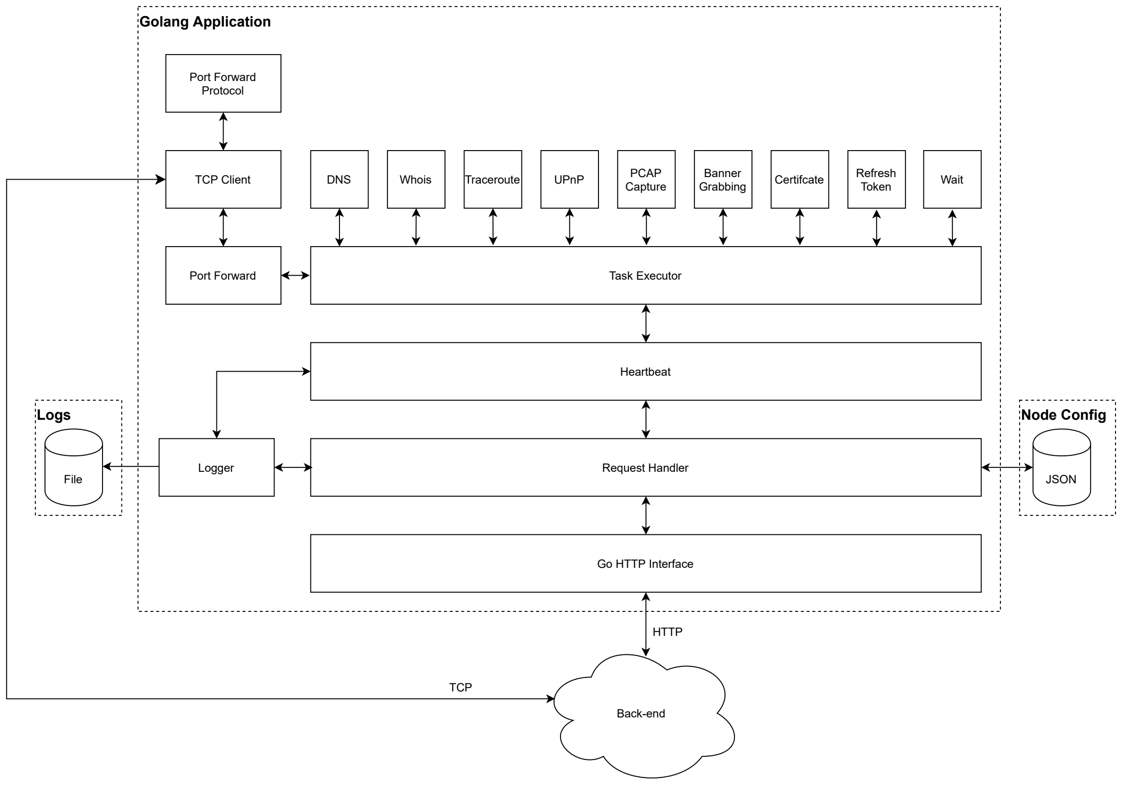


Figure 3.2: Component Diagram of the Measurement Endpoint.

Heartbeat
Service that continuously receives tasks from the backend, executes these tasks and sends back the result. The heartbeat protocol will be explained in section 3.3.

Task Executor
Parses the JSON tasks received from heartbeat, calls the correct component which will execute the task and returns the result to the heartbeat.

TCP Client
When the port forward task is being executed, packets need to be forwarded to the backend, which is done by the TCP Client.

3.2.3. Visualisation

The visualisation contains very few components (as can be seen in 3.3) to make the whole program work. The components that are there rely on the data gathered from the API to fill in the screens. Several of the big components will be discussed here.

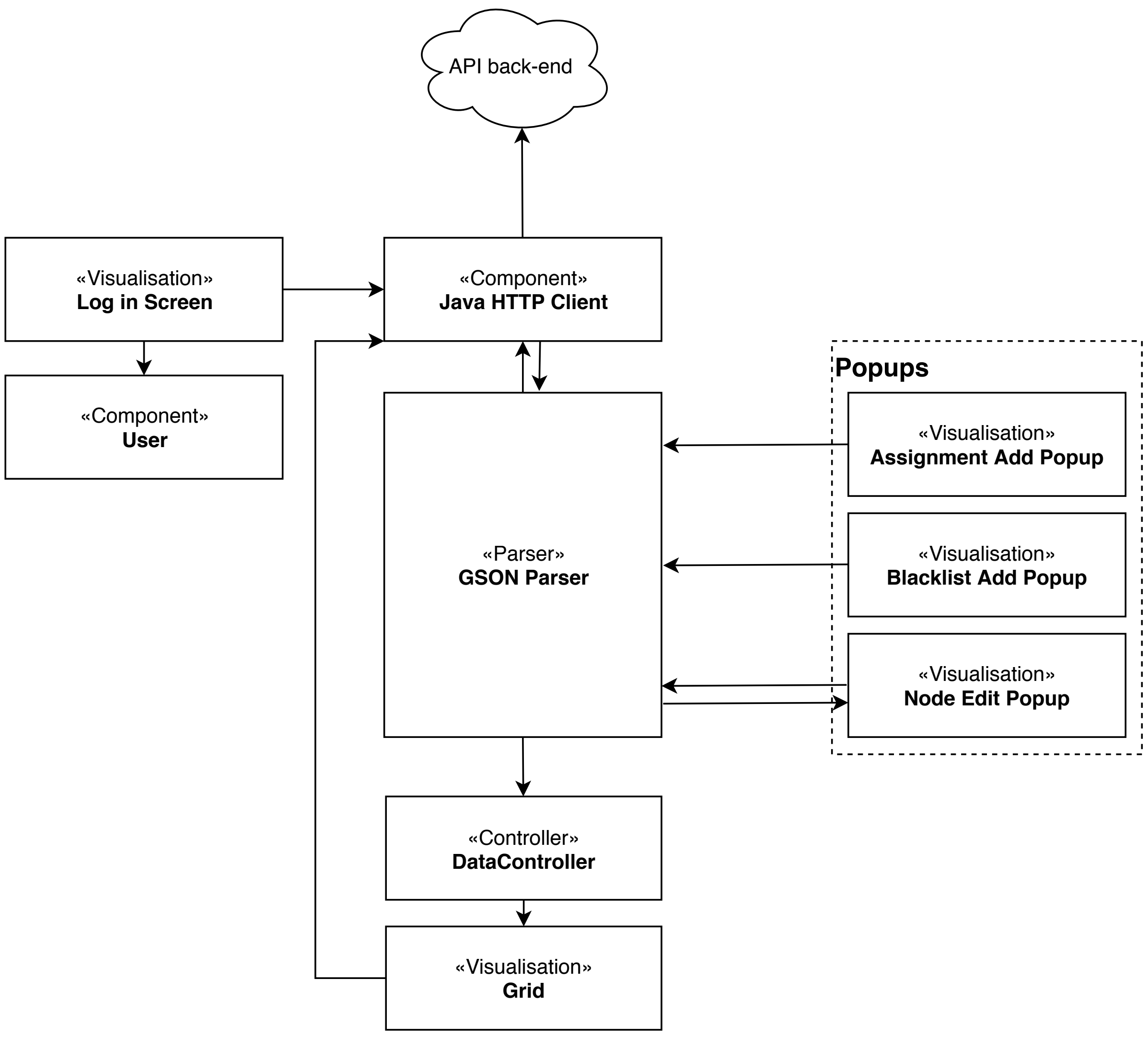


Figure 3.3: Component Diagram of the Visualisation.

Java HTTP Client
This is the REST Client, it connects to the API and uses GET, POST and DELETE requests to get data from the server, more about this in section 7.2.3.

GSON Parser
Using GSON we parse the data from the server to show it on the screen, or we parse the data going towards the server.

Popups
The popups are used to show a form for adding or editing things in the backend.

Visualisation
These are the items shown on the screen.

3.3. Heartbeat Protocol

This section will further elaborate on the heartbeat protocol. This protocol is used for receiving assignments. The Nodes will request new assignments based on a heartbeat protocol. This means that it will let the server know that it is still active and ready to perform new tasks in a timed interval. The protocol is described in figure 3.4.

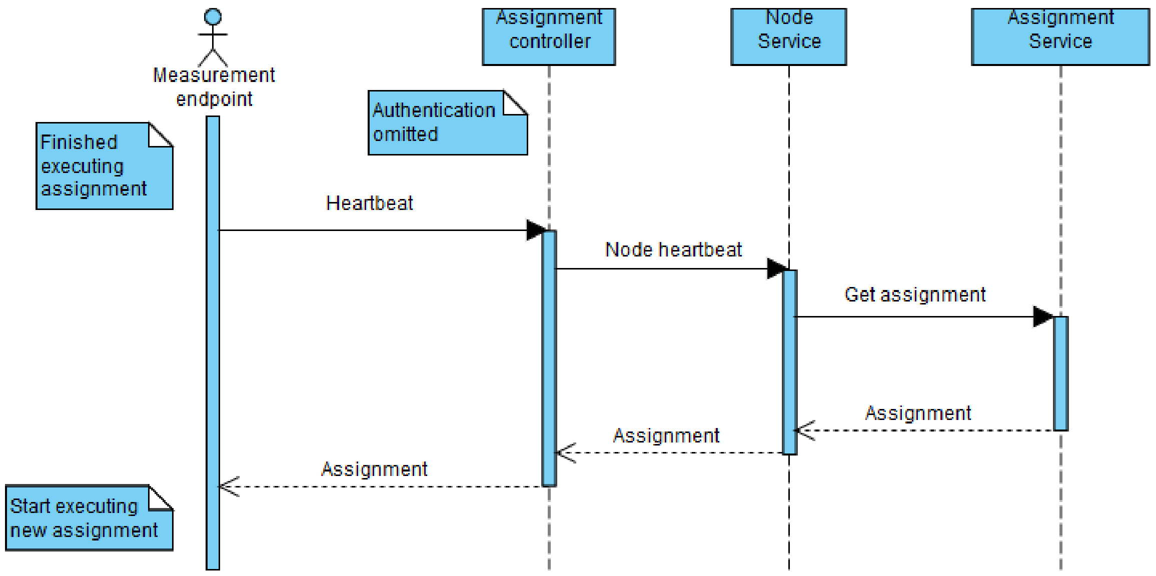


Figure 3.4: The heartbeat protocol

The Node will, after it has executed all tasks, request the Assignment Controller for a list of tasks, this message is forwarded to the Node Service (after authorization is successful). The Node Service will request all tasks from the Assignment Service. The list of tasks is then relayed back to the Node.

3.4. Sequence Diagrams

This section will explain the sequence diagrams of our program.

3.4.1. Task Allocation and Execution

On the following page the sequence diagram for task allocation and execution is shown, this diagram will show how a measurement endpoint requests a task, and the response it gets. It will also show how the results are handled.

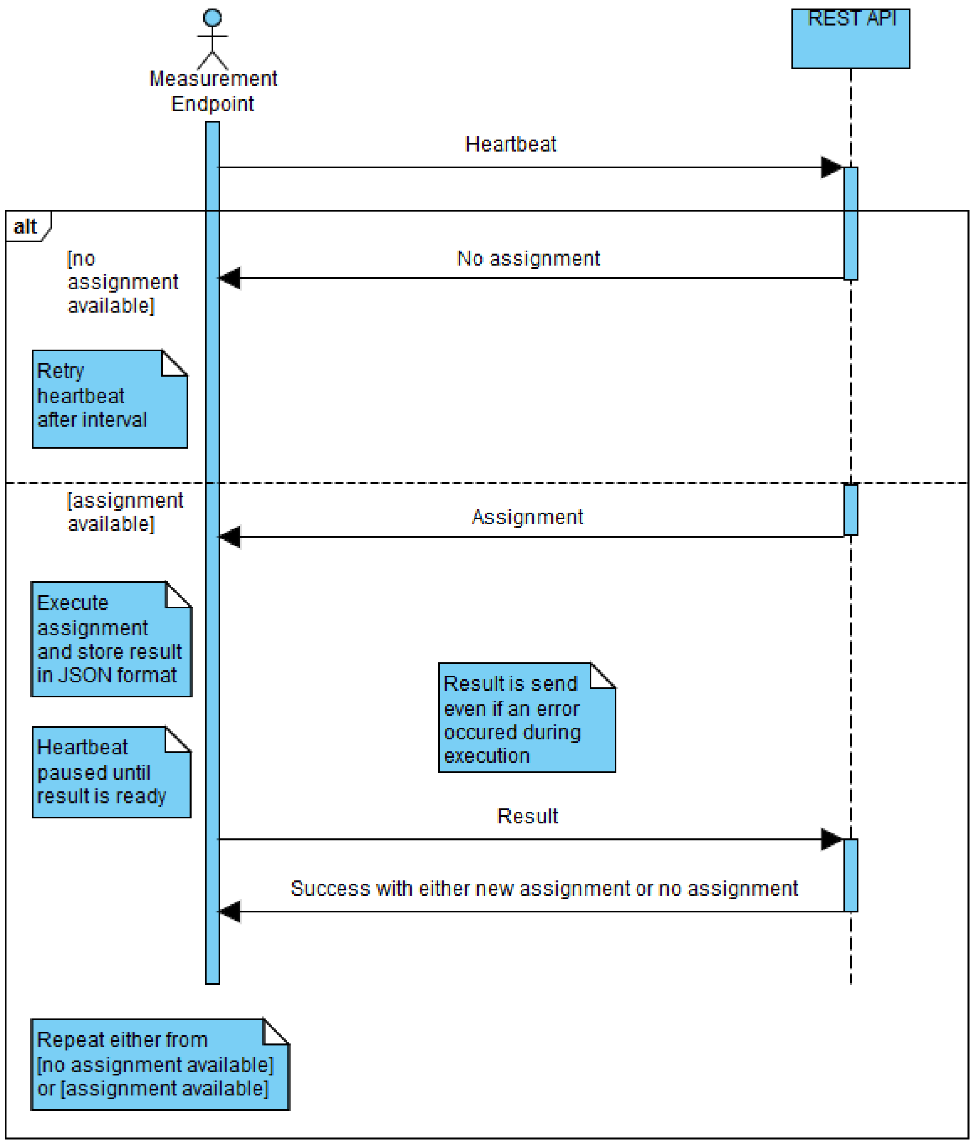

Measurement Endpoint
REST API
Heartbeat
alt
[no assignment available]
No assignment
Retry heartbeat after interval
[assignment available]
Assignment
Execute assignment and store result in JSON format
Heartbeat paused until result is ready
Result is send even if an error occured during execution
Result
Success with either new assignment or no assignment
Repeat either from [no assignment available] or [assignment available]

Figure 3.5: Task allocation and execution

First, the measurement endpoint, from now on referred to as Node, will send a request to the REST API. This API will handle the request and will return either:
In case of no Available Assignments the API will forward that there are no assignments currently available. This means that the measurement endpoint will try again after an interval of time.
In case of Available Assignments the API will return an assignment, which the measurement endpoint will execute and the result will be returned. After the result has been returned to the REST API it can either a new assignment or, if there are none available, send that there are no assignments.

3.4.2. User Login

A user should be able to login to the website. How the implementation works can be seen in figure 3.6.

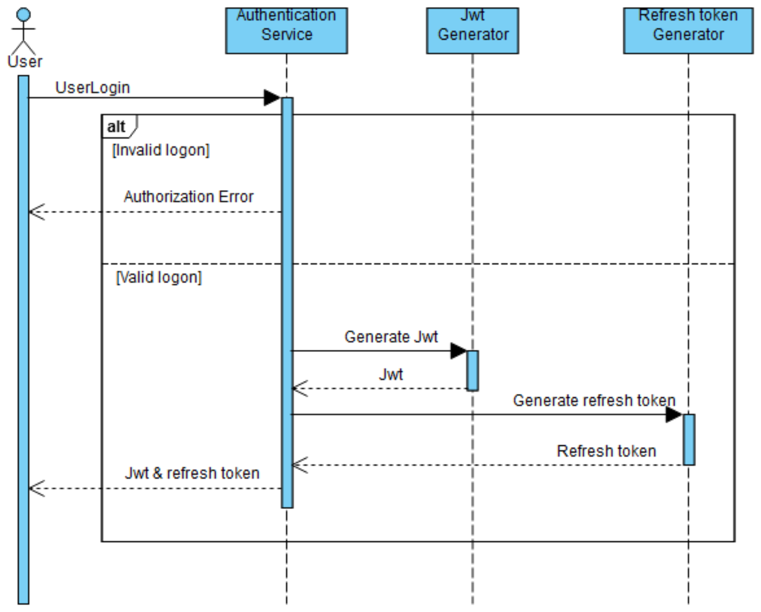


Figure 3.6: User login

The user will send a request to the Authentication Service with its credentials. The Authentication Service will return:
In case of an Authorization Error the error will be returned to the user. This error will happen because of invalid logon credentials.
In case of an Authorization Success the Authentication Service will request the JWT Generator for a JWT token, and it will request from the Refresh token Generator a Refresh token. Both the JWT and Refresh Token are then returned to the User.

3.4.3. Register a Measurement Endpoint

When a Measurement Endpoint (Node) is first installed by a user it will need to be registered. This process can be seen in figure 3.7.

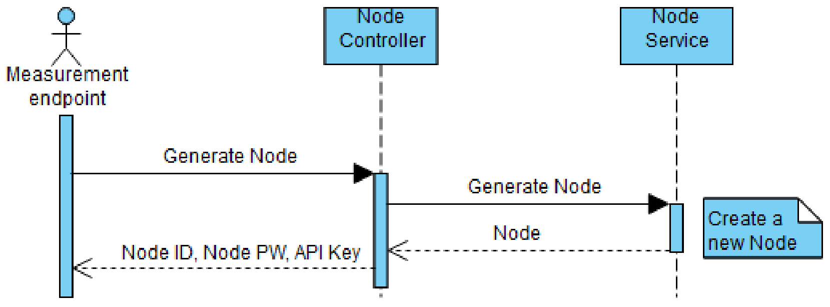


Figure 3.7: Registration of a new Node

The newly configured Node will send a Generate Node request to the Node Controller, which will forward it to the Node Service. The Node Service will create the credentials for the Node and returns it to the Controller. This Controller will then send the credentials to the Node.

3.4.4. Data Retrieval

A user might want to be able to see the data that his or her Nodes have collected over time. This process can be seen in figure 3.8.

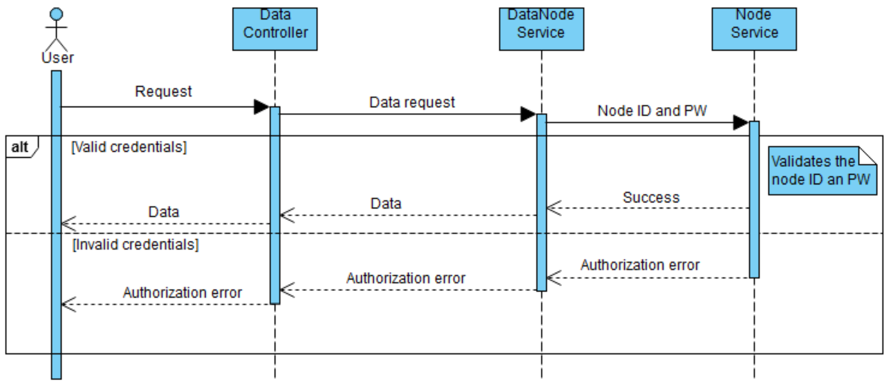


Figure 3.8: Data Retrieval from a Node

The user will send a data request to the Data Controller together with the Node ID and his or her password. The Data Controller will forward this data to the DataNode Service, which will forward the Node ID and password to the Node Service. The Node Service will then check the Node ID and will respond:
In case of an Authorization Success it will return a Success message to the DataNode Service, which will get the data and forward it to the Data Controller, which will forward it to the user.
In case of an Authorization Error the error is relayed back to the user.

3.4.5. Request New API Key

When the Node's API Key has expired it will need a new one. How this works is described in figure 3.9.

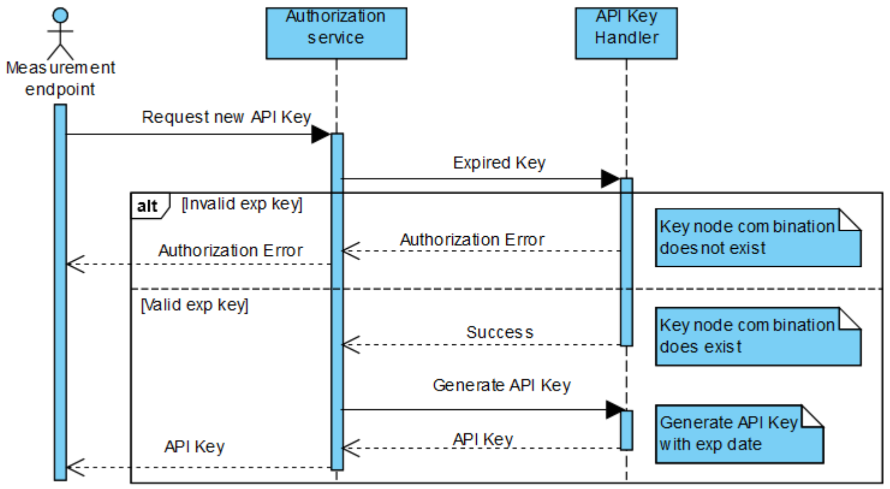


Figure 3.9: Requesting a new API Key

The Node will send a new API Key request to the Authorization Service. This authorization service will then send the Expired Key of the Node to the API Key handler which will return:
In case of an Authorization Error it will return an Authorization Error. This error occurs when the expired key is invalid.
In case of an Authorization Success it will return Success to the Authorization Service, which will request a new API Key from the API Key handler and returns it to the Node.

3.4.6. Authorization of a Node

Before the node does anything it will need to be authenticated using its API Key. This is described in figure 3.10.

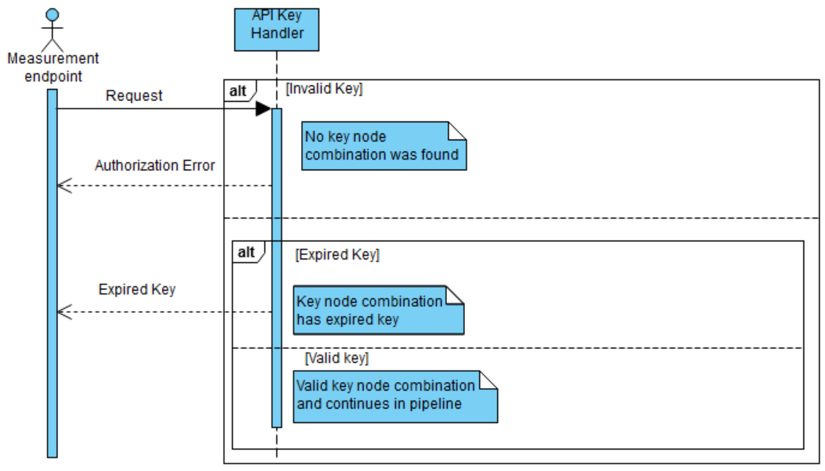

Measurement endpoint
API Key Handler
Request
alt
[Invalid Key]
No key node combination was found
Authorization Error
alt
[Expired Key]
Key node combination has expired key
Expired Key
[Valid key]
Valid key node combination and continues in pipeline

Figure 3.10: Authorizing a Node

The Node will send a Request to the API Key Handler, which will return:
In case of an Invalid Key an Authorization Error.
In case of an Expired Key a message stating that the key has expired.
In case of a Valid Key nothing will be returned, but the request continues in the pipeline.

3.4.7. Assignment Result

When a Node finishes all of its tasks it will want to return the results so we know what happened. A detailed overview of this process is shown in figure 3.11.

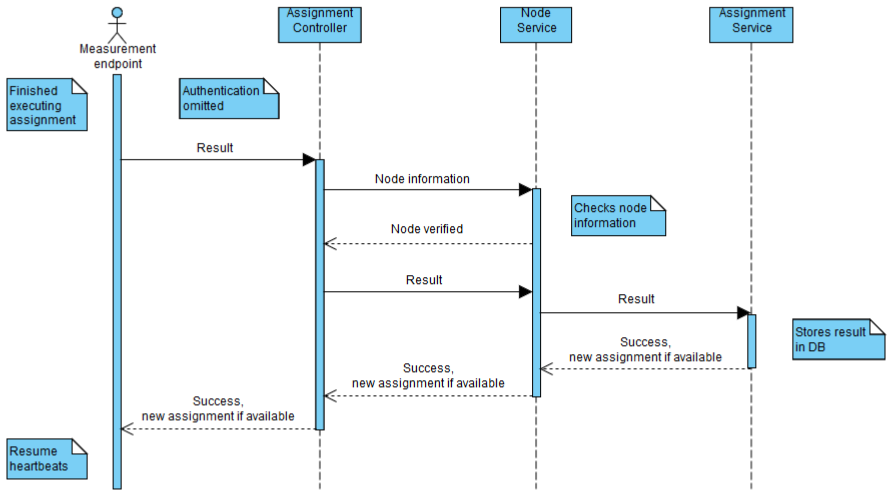


Figure 3.11: Assignment Result system

After the Node has executed all of its tasks it will send the results to the Assignment Controller, which which will, after Authentication, send it to the Node Service. The Node Service will check the Node information, and returns that the Node has been verified. After this, the Assignment Controller will send the result, which the Node Service will forward to the Assignment Service. The Assignment Service will then store the results in a DB and relays back a success message together with a new assignment, should one be available.

4 Measurement Endpoint

The measurement endpoint (ME) is one of the key components in our system. The ME is written in Go, which is also known as Golang. The ME serves one main purpose: executing tasks received from the API backend. To better explain how the ME fulfils its purpose, we will start by shortly explaining what all the components are and how they work. After that, We will also discuss the work that can still be done to further improve the product and what we would change if we had to do the project all over again. We will finish by drawing a short conclusion about the development of the ME and how we resolve the SIG feedback.

4.1. Components

The measurement-endpoint can be broken down into four components:

- **Heartbeat**, registration of nodes and execution of tasks.
- **Logger**, storing of results, errors and HTTP connections on the node.
- **Request handler**, handles the HTTP interaction between the node and the backend.
- **Result handler**, specifies a result object for tasks.

Please note that the *heartbeat* component captures almost all the measurement endpoint functionality and the other three are just small components to assist the *heartbeat*.

4.2. Heartbeat

The *heartbeat*, which is the main functionality of the measurement endpoint, is the component that makes sure the tasks received from the backend are performed. To be more precise, the *heartbeat* does the following routine:

1. Check if the node configuration exists, otherwise create one on the backend.
2. Fetch a new task from the backend.
3. Execute the task.
4. Send back the result of the task if needed.

This routine is repeated forever until the user shuts down the ME. While the routine is running, certain tasks can be performed. These tasks will be elaborated on in the coming sections. We will discuss which considerations were made concerning the implementation of the task, how we ended up implementing it and what issues we came across while implementing it.

4.2.1. DNS

At first glance, the implementation of DNS looked simple, it soon became clear it was the opposite. This complication came from the decision that the client also wanted to support the Windows operating system. No library in Go supports windows and, at the same time, is easy to use. In general, there were only two DNS libraries that could be used for the project. On the one hand, we have the library from

Miek Gieben[1] and on the other, we have the standard NET library[2] from Go itself. Both have their pros and cons. The library from *Gieben* has great support for all DNS records but lacks in providing support for windows and code-quality wise it is not the best library to use. The latter comes from the fact that the result objects returned should be handled separately and made into one big result. Similarly, the *NET* package also has its problems, it does, for example, not support all DNS records. This means that we would have to add all the other DNS records we wanted which would have taken around a week to implement, which is time we did not have. Therefore, we decided to use the DNS library from *Gieben*, which was also suggested by the client.

The DNS implementation can be split into two sections, *local* and *remote* DNS. Remote DNS is the main function used and makes the calls to the DNS library for all the DNS records specified. The responses received from those calls will be stored. Once all the results have been received, the results will be transformed into a large formatted string, which will be processed on the backend. Remote DNS was an additional feature and was added since we needed most of the logic anyways for the local DNS lookup. Local DNS lookup, on the other hand, was a requirement from the client.

Local DNS is the same as remote DNS lookup only now it first needs to find the local DNS resolver. Once this address has been found, it calls remote DNS with the address of the local DNS resolver. Remote DNS will then perform the actual lookup. So rather than running the DNS query at for example the google DNS server `8.8.8.8:53` it now does it at the local DNS resolver which might be `127.0.0.1:53`. Implementing the ability to find this local DNS resolver was quite cumbersome as a different method needs to be used based on the operating system. For UNIX systems a configuration file exists which specifies the local DNS resolver, while for Windows we had to parse the `ipconfig -all` command.

4.2.2. WHOIS

WHOIS is a query/response protocol used to provide information services to users, such as information about domain names[3]. The implementation for WHOIS was simple in the end, using the library made by likexian [4]. It queries the official IANA WHOIS server, located at [5] on port 43. There were, however, a few problems to get to the working implementation.

The problem with the WHOIS protocol is that it does not have a standardized format. This means that each domain queried can return a different result, making parsing difficult, if not impossible. After several discussions with both the team and the client, the decision was made to put all parsing for all assignment results in the backend. Because of this, the ME code remains 'simple' and only performs the given assignment, without having to worry about any data formats. All the complexity is thus shifted to the backend. This has as a consequence that the ME can do tasks faster and that the backend has to do more processing. An important reason for making this decision is that the backend can easily be updated, while an update for the ME is a bit harder to realize since it needs to be installed on all devices running the MEs.

We also had a look at using RDAP instead of WHOIS, since it addresses several of the issues that the WHOIS has, one of which is the lack of standardized format[11]. Everything about RDAP looked good, it responds in standardized JSON, has all of the features from WHOIS and several other improvements. However, the problem with RDAP currently is that a lot of RDAP servers are not added to the official bootstrap list of IANA[6]. This means that there is no way of (easily) finding the correct RDAP server for a domain. Because of this, we decided to drop RDAP and use WHOIS. When the bootstrap list is extended with more domains in the future, it will be the best solution to drop WHOIS and use RDAP instead.

[1]`https://github.com/miekg/dns`
[2]`https://golang.org/pkg/net/`
[3]`https://tools.ietf.org/html/rfc3912`
[4]`https://github.com/likexian/whois-go`
[5]`whois.iana.org`
[6]`https://data.iana.org/rdap/dns.json`

4.2.3. Traceroute

Similarly to DNS, traceroute also had the struggle to support multiple operating systems. In Windows there exists the native `tracert` command which uses ICMP but it does not offer support for TCP traceroutes. While for Linux there were libraries that supported both ICMP and TCP, but some did not offer enough customisation in the commands, had no documentation, or required the user to install some additional software. Eventually, we found the library from jacksontj[7] which has little documentation but offers a lot of customisation. This library doesn't compile on Windows, therefore, we separated the Windows and Linux code into two files of which only one will be included in the build depending on the operating system. To make sure that handling of the results on the backend would remain easy, we made sure that the Linux library returns the format exactly in the same format as is done by the windows `tracert` command. Lastly, we also added the option to run TCP traceroute on Linux, which can't be done on Windows at the moment.

4.2.4. Port forward

Port forwarding was the biggest and most time-consuming task of the whole ME. The description of the task looks simple and clear: ‘open a port and forward all traffic to the server and inject any replies it gets back’. However, since the requirement of the ME is to run both on Windows and Linux, a lot of different problems arose that took a lot of time to figure out and implement correctly for both platforms. First, we will explain the port forward task in detail and what it looks like on a high-level view, before discussing the implementation details and problems we encountered.

High-level overview

Port forwarding is not only something that is implemented on the ME, but it also has a dedicated Port Forwarding Server. For this section, we can assume that it already exists since implementation details are not relevant. The full details about the Port Forwarding Server can be found in chapter 5.

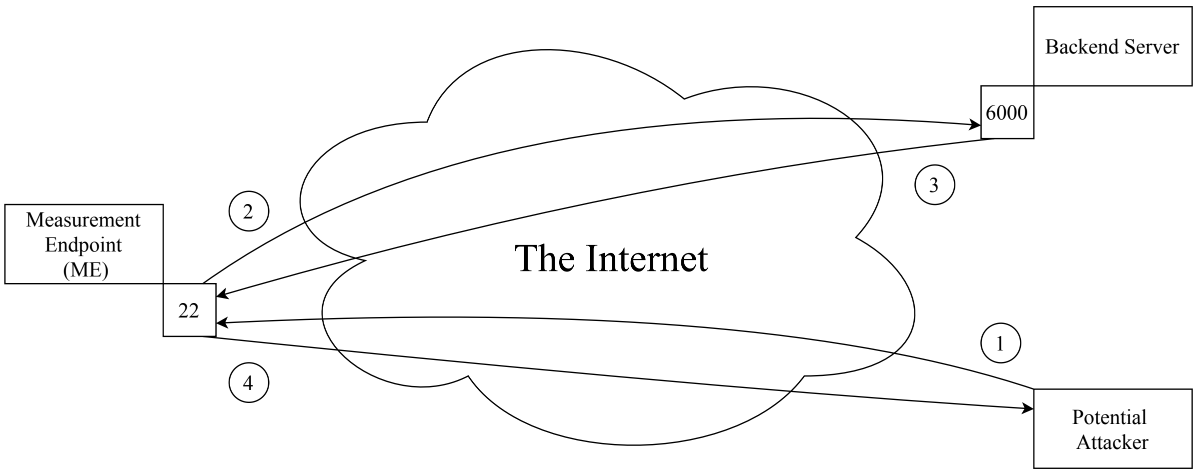


Figure 4.1: Visualisation of port forwarding

The Port Forwarding Server (PFS) serves the MEs doing the port forwarding task by saving the packets that they send and sending back any replies that the server wants the ME to inject. The idea of port forwarding is visualised in figure 4.1 and explained below:

1. An attacker wants to contact a computer on a specific port, in this example port 22
2. This computer runs an ME without the attacker knowing that. The ME forwards any data from the attacker to the server on port 6000 in this case
3. The server determines what to reply to the attacker and sends it to the ME
4. The ME injects the server's reply and the attacker receives it

The attacker in this situation thinks it is communicating with the computer, but in fact, it is secretly communicating with the server.

[7] `https://github.com/jacksontj/traceroute`

There were several requirements from the client which shaped the implementation details of the situation described above. Communication between the ME and the server had to be encrypted since the data would otherwise be sent in plaintext. The data can contain confidential information that we do not want to expose to any eavesdroppers, which encryption can prevent. Furthermore, the server and ME need to know that the data that they send between each other is authentic, so no one could have tampered with the data. The implementation of this is done with a custom protocol, called the *Port Forward Protocol*. The details of this protocol are not important for now, but further details about it can be found in chapter 8.

Implementation
After discussing with the team and client, the initial approach for implementing the port forwarding task was to use a raw socket. This raw socket can capture all incoming network traffic and send any traffic back. This approach works great for Linux, but after further inspection, it did not work for Windows. After a lot of debugging and research online, we found out that Go does not have an implementation for raw sockets on Windows, but this is not reflected properly in their documentation. Discovering that Go does not support this properly on Windows took several days due to the poor documentation on this specific part. Only after looking at the actual source code that is used on Windows, we found out that it is not supported at all. To look for any other approaches for Windows with raw sockets, we experimented with *Winsockets*[8]. *Winsockets* are the raw sockets counterpart for Windows. However, raw sockets on windows are very limited in what one can do with them[9]. To summarize, any recent Windows version does not support raw sockets fully and is therefore so limited in its use, that it cannot achieve the full requirements for the client. This was also confirmed after finding various posts and problems that other people had with Winsockets and their limitations.

Because the application has to be cross-platform, we had to take a completely new approach to packet capture and injection. The only other option we found was using PCAP. PCAP is an API for capturing network traffic and has libraries for both Linux and Windows, called *libpcap*[10] and *npcap*[11] respectively. The libraries are well maintained and are used by other large applications like *tcpdump*[12] and *Wireshark*[13]. The PCAP libraries achieved everything we wanted, which is: packet capture with filtering on port and protocol, packet injection and the fact that it should be cross-platform.

This is also how the final version of port forwarding is achieved. The ME starts a PCAP handle, which filters network traffic based on a given protocol and port from the assignment. Packets can be injected into the handle and are sent out by the network device, given that they have a valid Ethernet frame and IP packet. Since PCAP bypasses the kernel handling of packets, and thus the creation of proper Ethernet frames, the ME needs to craft them itself. The server makes sure it sends valid IP packets to the ME, so the ME does not have to worry about anything related to the network layer and above.

PCAP ended up being a perfect solution for the ME and especially in combination with Go. This is because Google has a library called *gopacket*[14], which provides packet decoding capabilities for Go and is fully integrated with the PCAP libraries for Linux and Windows.

For the last few implementation details for the ME, we assume that the PFS is fully implemented and the Forward Protocol is used for communication. The communication with the PFS is handled by a TCP client, which is run by the ME. The port forwarder component of the ME gives any packets it captures to the TCP client and retrieves replies from it that it directly injects into the network. The implementation details are thus hidden for the port forwarder.

The TCP client's workings are fairly simple. It contacts the PFS and establishes the connection ac-

[8]https://docs.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2
[9]https://docs.microsoft.com/en-us/windows/win32/winsock/tcp-ip-raw-sockets-2
[10]https://github.com/the-tcpdump-group/libpcap
[11]https://nmap.org/npcap/
[12]https://www.tcpdump.org/
[13]https://www.wireshark.org/
[14]https://github.com/google/gopacket

cording to the Forwarding Protocol. It retrieves packets from the port forwarder and sends them over to the server and handing back any replies that it gets. An error in the protocol aborts the connection to ensure that no one can tamper with the communication and to achieve the security goals.

The last big issue with port forwarding was discovered during system testing. During the system tests, the ME injected arbitrary UDP and TCP packets to a simulated remote attacker. However, the remote attacker never received any replies and aborted the connection immediately. Using Wireshark, we discovered that the kernel replies with ICMP or TCP RST messages to the attacker. This happens when the kernel cannot find any program that has a port open, which is true in this case. The ME has no actual port open but spoofs it by capturing and injecting packets. To suppress the kernel from interfering and closing connections, we discussed several options with the client. The first one consists of the ME opening a port, but not doing anything with it. This worked perfectly for UDP, but TCP is connection-based. This means that the kernel actively maintains the connection state by sending messages and hiding the underlying details from any programs. This can not be circumvented and we had to use another option. For now, the only way to suppress the kernel's responses properly is to block any ICMP and TCP RST replies in the *firewall*. Linux handles this perfectly, since *iptables* (the Linux firewall where one can add custom rules) has a lot of flexibility and options for blocking specific traffic. Windows, however, does not have a *firewall* with similar features. The *firewall* for Windows can only block specific protocols, but not protocol-specific information like an RST message for TCP, which makes it unsuitable for our use cases.

In the end, both our client and us could not come up with any other solution to solve the problems above for Windows, and we decided to drop packet injection support for Windows. This means that only Linux will support the ME injecting packets and that Windows can only forward packets to the server for analytic purposes.

4.2.5. PCAP Capture

PCAP capture is rather similar to port forwarding in the way it works. While port forwarding forwards every packet directly to the PFS, PCAP capture writes every packet received to a PCAP file. Then after a certain time duration or number of packets read, it sends the file to the API backend. So all that needs to be done for PCAP capture is writing packets to files. Luckily, *PCAPGo*[15] exists, which is a great library built for the sole purpose of writing PCAP files. Now that PCAP files could be made, we stumbled upon an issue. This issue arose from the fact that the files also needed to be uploaded to the backend. To do this efficiently, the files had to be compressed before sending them to the backend, to keep the load on the backend as low as possible. *GZIP*[16] was used for the compression, which is a Go library that allows for zipping files. After implementing the compression, we noticed that the compression was less effective than expected.

After the compression, an HTTP post request is created. This post request has the `multipart/form-data` encryption attribute which enables the attaching of files. The compressed file is copied to the request and then send to the backend.

4.2.6. UPnP port opening

Universal Plug and Play (UPnP) can be used for many different things. For this application, however, it will only be used to open a certain port on the local router. In Go there is only one library which offers the UPnP service for opening a port, namely *Go-UPnP*[17]. Therefore, we had to choose this library since implementing it ourselves would take up too much time. The library, however, also has a lot of downsides, which will be discussed more in section 4.6.5. The downsides are especially concerning customisation. But for now, we will have to make do with what we have. This means that we can open a port for both UDP and TCP and then close it after a certain duration.

[15]`https://github.com/google/gopacket/tree/master/pcapgo`
[16]`https://golang.org/pkg/compress/gzip/`
[17]`https://gitlab.com/NebulousLabs/go-upnp`

4.2.7. Banner Grabbing

Contrary to the previous ME tasks, banner grabbing was very simple to implement. This is due to the nature of banner grabbing itself. Banner grabbing essentially is when a device connects to a specific port on a domain, captures the data for a short duration and then returns the information. This can be easily implemented using the NET[18] library and making a dial to the address and closing it after a certain duration. For further improvement, it can be looked into whether it is possible to also use another method other than just dialling the address since these calls often result in access denied results.

4.2.8. Certificate Grabbing

Just like Banner Grabbing, the Certificate Grabbing was simple to implement and did not have any problems. Using the Go standard library, the ME requests the domain that the assignment specifies and checks if it uses Transport Layer Security (TLS)[19]. If it does, it returns the certificate chain in hex format, otherwise an error will be returned. An improvement could be to also support other protocols that make use of TLS. This could mean that the ME can also grab the certificates of, for example, email servers that use TLS.

4.3. Logger

During each *heartbeat*, the key information needs to be logged into files. To keep it structured, we created three files: *errors*, *requests* and *tasks*. The `errors` file contains all the errors which the ME might come across, such as a server being down or a library throwing an error. The `requests` file captures every request to the backend and the corresponding response. The `tasks` file contains every task received and also the corresponding result, such that can easily be seen what a single ME is running. These logs aid in debugging the different tasks and resolving the problem if there occurs one.

4.4. Request handler

To be able to fetch tasks, post results and register nodes, a series of HTTP requests must be made. Since most requests encapsulate the same query parameters, headers and body, the decision was made to extract all this behaviour into a separate package. This package has methods to do the following:

- Register a node and get the initial API key
- Refresh API key
- Fetch a task
- Post the result for a task
- Post compressed file results

This package also makes use of the logger to save all the traffic between the ME and the backend.

4.5. Result handler

Another small component of the ME is the result handler. It is an object that defines a format for returning results to the backend. It makes the handling of results on the backend a lot easier. This is because the object directly shows whether a task has failed, with the corresponding error or data if it succeeded. The object also contains a timestamp such that the backend can calculate how long it took to complete the task. For consistency, both the backend and the ME use this format as the body for HTTP requests and responses.

4.6. Future work

Although all features requested by the client have been implemented, there is always room for improvement. In the current state of the program, improvement can be made on DNS, the Logger, Traceroute, PCAP file uploading, UPnP Port Opening and Packet Injection on Windows. All of these improvements will be discussed in this section.

[18] `https://golang.org/pkg/net/`
[19] `https://tools.ietf.org/html/rfc5246`

4.6.1. DNS

At the moment we are using the library from *Miek Gieben*[20] for DNS, however, this library has some limitations. First of all, it only supports IPv4, at the moment this is not too big of a problem but as the internet switches to IPv6, this library will be of less use. Secondly, the library is not as fast as it can be. The normal net library is much faster but doesn't offer support for all DNS lookups. Hence, we suggest implementing the DNS records that are not included in the net library and put those in a custom library. In addition, this will also improve the code quality. *Gieben's* library requires a lot of extra parsing, which cannot be done in a good manner, due to the nature of the object returned by that library. It is a very complex object of which only a few fields are needed. Writing a custom library allows getting rid of these unnecessary fields.

4.6.2. Logger

When looking at log files, a user would like to have a small structured file to easily see what he or she was looking for. However, with the current way logs are implemented it just appends the new logs to the old ones. This requires the user to scroll all the way down, which is, of course, less user-friendly than we would like it to be. Also, if the ME runs for more than a couple of days, the log file size will become huge. To resolve this, log rotation needs to be added. This will remove all the data older than a specific time duration. Resulting in much smaller and cleaner log files.

4.6.3. Traceroute

One feature the client originally wanted but has not yet been implemented, was TCP traceroute on Windows. For future work, some more research can be done on whether it is possible to add TCP for Windows. However, for this project it was too much work to add this feature, especially considering our client gave this feature a lower priority.

4.6.4. PCAP file uploading

Two main improvements can be made with regards to the file uploading. Firstly, right now a specific number of packets is needed before the file is forwarded to the backend. However, the packets captured may vary a lot in size. This could cause the files, which are sent to the backend, also to vary in size and therefore give an inconsistent load to the backend. So, what would be better to do, is wait until the file reaches a certain size and then submit it to the backend. In this way, the load on the server from a single ME varies less. The other point of improvement would be to experiment with the different types of compression. Right now, the program uses the default *GZIP* compression method, but in the *GZIP* package, but other compression methods can be used. These methods can result in even smaller files but could increase the compression time. To make the correct decision here, some research needs to be done on which method would yield the best results for the ME.

4.6.5. UPnP port opening

The library we are currently using for UPnP has poor documentation and the code does not follow many of the basic software methodologies. For example, having less than 4 parameters and return types, no comments, etc. Nonetheless, those aren't the biggest issues with this library. The biggest issue comes from that it can only open the port for both UDP and TCP, but not for just one of them. So, if in the future a library would be introduced with proper documentation and more customisation, we would recommend switching.

4.6.6. Packet injection on Windows

The only way, for now, to implement packet injection for Windows, is to write a custom network driver. This was obviously out of scope for this BEP and would be time-consuming, especially given that none of the team members have experience in this field. Furthermore, the client indicated that most ME instances will run on Linux, so luckily this is not too big of a deal after all. However, it would make the functionality of the ME more similar for both platforms.

[20]`https://github.com/miekg/dns`

4.7. What we could/should have done differently

Looking back on the development of the ME, some things could have been done in a better way.

First of all, we would probably switch to a different programming language for most of the ME. Go is only required for mostly the port forwarding part, but for the rest, it brings a lot of downsides. The fact that a programmer has to catch every error directly, rather than throwing them and catching them later, gives a lot of redundant code. In addition, Go has no method overloading and the syntax of the language itself were downsides for us. The syntax of Go is different from most other languages as it requires variable names to be in front of the type, exporting functions depends on capital letters, etc. Furthermore, Go is a relatively new language, hence few libraries exist for it.

Another downside of Go is the scope of variables. Go only supports method variables, package-wide variables and exported variables. This means that a programmer cannot define a variable only to be used in a struct or file, but it is accessible in the whole package. Especially for writing tests, where variables can have similar names if tested in the same package, this resulted in several headaches and workarounds that do not make the test code cleaner. Furthermore, the only way to export a variable or method is to start the name with an uppercase. All lowercase methods, variables, structs, etc are not exported. We thought this was a poor design of exporting variables and does not improve code readability, especially when just starting with the language. Switching to another language, like Python, allows us to improve the code quality, but it will not necessarily improve the speed of the program.

Secondly, we should have written end-to-end tests which include the backend, database and ME together. All of the end-to-end testing was now done by hand, mostly because there was not enough time to implement these kinds of tests properly. During the last week of the project, we had to test some of the features multiple times to make sure that they work correctly, that any bugs identified were fixed, and to see if we could find new bugs. Although completing a full end-to-end test with all the components did not take a lot of time, automating it is better and saves time that can be spent elsewhere. Furthermore, it is more flexible and allows writing regression tests to prevent identified bugs from returning. These regression tests were now written only for the program that contained the bug, but not as an end-to-end test.

4.8. Conclusion

We have now discussed all the features of the ME. In addition, we have discussed what still needs to be done and what we could/should have done differently.

We are very content with the final result of the measurement endpoint. In the end, we have implemented all the features requested by the client and learned a lot from the project. We have learned a new language, gained a lot of knowledge about existing network protocols and cross-platform development. The knowledge gained with this project will be very useful for future projects.

4.9. SIG Feedback

The initial scores from SIG for the measurement endpoint were not as high as we expected. The overall score for the measurement endpoint was a 4.1. This meant we had to resolve some issues to improve the quality. In the coming sections, we will go into detail about how each violation was resolved or why the decision was made to not resolve it. Before we continue to those sections, we would like to make the disclaimer that due to the design of Go it is impossible to achieve a perfect score of 5.5. Go's conventions encourage programmers to explicitly check for errors wherever they might occur. This results in many checks for errors and thus, increasing the unit size and complexity.

4.9.1. What we did resolve

In this section, we will go over the feedback from SIG which we resolved.

Unit interfacing

The first violation was that we had a couple of methods with too many parameters. All of these violations were for `createResult` methods. In these methods, we would have all the parsed parameters and then use those to call the library required. To reduce the number of parameters we created a Go `struct` for the parameters for each task. This reduced it from four to six parameters to just two, namely the parameter struct and an error.

Unit size
The final violation we were able to resolve was a unit size problem. Although we did not resolve it for every method, we managed to remove many of the cases. This was done by extracting repetitive parts to separate functions, move the contents of a for loop to a separate method, and so on. For some methods, however, the decision was made not to resolve the unit size. The reason for that will be discussed in the next section.

4.9.2. What we did not resolve

In this section, we will discuss what we did not resolve from the feedback.

Duplication
During the time of the SIG feedback, we had only one violation for duplicate code. This was due to WHOIS and DNS requiring the same parameter. We refactored much of the code, so now less code is required for parameter parsing. However, it still might be the case that there is duplication between different tasks. This is for the simple reason that many of the tasks still share the same parameters, for example of port or domain. It would be less clear if we would refactor it since right now one can directly see which parameters are required for each task.

Unit size
As mentioned in the section 4.9.1, we did not resolve all the unit size violations. Some of the methods just cannot be refactored further, such as the `execute` method in the `TaskExecutor`. This method has a switch statement with a case for every task that is implemented in the ME, for this reason, this method already violates the unit size restriction. Another example of where we decided not to reduce the unit size was in the instantiation of objects. Instantiating complex objects could easily result in many lines of extra code. Take for example `ReadNewConnectionMessage` method in `forwardprotocol` package, which constructs a connection struct. Since a connection struct has five fields, refactoring this method would require five parameters, which would violate the Unit interfacing rule. Therefore, we decided also for these methods to not refactor them, because it would simply raise another violation or make the code less clear.

4.9.3. Conclusion

In short, the feedback we received from SIG only had a small number of violations. This meant we only had to refactor a small part of our code. This refactor was done quickly and efficiently. For the violations we did not resolve, we argued why we did not do this.

5 Port Forward Server

This chapter discusses the Port Forward Server (PFS), which is an important part of the Port Forward assignment, as described in section 4.2.4. The PFS is written in Go and handles ME connections. The MEs forward their captured packets which the PFS stores. Furthermore, the PFS determines what packets to reply back to the MEs with the help of two independent components, called the Glue and Docker containers. They are explained in the next section.

To better understand what these independent components are, how the PFS works and how it is implemented, we first give an overview of the different components and their explanations. After that, we explain how the connection handling is done. Next, the communication with the Glue component is explained, followed by our demo implementation for a Telnet server. Lastly, any future work and improvements for the PFS are discussed. Please note that the PFS was implemented after the first SIG code upload and does not have any feedback for that reason.

5.1. Components

The PFS can be broken up into a number of components that each have their own task, the components are given below with a short explanation:

- **TCP Server**, accepts any incoming connections and enforces the Port Forward Protocol for each connection.
- **Port Forward Protocol**, the implementation of the protocol that is explained in chapter 8.
- **Connection Map UDP Socket**, determines the IP of the 'attacker' that communicates with the ME and sends this IP and the one of the ME to the backend. This information is used to draw arcs for each connection on a big world map for visualisation purposes and is explained in more detail in chapter 7.
- **Client State**, keeps track of the current state of each ME connection and makes sure traffic is encrypted/decrypted and forwarded to the right Glue instances.

Besides the PFS components from above, there are three more components that are separate from the PFS but extremely important to its workings:

- **Node database**, the database in the backend that stores all information related to nodes and assignments. See chapter 6 for more details.
- **Docker container**, this is a program running on the same computer that can analyze the actual packet contents from each ME and determine what to reply.
- **Glue**, a small server that is unique for each different Docker container. The Glue is literally the glue to communicate between the PFS and Docker container. The task of the Glue is to convert a message from the PFS in a message that the Docker container can understand, which can be different for each container. It also needs to send any replies from the Docker container back to the PFS.

5.2. Connection handling

The TCP Server of the PFS listens on a specific port to any incoming connections and starts a Go routine, a lightweight thread of execution, per connection. The choice for a TCP is because of the Port Forward Protocol and is located in chapter 8. The Client State takes over here and initializes and enforces the Port Forward Protocol for the rest of the communication. The Client State contacts the Node Database to retrieve the API Key of a ME, which is used to encrypt and decrypt the traffic and make sure the ME is actually whom he claims to be.

Furthermore, the Client State also keeps track of which Glue to forward the packets to and receive replies from. Replies are sent back to the ME, where it injects the packets into the network. Any packets that are received from the ME and Glue are temporarily stored in a pcap file and are sent for permanent storage to the backend at intervals to not overload the servers.

5.3. Glue Communication

The communication with a Glue server is done with a message structure that we created, which can be seen in figure 5.1.

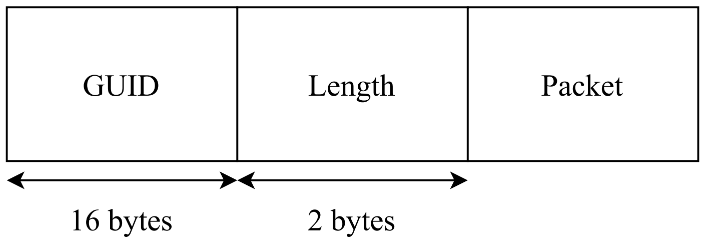


Figure 5.1: Structure of a Glue Message

Each Glue Message contains a GUID to identify the ME with the packet that it forwarded. After that, it contains the Length of the Packet field, which is a maximum of $2^{16} bytes$, hence the field only needs to be 2 bytes to cover the whole range. At the end of the message is the actual IP packet.

The Glue now needs to keep track of the replies from the Docker container and link it back to the GUID before sending it back to the PFS. This implementation can differ per Docker container and Glue. Furthermore, there is no encryption for the message since the docker instances and Glue are running on the same server.

5.4. Demo

For this project, the goal was to implement the PFS and have a demo ready to show its workings. This means that any actual Docker containers and Glue implementations are out of scope. For the demo, who chose a Telnet[1] Server as the Docker container and implemented a corresponding Glue with it. This Telnet demo was requested by our client to show that our implementation for Port Forward is working.

In this demo setting, the attacker is a local Telnet client running in the terminal. On the same computer, there is an ME running which captures and forwards any traffic destined to TCP port 23. Note that there is no actual port opened locally, instead, it is spoofed by the Port Forward assignment.

The PFS handles and initializes the communication and forwards all captured packets to our Glue implementation. This Glue server is fairly simple and locally injects the packet that it received and listens for any replies that are linked back to the GUID and send to the PFS. The PFS sends it back to the ME which injects it on the local network, where the Telnet client receives it. There needs to be a Telnet Server running on port 23 on the same machine where the PFS and Glue are located to make this setup work. The client is satisfied with the results of this demo, which shows that the Port Forward

[1]`https://tools.ietf.org/html/rfc854`

assignment works correctly and that the port can be successfully spoofed on any machine running an ME on Linux.

5.5. Future work and Improvements

In the current version of the PFS, each PFS can only communicate with one Glue server. This has the consequence that if there need to be multiple Glues running at the same time, they each need to have their own PFS. This is not necessarily bad on its own, however, we did not think about one consequence that this has. There needs to be a port opened in the firewall for a PFS to receive traffic from the outside. If there are multiple servers running, there need to be multiple ports opened in the firewall as well. This is not ideal and increases security risks since we want as little ports as possible opened in the firewall.

To combat this problem, there should be a special server that handles all Port Forwarding traffic and thus only one exposed port in the firewall. This server determines for each ME the right PFS and forwards the traffic to it. This means that there can be multiple PFSes running locally and that there is only one main server that handles all the incoming and outgoing traffic. Due to time limitations, we were not able to implement this server after the client requested it. Since it was not listed as a requirement in the original MoSCoW list (see appendix A), we were not able to implement this from the start.

Furthermore, the PFS currently only supports Glue instances that are running on the same machine. It could be possible that the client wants to offload docker instances or Glues to different servers in the future, but this was not the plan at the time of the project. This change should be easy to do but might require encrypting the Glue Messages to not expose any packet information to potential eavesdroppers. Implementing encryption for the PFS and Glue communication can also be considered future work, but was not required or needed at the time of the project.

5.6. Conclusion

This chapter discussed all the features of the Port Forward Server and how they are implemented. Next to this, we discussed any future work that can be done to improve or extend the PFS that did not have to be considered at the time of the project due to the requirements set by the client.

6
API backend

The API backend is the central point of our system. The backend is written in C-Sharp and makes use of the .NET Core framework. The application serves multiple purposes. Firstly, it serves a central point of communication for the nodes (measurement endpoints). Secondly, it allows for the administration of the measurement endpoints. And lastly, it serves as a central data collection point, where all data is gathered and stored. A simple overview on the API calls we implemented can be found appendix B and the full details can be found here[1].

To better understand the implementations and decision making, we will start by shortly explaining what all components are, how they interact and what some of the requirements are. After that, we will explain every implementation and the decisions making behind them. We will also discuss the work that remains to be done after the project is finished and what we would have done differently if could start again.

6.1. Components

The backend can be most easily explained and understood if we break it up into several components:

- **Node interaction**, the interaction between the nodes and the central server. For example, automatic generation of nodes and registration.
- **Assignment handout**, the initialization and continuation of communication between the nodes and the server. For example, gathering assignments.
- **Result storing**, the storing of the results from assignments executed by nodes in the network.
- **Administration**, the administration of nodes. For example, adding assignments or blocking IPs.
- **Node configuration**, the configuration the node owner can do for his nodes. For example, block certain assignment from being run.

6.2. Node interaction

The first step in creating the application was node interaction. This meant we first had to create a way for a user to generate a node. Generating a node returns an ID, password and API key.

The API key is used for the signing the results as being authentic and as an extra layer of security. Although our client did not request this, we made sure that an API key expires after one week and must be refresh by requesting a refresh. We did this to improve security.

To generate a node, some data from the node is required. Initially, we required the region, operating system, device and IP address. However, later in the project, we realised this did not make much sense, as the region and IP are dependant on each other and can change with each request. We thus made it, such that the region and IP are dynamically determined for each request, and not stored in

[1] `https://app.swaggerhub.com/apis/alexj123/internetmonitor-backend_api/1.0.0`

the database. Also, the device of a node did not make sense, as it did not serve any purpose, and we later changed this to be the architecture. In the end, only the operating system and architecture are required to generate a new node.

6.3. Assignment handout

After creating a way to generate a node, we could start building the core of the application. This core consists of handing out assignments to nodes. In other words, it allows nodes to send a heartbeat and retrieve an assignment. The client had several requirements for handing out assignments:

- A node can only do an assignment once.
- Restrictions can be set on assignments, to restrict which nodes can/cannot do the assignment.
- A node can block certain assignments.
- Assignments should be randomly handed out.

These requirements together made it quite the task to do this efficiently and consistently. After writing the first version we ran into performance issues while load testing. After looking more in-depth into our code, we optimised the performance significantly.

Optimising was not a trivial task, and together with the client, we looked into many possible solutions. Our client mentioned caching of assignments as a solution on multiple occasions as that would have the most impact on performance. However, when looking more in-depth into caching, we realised that with the requirements set for handing out assignments, it would be (near) impossible to cache assignments, especially considering the time-frame of the project. The main reason for this was that our client required the assignment data to be consistent at all times. Together with the asynchronous nature of ASP.NET requests, it would be very difficult to implement this caching. In the end, we optimized by writing a much more efficient query, a check for an already assigned assignment beforehand, handing out new assignments after posting the result of an assignment, and minimizing the round trips to the database. This way we achieved the desired efficiency from the client without the caching.

During the project, our client requested an additional feature: every assignment should have a status. This meant extra load when requesting assignments. Updating the assignment had to happen every time an assignment was handed out. Luckily, when implementing we managed to keep the extra load minimal, and almost no difference was noticeable.

6.4. Result storing

Another key component of the application is the storing of results. The client required to store results from nodes in a database. For this, the client recommended using Hadoop[2], as it made for very large datasets. And, the datasets here would be large. However, as we had no experience with big data beforehand, setting up Hadoop was not a trivial task. In addition to this problem, we discovered that hdfs, the storage system from Hadoop, always creates 128Mb blocks on your system regardless of how much data you're storing. To resolve this, we had to create some way to cache the results before writing to Hadoop. We went through quite a bit of brainstorming, after which, together with the client, we came to the conclusion to use Kafka[3]. Sadly, writing to Kafka also came with issues. Opening the connection and writing to Kafka is not a lightweight task. To resolve this, we had to create some in-memory caching before we would write to Kafka.

After creating the in-memory cache and setting it up in such a way that the application would write to Kafka after a certain threshold of results was met, we had to create a way to get the data from Kafka to hdfs. Due to time constraints and other priorities, we decided that we would not set up this part. We did give a small attempt to get it set up, but it would require us to dive deep into Kafka streaming and big data. Luckily, because Kafka is set up and running, the results are still stored in a big data capable system. The client could later set up the streaming from Kafka to hdfs.
Using Kafka almost all results could be stored, but not all of them. The results from PCAPCapture

[2]https://hadoop.apache.org/
[3]https://kafka.apache.org/

and PortForward assignments are in a different format. It consists of .pcap files. To resolve this, we added a way to upload files to the backend. These are stored in a separate folder for each assignment. To do this efficiently we made sure that compressed pcap files will be uploaded from a measurement endpoint. Here we also ran into some performance issues, but these were resolved issues in a similar way as with the heartbeat. Also, as mentioned before, we had to take into account the assignment status. This meant that we also needed to check for the assignment's status when receiving a result.

The client also requested a real-time view of the incoming traffic to the nodes. Incoming traffic would be captured when doing a PortForward assignment. In addition to this, we created a real-time view of the results being sent to the backend. This was not a requested feature, however, we felt that this would really add to the application. And because we already had created the in-memory cache for Kafka, it was easy to add this feature. We did this by creating a WebSocket which will send out data for the visualisation every time a result was posted. We thus have two types of real-time traffic, the first being during the PortForward assignment and the second being when a result is posted. The data is all sent out to the visualisation in one single format and consists of the following: *Source Location, Destination Location, and Assignment Type*.

For the source and destination location during a PortForward assignment we created a background service in the application. This service will open a socket and accept data from the PortFoward. It then simply receives one source and one destination IP from the server. To translate the IP to a location, we used an IP to location database (given to us by the client). The application simply queries this database with an IP and we simplify the results. The simplified result consists of the location (either a region or a city), the latitude and the longitude.

The determining of the destination and source of results (so not a PortFoward assignment) was a bit different. Setting up the source location was a simple task. As every node has an IP, retrieving the locational data is very simple.

However, for the destination location, it was not that simple. This is because an assignment does not have an IP. To resolve this, we made it in such a way that our application looks if there is a parameter with the name "domain". It then uses the value of the domain, if it is not an IP already, to do a local DNS lookup. With the resulting IP of the DNS lookup the locational data can easily be retrieved in the same way as it is done for the source location.

Determining the assignment type for both types of live traffic was a simple and easy task.

6.5. Administration

The client required us to create admin operations within the backend. These operations had to consist of creating, deleting, retrieving assignments, and creating a blacklist to block IP ranges of nodes. During the project, the client added several requirements to this list. Firstly, an admin should be able to see which nodes are assigned to an assignment. Secondly, an admin should be able to retrieve all uploaded files of an assignment.

We started by creating the admin privileges in the application. To do this, we created an API key, which must be sent in the `X-Api-Key` header. If correct, then the user can perform said action. We created the following operations for admins:

- Creating, deleting and retrieving assignments
- Retrieving all assignees for one assignment
- Retrieving all uploaded files of certain assignments
- Creating, deleting and retrieving blacklisted IPs

All of the aforementioned operations were quite straightforward to implement. No problems arose when doing this. We must note that the prior knowledge in API key authentication helped with that.

6.6. Node configuration

To easily see what a node is or has been doing we also created some way of configuring and retrieving information about a node. For this we implemented the following two API calls:

- **Configuring nodes**, allows the owner of a node to retrieve and update the configuration of the node. Only the name and the forbidden assignment types of a node are configurable.
- **Retrieving all assignment for a node**, allows the owner of a node to see all the assignments his/her node has ever been assigned to.

6.7. Additional features

We have now discussed the main features of the API backend. But, there are several smaller, but might even be just as important, features which do not fit in any of the previous chapters. These will be discussed here.

Firstly, when a user or a node sends a request all of the data is validated. This is to make sure no invalid data can be stored in the database, and done for every request except for the uploading of files. Validating files is a very large task, and does not fit within the scope of this project.

Secondly, a very minor, but still very useful, one. Every response from the API backend is wrapped in a result object. This gives the user an easier indication of whether or not the request was successful and if not, why not.

Finally, many of the settings in the application are configurable. We did this to allow the client to change the settings to his preferences, which might not necessarily align with ours.

6.8. Future work

Although we implemented all features eventually agreed on by the client, there are some points of improvement that remain open. Namely the following: implement long term result storing and analytics, improve input data validation, historical data, and the addition of users.

6.8.1. Result ingestion and analytics

As previously mentioned, the data that is collected from assignments are stored in Kafka streams. This data is never written to permanent storage however, therefore it is currently not possible to perform large scale analytics on the acquired data. Despite not implementing it, we did put a lot of time into designing and thinking about how this data would be stored and represented. Implementing the data ingestion from Kafka into HDFS should therefore not require modification of the backend code. What remains to be implemented is the actual ingestion into HDFS from Kafka, although we did do some research on what could be used for this, we, unfortunately, did not get any conclusive results. For the actual analysis of the data, we recommend the use of Spark[4], as this works very well with our representation of the data. A more detailed description of the process and what remains to be done can be seen in figure 6.1 below.

6.8.2. Input data validation

When writing an application which takes user input, you are going to get a lot of bad input. It is therefore very important that you validate all user input for either malicious or incorrect input to prevent crashes further down the system. Although a significant amount of time was spent validating input, a lot remains to be done on this front. One of the most important being the validation of the input arguments for assignments. Despite that, this is only done by system admins, and we can, therefore, assume some level of competence, people still make mistakes. Because this part of the system is not exposed to the world, this did not have the highest priority, it is however still something that might be desired in the future.

[4]`https://spark.apache.org/`

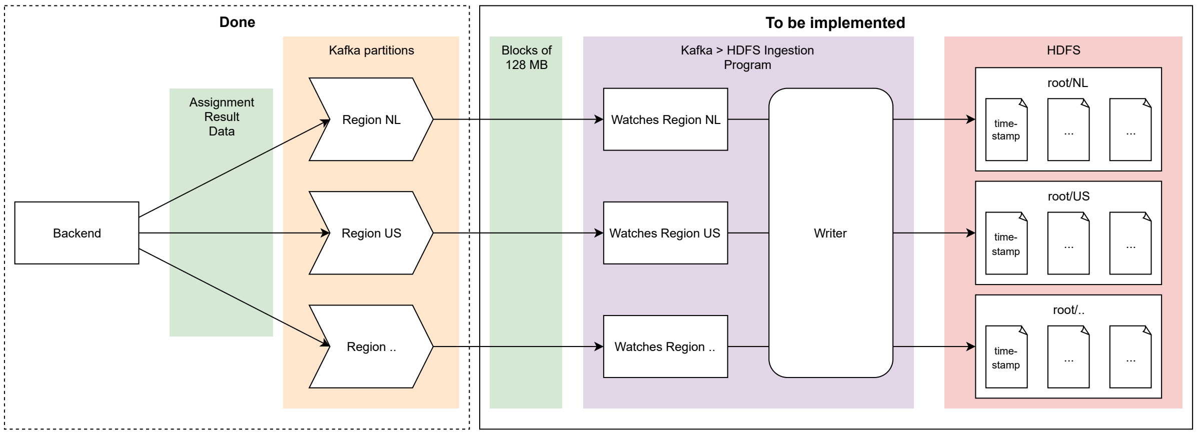


Figure 6.1: The result pipeline

6.8.3. Historical data

One feature the client wanted at the start, was the ability to compare the results of our application, with older datasets. This would allow a user to spot discrepancies in the timeline. However, there was simply not enough time for this feature, and together with the client, we decided to drop it. We think that even if no problems would have occurred during the project, it still would not have been possible to create this feature.

6.8.4. Users

At the beginning of the project, the client talked about the addition of users, allowing users to 'own' measurement endpoints, giving them an overview of all their endpoints and allowing for easy administration. Currently, a user has to log into each measurement endpoint individually, instead of having an overview of all of them. Unfortunately, we did not have enough time to complete this could have. We did, however, keep this in mind while developing, therefore allowing another developer to implement users without making a lot of changes to the existing code. We already created the design for the user login during the design phase3. Due to other priorities and time constraints, we did not think it was responsible to rush something as important as user data, thus leaving this for future implementation.

6.9. What we could/should have done differently

During the development of the backend, there were a couple of occasions where we realised it would've been better if we had chosen a different method.

Firstly, when thinking about what database to use, we thought about using MongoDB. However, our client recommended against this, and together we decided to use PostgreSQL. During the development, we realised that a lot of the data is much more easily stored and retrieved in a non-relational matter. Therefore, we think it would have been better to use a non-relational database, like MongoDB. After doing some research, we concluded that MongoDB would be suitable for this. MongoDB already has examples of working together with big data and claims to be very scalable. Although we are using Entity Framework, which makes it very easy to change from db, it would still require too much time for us. This was simply not possible with all other tasks we still had left.

Secondly, also related to databases, are the issues with Hadoop. The task of setting up Hadoop was thought to be a lot easier than it actually was. As we had no experience beforehand, this task took a lot longer than thought. We should have realised this earlier and changed our priorities earlier.

6.10. Conclusion

We have now discussed all the features of the API backend. In addition, we have discussed what still needs to be done and what we could/should have done differently.

We are very content with the result of the API backend. We have implemented all features eventually agreed on by the client and learned a lot from this project. We have improved on our knowledge working with .NET Core and big data. The knowledge gained with this project will be very useful for future projects.

6.11. SIG Feedback

The initial scores from SIG for the API backend were very good. The overall score for the API backend was a 4.5. This meant we had only a couple of issues needed to be resolved. We will now go into detail about each violation and if we resolved it, how we did it and if we chose not to resolve it, why not.

6.11.1. What we did resolve

In this section, we will discuss what we did resolve from the feedback.

Duplication

We had one violation with code duplication between two methods within one class. This was easily resolved by creating a single method which both methods could use.

Unit complexity

Another violation was that we had a method with a complexity of 13. However, this is because SIG does not recognize LINQ queries (which get translated to SQL queries). This means it's a false flag. In addition, because we needed to improve the performance of our database interactions already, this method was removed shortly after we uploaded to SIG, before we even looked at the feedback.

Unit size

The final violation we were able to resolve was a unit size problem. The `ConfigureServices` method, in which services are registered for dependency injection in the application. Luckily, this violation is easy to resolve by creating a separate method for services which can be grouped together. Thus, by doing this we created several methods which the `ConfigureServices` method calls.

6.11.2. What we did not resolve

In this section, we will discuss what we did not resolve from the feedback.

Unit interfacing

We have two violations with unit interfacing. More precisely, both issues are regarding too many parameters in the constructor. However, the problem here is that SIG does not realise we are using dependency injection. This means that we are not constructing any of the objects we need. Therefore, there is no violation with instantiating any of the classes.

In addition to the above reason, both of the classes with too many parameters cannot be changed in the first place. The first class, `ApiKeyAuthenticationHandler`, extends a class from `Microsoft.AspNetCore.Authentication` which already takes four parameters. We only added a single parameter. There is thus no way to change this.

The second class, `AssignmentController`, is the endpoint for all assignment interaction. Every single class used in this class is different from one another. And reducing the number of parameters will only lower the readability of the project. Also, we are injecting classes like loggers, these are classes we cannot just simply remove or change. There is thus also no way to change this.

Module coupling

Another violation we did not resolve, was a violation regarding module coupling. In our application, we use Entity Framework. One of the many advantages this has is that it allows for interaction with the database via code, there is no need to write any raw database queries. To simplify the interactions with the database we used the repository pattern[5]. Like mentioned in the article, we also created a generic interface for interacting with the database. This generic interface is thus used by every repository in

[5]`https://deviq.com/repository-pattern/`

the application, and since we have two repositories which use this generic interface, the module will have a high value for module coupling.

Illegal catches
The final violation we did not resolve, was a severe violation. In the `main` method, which starts the entire application, we have a try-catch block for any type of exception when we ”construct” the application. However, all this try-catch does, is log the error which occurred and then rethrow the error. Also, we used the example from our logging package, which uses this try-catch block, when creating the main method[6].

6.11.3. What we changed in advance

After receiving the feedback from SIG, we knew what made good maintainable code. We used this information when writing new code. For example, we had a method with 3 parameters already, and when we were expanding this method, we created several DTOs (Data Transfer Object) to wrap the parameters in to reduce the number of amount parameters.

6.11.4. Conclusion

In short, we received only a small amount of feedback from SIG as our code already had a high score for maintainability. This meant we only had to resolve a couple of issues. The issues we resolved were quick and easy to do, and all issues we did not resolve, we argued why we did not.

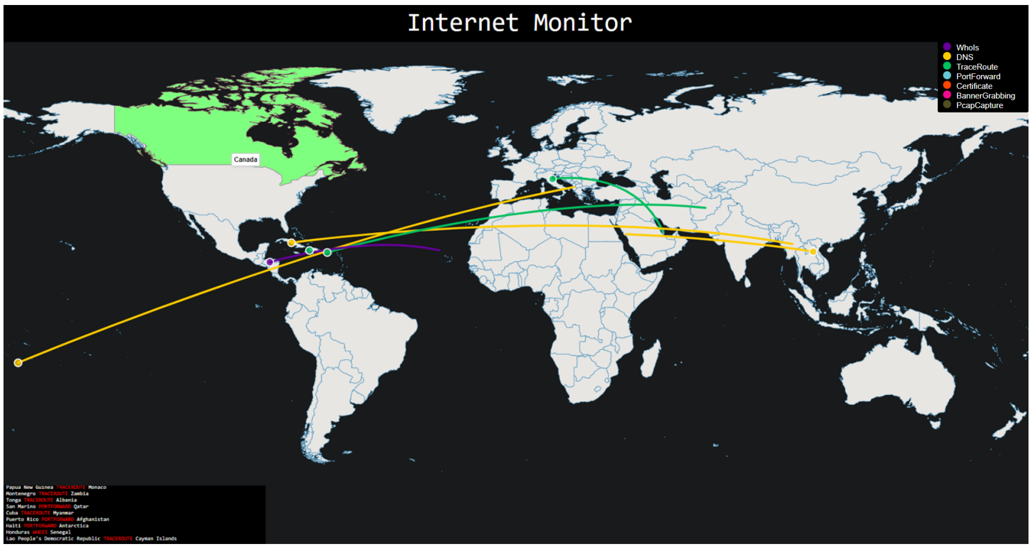
[6] `https://github.com/NLog/NLog/wiki/Getting-started-with-ASP.NET-Core-3`

7 Visualisation

In this chapter we will discuss the visualization. The visualisation is a combination of both *UI* and *UX* designs that are implemented using Java. These design are made to ensure that the user has a pleasant experience working with the program. It wraps the API calls using buttons, ensuring that the user does not have to do raw API calls. The specific requirements from the client can be found in appendix A. We developed visual aspect of our project by creating the **Map Screen** and the **Configuration**. We will discuss both of these in this chapter. We will go in greater detail on what design decisions were made. After discussing both elements of the visualisation in greater detail, **the REST Client** is discussed, which is an element that connects backend and the visualisation together. Finally, we will conclude this chapter by giving an overview of what we learned and how we might improve on it should we have more time, or if we would have to do it again.

7.1. Map Screen

The map screen forms one of the main pages that the user will visit. It is the page where everyone can look at the map to see all assignments that are being performed and the incoming traffic. It is made in such a way that everyone, with a basic understanding of the assignments that are being performed (know the definition), can understand the map and the meaning of all the arcs. This screen was quite easy to implement with the features described in the next sections. It also adheres to the client's main requirement for the visualisation, aesthetics. This means that this screen is implemented to be as beautiful as possible.



```
Papua New Guinea TRACEROUTE Monaco
Montenegro TRACEROUTE Zambia
Tonga TRACEROUTE Albania
San Marino PORTFORWARD Qatar
Cuba TRACEROUTE Myanmar
Puerto Rico PORTFORWARD Afghanistan
Haiti PORTFORWARD Antarctica
Honduras WHOIS Senegal
Lao People's Democratic Republic TRACEROUTE Cayman Islands
```

Figure 7.1: The map screen

7.1.1. DataMaps Library

To create the map and the arcs that you can see in figure 7.1, we used the DataMaps[1] library. This library can load in a map and has a method for drawing arcs and bubbles. Using this library enables us to quickly create this home page with the basic functionalities. Both the arcs and bubbles were used to ensure that the user can see the starting point of the arc and see the destination of the assignment shown. The arcs are drawn in real-time based on the data that we get from the backend through a WebSocket, which will be discussed in a later section.

7.1.2. Hub

The hub is a connection to the backend that will get the live results. Whenever a result is received the `showData(src, dst, type)` method is called, which will add a bubble and an arc to the map. At first we had the idea to only draw seven arcs at once, since it will be easier to understand what is happening, but after consulting with the client he told us that he wanted it to be flashy. Because of this we decided to remove the limitation on the amount of arcs drawn on the screen. By doing so, we lose some ease of use, but this is compensated by the visual presentation. Arcs will be drawn on the screen and only after 10 seconds they be removed, this is done to avoid overcrowding the map.

To get the correct locations of the countries we first used the ALPHA-3 Country Code with DataMaps as retrieved from the WebSocket. But this caused a bug where the arcs and bubbles were slightly misaligned. This is shown in figure 7.2. In order to fix this bug we, instead of sending ALPHA-3 Codes, ended up with sending latitude and longitude. This also allowed us to implement location based on not only countries but cities.

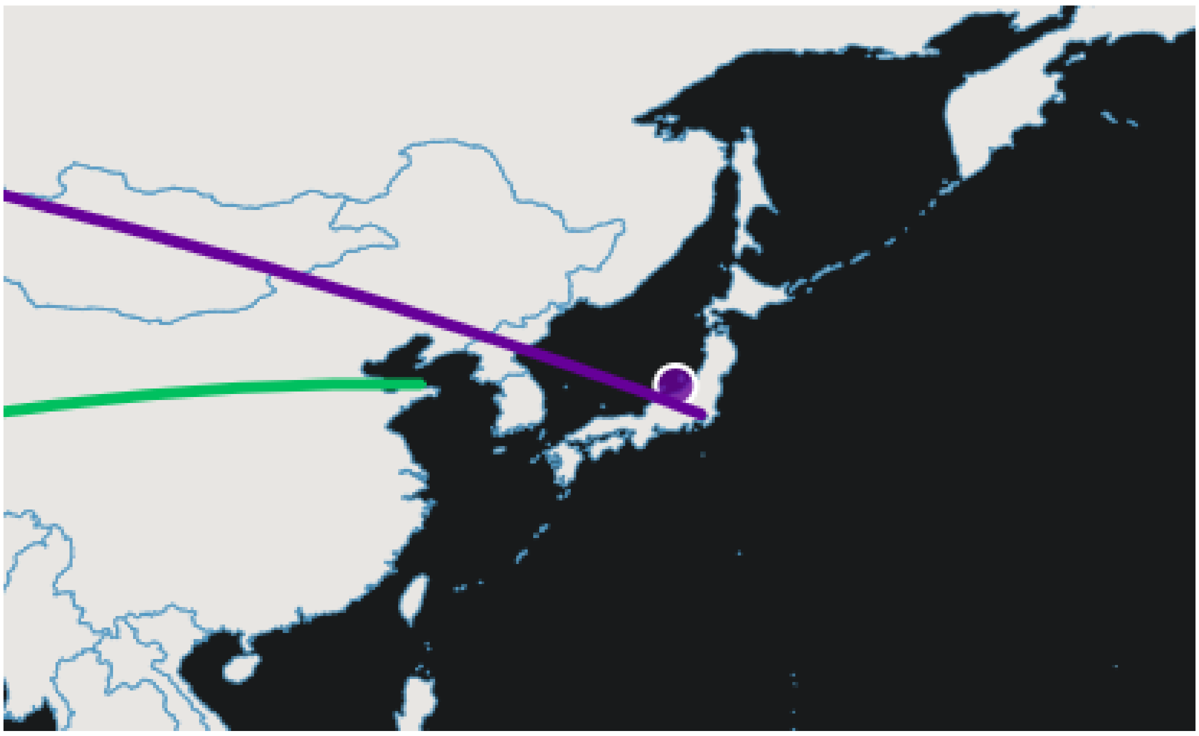


Figure 7.2: A bug when using ALPHA-3 codes.

[1]`http://datamaps.github.io/`

7.2. Configuration

The Configuration, consisting of a collection of screens, is an important part of application. It allows admins to add new assignments and it allows for users to view results and edit their nodes. And while this was not a direct requirement from the client, we did believe that this was an important feature to have. Since it allows for easier configuration, and works with the aesthetics that the client requested.

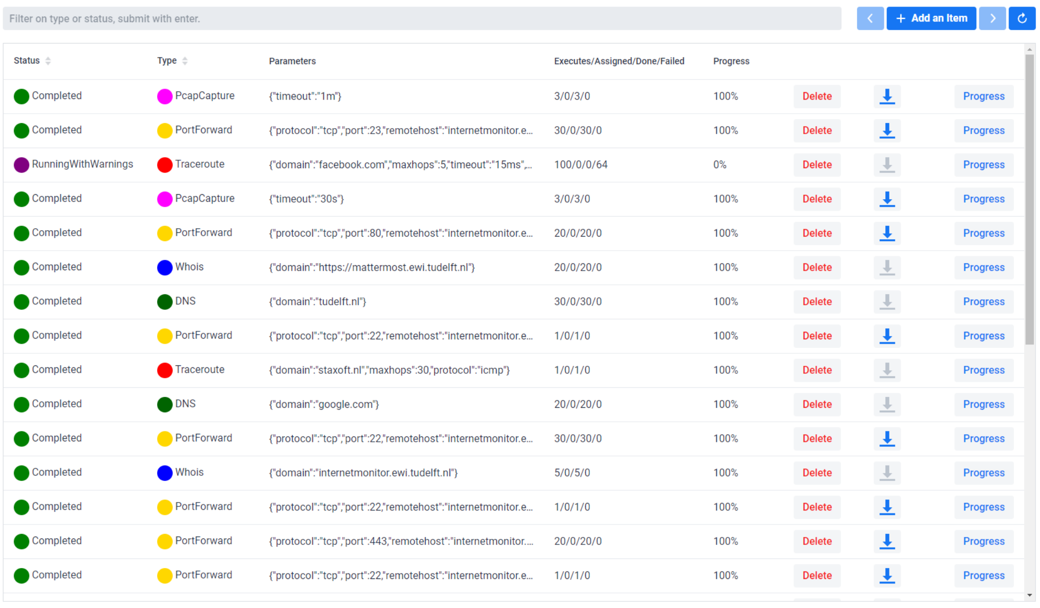
We will start by going into more detail on the *UI* and *UX* of the configuration screens. More information on how we show data on the screens can be found in section 7.2.2. Finally, in section 7.2.3 the REST client will be discussed, this is the connection between the Visualisation and the backend.

7.2.1. Visuals

In this section the two main components will be discussed, the **grid** and the **dialogs**. These components make up a huge portion of our application.

Grid

A grid is a component that can be used as a list. It allows us to show all of the data in a clean and easily readable format. A grid is also used to keep track of assignments and ensure that whenever a user clicks on the item the correct behaviour occurs.



Status	Type	Parameters	Executes/Assigned/Done/Failed	Progress			
● Completed	● PcapCapture	{"timeout":"1m"}	3/0/3/0	100%	Delete	⬇	Progress
● Completed	● PortForward	{"protocol":"tcp","port":23,"remotehost":"internetmonitor.e...	30/0/30/0	100%	Delete	⬇	Progress
● RunningWithWarnings	● Traceroute	{"domain":"facebook.com","maxhops":5,"timeout":"15ms",...	100/0/0/64	0%	Delete	⇩	Progress
● Completed	● PcapCapture	{"timeout":"30s"}	3/0/3/0	100%	Delete	⬇	Progress
● Completed	● PortForward	{"protocol":"tcp","port":80,"remotehost":"internetmonitor.e...	20/0/20/0	100%	Delete	⬇	Progress
● Completed	● Whois	{"domain":"https://mattermost.ewi.tudelft.nl"}	20/0/20/0	100%	Delete	⇩	Progress
● Completed	● DNS	{"domain":"tudelft.nl"}	30/0/30/0	100%	Delete	⇩	Progress
● Completed	● PortForward	{"protocol":"tcp","port":22,"remotehost":"internetmonitor.e...	1/0/1/0	100%	Delete	⬇	Progress
● Completed	● Traceroute	{"domain":"staxoft.nl","maxhops":30,"protocol":"icmp"}	1/0/1/0	100%	Delete	⇩	Progress
● Completed	● DNS	{"domain":"google.com"}	20/0/20/0	100%	Delete	⇩	Progress
● Completed	● PortForward	{"protocol":"tcp","port":22,"remotehost":"internetmonitor.e...	30/0/30/0	100%	Delete	⬇	Progress
● Completed	● Whois	{"domain":"internetmonitor.ewi.tudelft.nl"}	5/0/5/0	100%	Delete	⇩	Progress
● Completed	● PortForward	{"protocol":"tcp","port":22,"remotehost":"internetmonitor.e...	1/0/1/0	100%	Delete	⬇	Progress
● Completed	● PortForward	{"protocol":"tcp","port":443,"remotehost":"internetmonitor....	20/0/20/0	100%	Delete	⬇	Progress
● Completed	● PortForward	{"protocol":"tcp","port":22,"remotehost":"internetmonitor.e...	1/0/1/0	100%	Delete	⬇	Progress

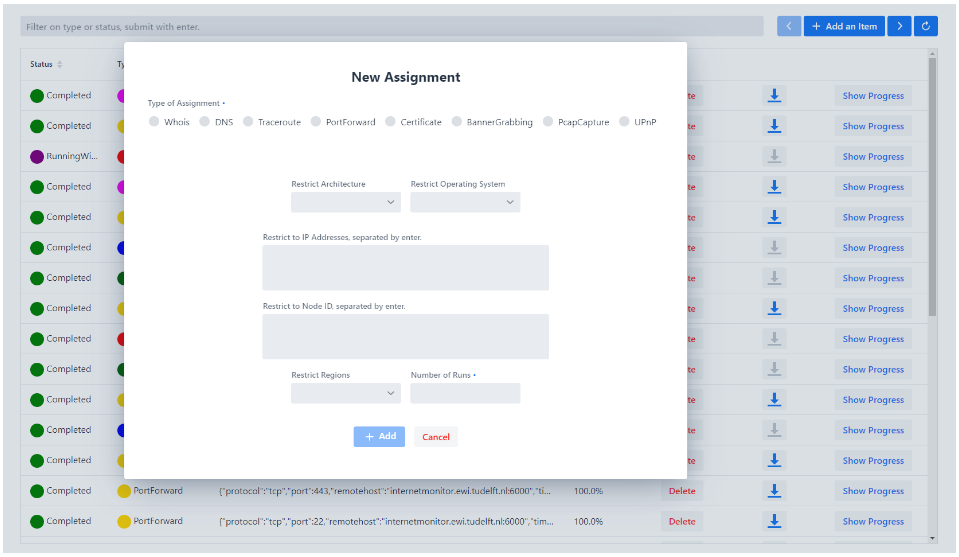
Figure 7.3: An example of the grid format

As figure 7.3 shows, a user is able to filter the data (more about that in section 7.2.2), navigate through different pages, add a new assignment (more about this in the next section about dialogs), refresh the grid, and interact with the different assignments. We created a way to click on an item in the list to view more information about the parameters. The buttons are used for various other features, like deleting the assignment, downloading the result and viewing the progress of the assignment. We wanted to make it easy for users to distinguish between assignments, so we used colours for each assignment and progress status. Another way to show the progress is by showing it as a %. This will allow the user to quickly see the progress of the assignment. Creating the grid was quite easy, but finding the balance in the amount of information on display was challenging and time consuming.

To ensure that the data is correctly shown on the list we use our own very simple grid implementation, `MainGrid<T>`. This class is then extended by, for instance, `AssignmentGrid`, which sets all columns for the grid showing an assignment. By using custom implementations of the grid we ensure that every column is in the correct order and has the correct size.

Dialogs

Dialogs are used to show popups on the screen. We used these to show information, or allow for adding/editing items. We ensure that the program is user friend by providing clear windows that shows all necessary information, but not too much to avoid overcrowding the screens.



New Assignment

Type of Assignment •

Whois · DNS · Traceroute · PortForward · Certificate · BannerGrabbing · PcapCapture · UPnP

Restrict Architecture | Restrict Operating System

Restrict to IP Addresses, separated by enter.

Restrict to Node ID, separated by enter.

Restrict Regions | Number of Runs •

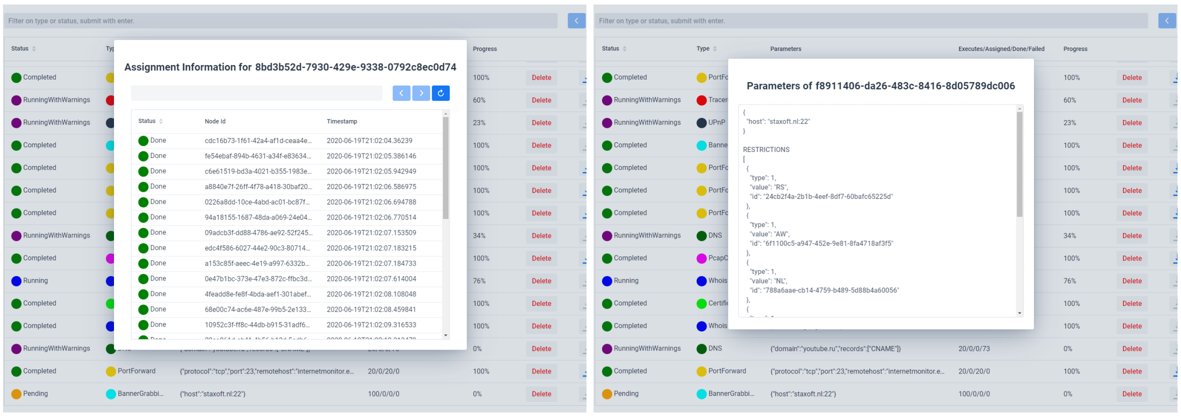
+ Add | Cancel

Completed	PortForward	{"protocol":"tcp","port":443,"remotehost":"internetmonitor.ewi.tudelft.nl:6000","ti...	100.0%	Delete		Show Progress
Completed	PortForward	{"protocol":"tcp","port":22,"remotehost":"internetmonitor.ewi.tudelft.nl:6000","tim...	100.0%	Delete		Show Progress

Figure 7.4: An example of a dialog for adding a new assignment.

The popup as shown in figure 7.5 has been created to allow an admin to add new assignments from within the application. It will do all of the backend work for the admin; they only have to enter the correct values.

Figure 7.5 shows a way that dialogs are used for showing more information. In this dialog we show some more information about an assignment. We show what nodes have done them and at what time. We also show whether or not the node has completed the assignment. Dialogs can also be used to show the parameters of the assignment, which might come in useful when adding a lot of assignments.


Filter on type or status, submit with enter.
Status
Type
Parameters
Executes/Assigned/Done/Failed
Progress
Delete

Assignment Information for 8bd3b52d-7930-429e-9338-0792c8ec0d74

Status	Node Id	Timestamp
Done	cdc16b73-1f61-42a4-af1d-ceaa4e...	2020-06-19T21:02:04.36239
Done	fe54ebaf-894b-4631-a34f-e83634...	2020-06-19T21:02:05.386146
Done	c6e61519-bd3a-4021-b355-1983e...	2020-06-19T21:02:05.942949
Done	a8840e7f-26ff-4f78-a418-30baf20...	2020-06-19T21:02:06.586975
Done	0226a8dd-10ce-4abd-ac01-bc87f...	2020-06-19T21:02:06.694788
Done	94a18155-1687-48da-a069-24e04...	2020-06-19T21:02:06.770514
Done	09adcb3f-dd88-4786-ae92-52f245...	2020-06-19T21:02:07.153509
Done	edc4f586-6027-44e2-90c3-80714...	2020-06-19T21:02:07.183215
Done	a153c85f-aeec-4e19-a997-6332b...	2020-06-19T21:02:07.184733
Done	0e47b1bc-373e-47e3-872c-ffbc3d...	2020-06-19T21:02:07.614004
Done	4feadd8e-fe8f-4bda-aef1-301abef...	2020-06-19T21:02:08.108048
Done	68e00c74-ac6e-487e-99b5-2e133...	2020-06-19T21:02:08.459841
Done	10952c3f-ff8c-44db-b915-31adf6...	2020-06-19T21:02:09.316533

Parameters of f8911406-da26-483c-8416-8d05789dc006

```
{
 "host": "staxoft.nl:22"
}

RESTRICTIONS
[
 {
  "type": 1,
  "value": "RS",
  "id": "24cb2f4a-2b1b-4eef-8df7-60bafc65225d"
 },
 {
  "type": 1,
  "value": "AW",
  "id": "6f1100c5-a947-452e-9e81-8fa4718af3f5"
 },
 {
  "type": 1,
  "value": "NL",
  "id": "788a6aae-cb14-4759-b489-5d88b4a60056"
 },
 {
```

Status	Type	Parameters	Executes/Assigned/Done/Failed	Progress	
Completed	PortFo			100%	Delete
RunningWithWarnings	Tracer			60%	Delete
RunningWithWarnings	UPnP			23%	Delete
Completed	Banne			100%	Delete
Completed	PortFo			100%	Delete
Completed	PortFo			100%	Delete
Completed	PortFo			100%	Delete
RunningWithWarnings	DNS			34%	Delete
Completed	PcapC			100%	Delete
Running	Whois			76%	Delete
Completed	Certifi			100%	Delete
Completed	Whois			100%	Delete
RunningWithWarnings	DNS	{"domain":"youtube.ru","records":["CNAME"]}	20/0/0/73	0%	Delete
Completed	PortForward	{"protocol":"tcp","port":23,"remotehost":"internetmonitor.e...	20/0/20/0	100%	Delete
Pending	BannerGrabbi...	{"host":"staxoft.nl:22"}	100/0/0/0	0%	Delete

Figure 7.5: An example of a dialog for showing more information about an assignment.

Before we decided on working with dialogs we had the idea to work with popups, since they were very easy to implement. But problems arose when these grew too big. Since the popups are anchored to a button it would fill up the whole screen without clearly showing what was clickable and not. So after this issue, we went on looking for a new way to implement these sub-windows, and while dialogs were a bit harder to implement, they do result in a cleaner looking *UI* overall.

7.2.2. Data Controller

While Vaadin uses `ListDataProvider<T>` to provide data to a grid, we decided on using our implementation, `DataController<T>`. By doing this, we also allow for filters and the adding/replacing of items. The `DataController<T>` can be extended for every type of data we provide to the `Grids` to ensure that the correct layout is shown, and all features of said data work correctly.

7.2.3. REST Client

The REST client is what combines the frontend to the backend. By using this client we ensure that data is always up to date on a refresh. A REST client works by sending out GET, POST or DELETE requests to the server. To ensure that the code is easy to read we decided to create a `RestClientUtil` class which will do the basic (authorized) GET, POST, and DELETE requests. And create the `RestClient` class to nicely wrap these requests. So instead of calling `getRequest(url, params..., apiKey)` the user can, for instance, call `getAssignments(index, apiKey)` and the method will handle the rest. Doing it like this will enable someone to easily add new requests just by creating another method in `RestClient`.

7.3. Future work

The components discussed in the previous sections are all the features currently implemented, but should we have had more time we would also have implemented the following few features.

7.3.1. Admin Dashboard

Having a dashboard is a great way to show a lot of information on one screen. For instance, the number of nodes currently doing an assignment or the number of assignments. But, this is also something that is not necessary for the program to work, and thus it was given a low priority, and eventually scrapped due to time constraints.

7.3.2. Generic Error Handler

Right now most errors for calls are handled in the method they are used in, but most of the logic whenever an error occurs is the same: write it to a log and show it. Creating a generic error handler was one of the methods we thought would help with this program, but we decided against implementing it for now due to time constraints, and other things having a higher priority.

7.3.3. Historic Data

One of the things that the client would have liked to see was the use of historic data on the map screen. But due to time constraints, the backend ended up scrapping the feature and we decided, together with the client, to also not create any visuals for this.

7.3.4. Conclusion

Having these features would complete the configuration part of the visualisation. It will allow a user to do almost anything through the *UI*. The reason that these features are not yet implemented is due to prioritising. We believe that the features we currently have implemented are enough to use the program for the client.

7.4. What we could/should have done differently

While Vaadin is a very powerful tool to create simple *UI* and *UX* elements, we did notice that there were some shortcomings. It was very difficult to integrate our Static HTML map page into the program, and we ended up doing it differently. We also encountered some issues with features that we wanted to use, but they ended up being PRO features. This meant that we had to settle for some less flashy designs, to avoid using these PRO features. And while there is a lot of documentation, a lot of documentation is out of date and uses older versions of Vaadin. Vaadin tends to be a bit buggy, it will sometimes not show information in a grid without any proper explanation, or the grid will all of a sudden be stretched more than the size of your screen.

Should there have been more time, we'd put in more effort in creating the best design we can. But if we had to do this project again we would most likely pick some other library/language for writing the visualisation, like Bootstrap or React JS.

7.5. Conclusion

When working on the visualisation we realised that you are never really done, there are always things that you can improve or colours that need to be changed. But we are all very happy with the progress we have made and the product we have created. The visualisation contains a lot of features that were not required by the client. While we did not include one of the desired aspects of our client, we implemented many additional features that greatly benefit the overall product. All of this was carried out after consulting our client. Working on this project gave us a lot of new experience in *UI* and *UX* designing, both of which will most likely be very useful in future projects.

7.6. SIG Feedback

The overall score we received from SIG was a 4.5 which means that the maintainability is very high, and thus little feedback had to be resolved.

7.6.1. What did we resolve

In this subsection we will explain what feedback we resolved.

Code Duplication

There was a little bit of code duplication, this was all very easy to fix. We created separate methods to handle this duplication.

Unit Interfacing Violation

Unit Interfacing Violations have to do with too many parameters for methods. In our case these were violated on constructors. Since these constructors were only used for testing purposes, and the getters and setters were also created, it was very easy to just replace the constructors with a default constructor and setters.

Unit Size Violation

Unit Size Violations have to do with the length of the methods. It was easy to make the methods shorter by separating them, but it is unclear what SIG believes is a correct length of methods.

7.6.2. What did we not resolve

In this subsection we will explain what feedback we did not resolve.

High Coupling

This error was thrown on abstract classes for grids and data controllers, these classes are the basis of the visualisation and are used in almost every screen. So because of the abstract nature and the fact that it is used in a lot of places means that a higher coupling is normal to have.

Code Duplication

While we fixed some code duplication, there was one duplication violation that we did not end up fixing. These were violations in the `equals()` methods, but since these are just generated methods that have nothing to do with each other we decided not to do any unnecessary fixing.

Illegal Catches

Illegal catches have to do with catching general exceptions instead of more specific ones. Some of these were fixed, but some were not fixed. The ones that were not fixed had to do with our way of catching errors. The behaviour for any problem is to log the issue, it would be unnecessary to catch every exception and respond with the same behaviour.

7.6.3. Conclusion

Since our code already had a high score there was little feedback, and thus little code smells to resolve. Everything was resolved in a short amount of time, and what we did not end up fixing was discussed with the entire group before deciding on not fixing it.

8 Port Forward Protocol Design

One of the requirements for the Port Forwarding assignment, as requested by the client, is that the communication between the MEs and the PFS is encrypted and authenticated (see appendix A). Furthermore, the communication between the two should be reliable and efficient. In this chapter, we will discuss the protocol we developed during the researching phase of this project and how it is implemented in the end. We start by explaining the choice for the Transport Layer, before discussing the different kinds of protocol messages and finishing with the algorithms used for encryption and authentication.

8.1. Transport Layer

As mentioned before, the exchange of packets between the MEs and the PFS needs to be reliable. This is important to achieve the goals for the Port Forwarding assignment since every single packet that the client receives should also be received by the server and vice versa. There are a lot of things that can go wrong during the transport of packets over the internet. For example, packets can be dropped, arrive in a different order than they were sent, etc. Without any reliability, successfully spoofing the port for the attacker in the port forwarding assignment will not work. The delivery of each packet needs to be guaranteed and in order.

The client gave us two choices for the transport layer, either UDP or TCP. Since UDP is a connectionless protocol, which means that it is fire and forget, there are no reliability guarantees. With UDP, reliability needs to be implemented by the protocol itself, by keeping track of packets sent, resending dropped packets, etc. Implementing the reliability in the protocol would be like reinventing the wheel and take time to get completely correct. That is why we decided on running the port forward protocol on top of TCP, a connection-based protocol which is reliable and takes care of reordering packets, resending packets when they get dropped, and a lot of other issues to achieve reliability. The only disadvantage when using TCP is the additional overhead since maintaining the connection requires more traffic and complexity. After testing, we concluded that the additional overhead of TCP is not an issue for the amount of MEs the client wants to run at the same time and thus a perfect solution in this case.

8.2. Protocol Messages

The protocol needs to have several kinds of different messages that indicate events that can occur during Port Forwarding. In this section, the different messages of the protocol are explained and discussed and why they are needed. In total, there are five different protocol messages, each with the same general layout as shown in figure 8.1.

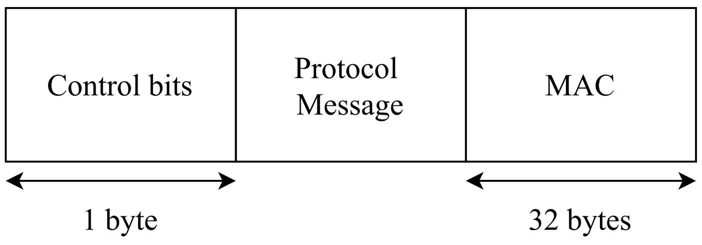


Figure 8.1: General structure for protocol messages

As can be seen in figure 8.1, the first field contains the `Control bits`, consisting of one byte. The decision for using one byte was that it is the smallest unit available. The bits here indicate the kind of protocol message that is carried inside. In total, there are $2^8 = 256$ different messages possible but, in the current version of the protocol, only five are in use. Therefore, there is plenty of room left to extend the protocol with more messages if needed in the future. After that, the actual protocol message, indicated by the control bits, can be found. This message can have different lengths depending on the type and the different types are explained below in the subsections. Each message ends with a Message Authentication Code (`MAC`), which provides authenticity of the message. More information about the MAC can be found in section 8.3.

8.2.1. New Connection

The protocol is initiated by a ME which sends a `New Connection` message to the PFS. This message type, with control bits 00000000, indicates that the ME wants to initiate the protocol with the server and start communication. The different fields for the `New Connection` message can be seen in figure 8.2.

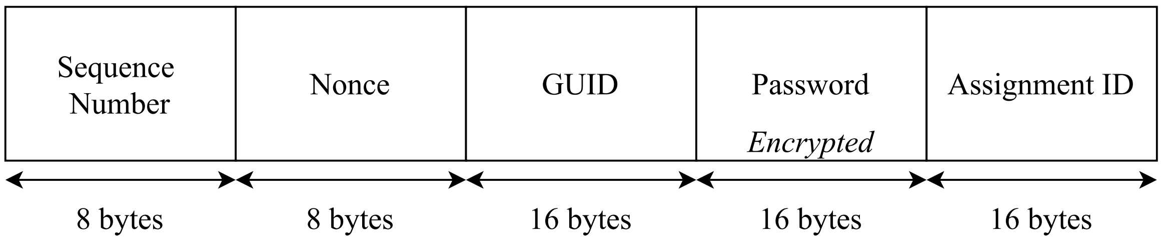


Figure 8.2: Fields for the New Connection message

The `Sequence Number` and `Nonce` are used as the Initial Vector (IV) during encryption and decryption of messages from the ME. More details about the sequence numbers and nonces can be found in the security section 8.3. Each ME is uniquely identified using a `GUID`, a random 128-bit value, and this field identifies the precise ME that the server is dealing with. The `Password` field is encrypted using the ME's API key and contains the ME's unique password, which is needed to upload files to the server. Lastly, the `Assignment ID` indicates the specific ID of the port forwarding assignment that this specific ME is doing.

All of this information is required to initiate communication with the server. It identifies the ME to the server, so it can look up the API key and decrypt the `Password` field. The `Password` field is used when uploading files to the backend and makes sure that an ME is authorized to do so. To successfully tamper with the communication, or to impersonate an ME, an attacker will have to obtain both the password and API key.

8.2.2. Confirm Connection

After the server receives a `New Connection` message, it looks up the API key for the specific ME in the database, authenticates the message using the `MAC` and decrypts the `Password` for later use.

This message type is indicated by the control bits 00000001 and indicates that the initialization of the protocol was successful and that the server is ready to receive traffic from the ME. The different fields for this message can be seen in figure 8.3.

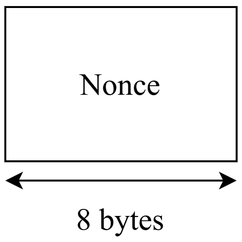


Figure 8.3: Fields for the Confirm Connection and New Nonce messages

This message only contains the Nonce field, which is part of the IV used for encrypting and decrypting the server's messages.

8.2.3. Close Connection

A lot of things can go wrong during the communication between an ME and the server. For example, the protocol fails, the server runs out of memory, the database is offline or a lot of other unexpected failures. To communicate that an error has occurred at one of the participants of the protocol, they can send a `Close Connection` message to the other party, indicated by the control bits 00000010. This is only done in the event that the error is unrecoverable. Upon receiving this message, the connection is closed. The different fields for this message can be seen in figure 8.4.

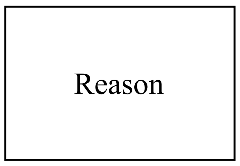


Figure 8.4: Fields for the Close Connection message

This message contains a `Reason` field, which is a zero-terminated string containing a description of the error that occurred. There is no length field required here, since the end of this field is indicated by the zero byte and this byte does not occur in a string otherwise. The reason for the error is logged locally and also send to the backend.

8.2.4. New Nonce

As will be discussed in subsection 8.3 about security, the nonce needs to be refreshed when the sequence number reaches its maximum number of 2^{64} to not compromise the security of the protocol.

The `New Nonce` message type is indicated by the control bits 00000011 and indicates that, from this message onward, a new nonce is used for encryption and decryption of the messages from the party that send the `New Nonce` message. The fields are the same as the `Confirm Connection` message and can be seen in figure 8.3.

8.2.5. Packet Inside

The last protocol message is the `Packet Inside` message, indicated by control bits 00000100. It indicates that a packet is carried inside of this message and should be handled accordingly by who receives it. The server saves the packets for storage and possibly forwards them to a Glue instance that analyzes the contents and determines what to reply (see chapter 5). The ME, on the other hand, injects the packet on the network upon receiving it. The fields for this message type can be seen in figure 8.5.
The `Sequence Number` is again used for encryption. The `Length` field indicates the length of the `Payload` field, which is required for decoding. Since the program essentially receives a stream of bytes, we need to know until where to read. The `Payload` can have any length up until $2^{16} = 65536$ bytes (the maximum length of an IP packet), so it only needs to be 2 bytes. For obvious reasons, the `Length` field is required instead of always using a `Payload` size of 2^{16}. Otherwise, there would be a

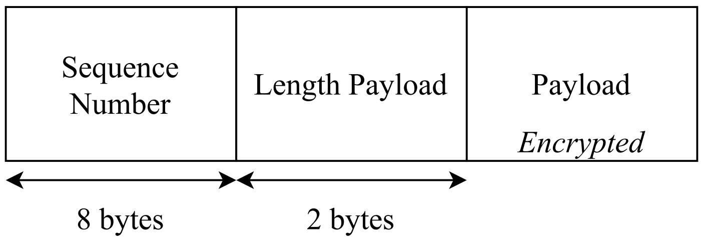


Figure 8.5: Fields for the Packet Inside message

huge amount of overhead (and the consequences of more overhead on the network, sender, receiver, etc) since not every packet is the maximum length of 2^{16}.

In our project, the `Payload` field is not exactly the IP packet, but also contains one additional field for constructing correct Ethernet frames. The server sends the Ethernet Type, which is required for constructing Ethernet frames, along with the actual packet. Because of this, the ME does not have to decode each packet it receives to determine the correct Ethernet type before injecting it in the network, which saves processing power for a small overhead per packet sent.

Lastly, the `Payload` field is encrypted with the API key of the ME to adhere to the requirements from the client about encryption. The `Payload` contains packets which can contain sensitive information and needs to be encrypted to make sure that no eavesdropper can determine the contents.

8.3. Security

In this section, we will document and explain the choice of the encryption algorithm and other details related to the security of the protocol. As can be seen in the previous section, all sensitive information is encrypted in the protocol and authenticated with a MAC. In this case, that is the password and any payloads that are sent over the network between the ME and server. All other fields either do not carry sensitive information, so they do not require encryption or are necessary to decrypt the ciphertexts.

8.3.1. Encryption

All encrypted fields use AES-256 encryption in Counter (CTR) mode[1]. The details about encryption are out of scope for this report, but this decision was made together with the client. The encryption and decryption in CTR mode need an IV. The IV is 128 bits in total, where the upper 64 bits are a Nonce and the lower 64 bits a Sequence Number. Each nonce in the protocol generated with a cryptographically secure pseudorandom number generator. The sequence numbers used in the protocol start from 0 and increase by 1 for each message sent, which guarantees unique IVs necessary for the security.

To extend further upon the unique IVs, the security is compromised if we reuse IVs, since encrypting the same messages with the same IV results in the same ciphertext. If we were using a different IV for each message, the ciphertext would be different and an attacker cannot see if the plaintexts are equal. Reusing IVs would open up the opportunity for attacks. To guarantee the uniqueness of the IVs, we need a new nonce once the sequence number reaches its highest value. This is guaranteed by the `New Nonce` message in the protocol.

Lastly, it is important to refresh the key used periodically. This makes sure that, in the event of an attacker obtaining your key, the attacker can only obtain a part of the communication with the server and not all of it. We have decided to refresh the API keys weekly, which means that an attacker can only decrypt a maximum of one week of data.

8.3.2. Authentication

The last part of the security of the protocol is authentication. Each protocol message has a MAC field which authenticates that this message was sent by a party that has access to the API key. Any alter-

[1]`https://tools.ietf.org/html/rfc3686`

ation to the original protocol message will result in a MAC failure and changing the MAC to a valid one is as good as a random guess when using secure MAC algorithms.

Each MAC field is generated by an HMAC SHA256 algorithm over all the protocol message fields using the API key. The MAC is recomputed by the receiver before continuing to do anything else.

8.3.3. Errors

To make sure the security of the protocol stays intact, any errors that occur in either the decryption phase or MAC validation phase will abort the connection between ME and server immediately. Not closing the connection when an attack is happening can result in a compromise of security. This is the only viable option for us to do when something goes wrong related to security.

8.4. Improvements

For now, the encryption and MAC algorithms that we use are considered safe with key sizes of 256 bits[3]. However, it could be that a flaw in one of the algorithms is discovered later, which would require a switch of algorithms. Should computers become fast enough to brute-force the current key length, the key size will need to be increased accordingly.

Furthermore, there is lots of room for other protocol messages that can be indicated by the control bits. If the client wants new options for port forwarding or the protocol needs to be extended with more messages, this can be easily done. The only catch here is that both server and ME need to run the same version of the protocol. A version indicator for the protocol is a good improvement to add to the current protocol to identify if both the ME and server are compatible.

Lastly, the size of the control bits can be reduced to four bits, since $2^4 = 16$ is still a lot of playroom for more types of messages. But since there is no use for the other four bits yet, this is not possible. One possible use for the other four bits is to extend the current protocol with a version indicator. The current protocol contains no version indicator yet, but this is needed when the protocol is extended in the future. For now, we think that current computers and network speeds should be able to handle the overhead of four bits easily.

9 Process

In this chapter, we explain the process which we went through during the project. We will explain the development strategies and the communication between all parties involved. We will end this chapter with a conclusion that will give more insight into what we might do differently next time.

9.1. Development strategies

At the start of the project, we defined development methods which we were going to use. Firstly, our iterative development would happen in weekly sprints. The choice for weekly sprints over biweekly sprints was made based on the duration of the project. The project only lasts ten weeks from which the first two weeks is research, so doing biweekly sprints would only give us four prototypes. Having only four prototypes would result in less and later feedback from the client/coach on the product.

In each weekly sprint, we planned four meetings, namely:

- Team meeting on Monday's for creating a backlog
- Coach meeting on Wednesday's
- Client meeting on Thursday's
- Team meeting on Friday's for creating a retrospective

The team meetings were mainly for deciding and evaluating the sprint, while the coach and client meetings were for progress updates and questions. The number of meetings could increase if something needed to be discussed with the whole group, for instance when designing the architecture. These sprints turned out to be very useful throughout the project. Mainly because we were able to easily see our progress and what we were going to do this for the week. This also meant better communication with the client and the coach about the current status.

Secondly, we defined roles for every team member. We assigned the following roles:

- **Chairman**: in charge of creating agenda's and contacting client/coach
- **Secretary**: takes notes during meetings and helps the chairman with planning meetings
- **Scrum Master (twice)**: in charge of creating the backlogs and retrospectives
- **Head of Report**: in charge of making sure the report is updated regularly and in a consistent style

These roles allowed us to better focus on our tasks. Although, during the middle of the project we realised that the Head of Report function was difficult to satisfy. This was because there was simply not enough time to focus on the report during the development of the project. However, in the end, when writing the report the Head of Report could focus again on this task.

Thirdly, developing the project would be done using test-driven development (TDD). We used this method since “Test-first programming, or test-driven development, is included as a best practice in agile software development methodologies”[7]. In addition, TDD would allow us to quickly identify errors and, therefore, reduce the time we spend on debugging the application. TDD went very well for the first 5 weeks (until sprint 6) of development. The last two development sprints we did less TDD, as we were more focused on fixing bugs. TDD was very useful for implementing brand new components at the start, but for bug fixing and minor features later it was not so useful to us. This was because we still had quite a few features left to finish, and with the limited time, it was not ideal. Also, because we were inexperienced with Go, it meant that we had to experiment a lot with the language while developing which hindered TDD. Luckily, once we got the hang of Go, we could utilise TDD to the max.

To assist us in the development process, we also used GitLab[1] as our version control system and the continuous integration and continuous deployment (CI/CD) that can be configured on GitLab. The CI allows us to maintain a high-quality codebase since the CI will check whether all tests pass, output the coverage, check whether the style is consistent and checks the code maintainability. Setting up the Gitlab CI/CD for each component was quite the task. We did not have all the required experience with setting up all the CI/CDs. After trial and error, we finally managed to configure the CI/CDs. Each CI is configured in such a way that we always have a test, code quality and deploy stage. We believe that these three stages give us enough confidence in our code. We used Code Climate[2] for the quality stages with their default configurations for analysis[3].

Furthermore, as an additional quality assurance within the team, we made sure that at least two team members reviewed each merge request before merging it in the dev branch. Team members reviewed the code critically, made sure the documentation was clear, and made sure that enough tests were written.

9.2. Communication

During the course of the project, we had several communication channels to communicate within the team, with the client and with our coach. In this section, we will explain how this communication was done, what we were able to discuss with our client and coach, what they were able to help us with and any challenges we encountered. Overall, we think the communication was efficient and effective with everyone.

9.2.1. Team

All communication within the team was done via text chat and meetings. We set up a server for the project members with several text channels like announcements, meeting notes, bugs, report, design, etc. Making these separate channels made sure that information can be quickly found by subject and resulted in more efficient communication overall. Without channels, information can get lost when there is a lot of communication and catching up with messages is harder to do.

In the meetings, we discussed sprint plannings, retrospectives, or any other problems that needed fixing and that could not be done effectively via text. We made internal rules about how we handle conflicts within the team. These rules consisted of the famous cake rule within project teams: a late team member needs to get a cake for the rest of the group if not present without notifying the team beforehand (with a good reason). Any conflicts that we were not able to resolve on our own are communicated with our coach. We are happy that we did not have any major issues and that we were able to resolve the minor ones on our own without getting any other parties involved.

All team members were mostly present throughout the whole day, which meant that we could quickly arrange meetings or get things done when we needed help from each other. We are quite happy with the effectiveness and efficiency that we were able to work with each other and how the communication was handled, especially in the times that physical meetings were not possible due to COVID-19 restrictions. Although physical meetings and working in the same space will always be better, we think

[1] `https://about.gitlab.com/`
[2] `https://codeclimate.com/`
[3] `https://docs.codeclimate.com/docs/default-analysis-configuration`

we managed it quite well given the circumstances.

9.2.2. Client

All communication with our client was done via text chat and digital meetings. The text chat was used to ask questions or arrange meetings and was very effective for that purpose. Digital meetings were held once a week or when a big question or issue had to be discussed.

The chairman sent the client an agenda before every meeting containing everything we wanted to discuss and the questions we have. Any files necessary were sent with the agenda, which always included the sprint planning for that week. Each sprint was made in a spreadsheet and contains the user story, deliverable, description, tasks, assignee, time estimation, status, time spent and any remarks. Besides that, all issues in this spreadsheet were also created on GitLab. This allowed the client to keep track of our progress for that week, the project overall and any issues we were struggling with.

An example of a meeting agenda is given below:

```
Here is the meeting agenda for tomorrow’s meeting at 1 PM:
- Questions
    - List of questions...
- Demos
    - List of demos...
- Discuss sprint
- Further questions if needed
```

It is good to note here that in the beginning, especially the early stages of development, there were not a lot of demos.

The client was able to help us with most problems that we encountered during development or questions about technologies he wanted us to use. We made sure that we got his requirements clear on paper and that his wishes regarding the end product were fulfilled. This ranged from sending a quick text message with a question regarding one of his features to having a meeting with him to get the requirements and wishes clear for us.

9.2.3. Coach

The communication with our coach was done via text chat and digital meetings as well. Just like with our client, we sent our coach an agenda before each meeting with what we wanted to discuss. This also included the sprint planning, so our coach was also up-to-date with our progress and issues. The same format for the meeting agenda as before was used, except any demos. Meetings were held once a week and sometimes bi-weekly if there was not much to discuss.

Our coach was able to help us with any issues that we encountered, be it within the team, with the project itself or with our client. He gave us feedback on the report drafts that we submitted and advice on how to tackle problems. One problem was, for example, how to effectively communicate with the client that not all features can be realised in-time, how we want to proceed with the last few weeks and what the client wants to focus on. We are happy with the advice our coach was able to give us for this problem and other smaller issues, like with the GitLab servers (see section 9.3).

Furthermore, the coach was able to give us feedback on our report that we submitted to him and trial running our presentation. It is always good to hear the feedback from someone outside of our own team to get a fresh perspective. This was especially useful since we can get blindsided on something that is clear to us but might be confusing or unclear to others not directly involved with the project.

9.3. Problems

During the project, we encountered several problems that slowed down our progress. In this section, we will discuss the problems we had with the TU Delft servers, the GitLab servers from the client and the change of requirements during the project. These are not all of the issues we had during the project, but external problems that influenced our progress. The other problems are discussed in chapters 4 to 7.

9.3.1. TU Delft Servers

For testing and deploying our application, our client arranged some Virtual Machines (VMs) on the TU Delft VM network. We initially thought that these servers were ready for use and powerful enough to suit all our needs. However, when it came to deploying the application, it turned out that the servers were not ready. The ports that were promised to us were not actually open. This lead to us wasting several days debugging because we assumed that the server that was provided would be as was promised. This was reassured multiple times by both our client and our contact at TU Delft. However, this later turned out to not be the case. After several days of emailing back and forth, the issue was resolved. This had, however, cost us a lot of time that would have spent better otherwise.

After spending a few days testing, we found out that the specifications of the server were not what we had anticipated. We found out that the performance of our application was unexpectedly slow when compared to performance on our own machines. An investigation into the specifications of the machine revealed that it was only running 2 GB of RAM and only a few virtual CPU cores. This meant that the hardware was severely underpowered for what we were trying to run on it.

After discovering the lacklustre performance of the machine provided to us, we contacted our contact at the TU Delft to request a new server and upgrade the current one. This contact was very slow and there was a lot of misunderstanding of our needs and the possibilities that TU Delft had for us. To make matters worse, there were suddenly questions about the ports that we wanted to have opened in the firewall. Where this was previously no problem, it was suddenly not possible anymore to open them to the outside world, requiring us to connect using a VPN. This did not work for all team members, and due to the short time left in the project, we were forced to rely on the team members that were able to connect using the VPN.

In hindsight, a lot could have been done better during the communication. First of all, we should have checked better what the machines were and if they were configured correctly. However, almost none of our requests were correctly handled the first time and a lot of emails were needed to get what we actually needed. This cost us a lot of time and effort and caused a lot of frustration. All in all, this, on first sight, easy task cost us a lot of unnecessary time which could have been spent better on development.

9.3.2. GitLab Servers

At the beginning of the project, the GitLab servers of the client needed an upgrade to support the latest GitLab runners and features we use for the project. Something went wrong during this upgrade, which resulted in the data loss of one week worth of development. Luckily, we had most of the data locally on our own computers, but all GitLab project settings and files we edited via the Web Editor in GitLab were lost. This caused a slow down of two days, in which we had to set up our GitLab projects again and make sure all the Continuous Integration and code was working again.

During the project, the GitLab servers went down a few times as well. The downtime was usually a few hours to half a day and meant that we could not review each others code, merge it to the dev branch, or simply share code amongst each other easily. The reasons for the downtime was most of the time the server being updated to the latest version or due to full hard drives. The client was the only person that can restore and fix the servers and he was also busy during the day, so issues like this could take a few hours before they were fixed. We estimate that the downtime of the servers during the project, which happened four or five times, caused a total slowdown of at most 2 days.

In the end, it was probably the best idea to do the development on our own servers that we have access to. This allows us to schedule any updates and immediately fix problems that occur.

9.3.3. Change of requirements

When nearing the end of the software development, we needed to discuss with the client what was still possible to achieve for the project and what we were not able to do anymore. Because of a lack of time, we decided together with the client to drop the requirements for historical data view and BGP data storing. The client wanted us to focus on implementing as many features as we still can in the last few weeks, which we prioritized on. This means that the project contains all of the features, even the could and should-haves, and the extra ones that we added. The client agreed that these extra features, together with the other ones, are a worthy replacement of the historical data view and BGP data storing and these features will be left for future development within his own team.

This change of requirement did not necessarily slow down our progress, but it did mean a change of direction and focus for our team. We had several internal meetings where we discussed what we can still achieve and communicated this with the client and discussed it with our coach. With all meetings together, the slow down is at most one day.

9.4. Conclusion

In this chapter, we discussed our process while working on the project. We started by explaining our development strategies, and while we are happy with how it went overall, there are always points of improvement. As mentioned before, TDD works wonders during the start of the process, but as we were nearing the last couple of weeks this proved to be more and more difficult to adhere to this development strategy. To do this differently next time we will look at each sprint on its own and see what strategy would be best. So maybe for the first 5 weeks use TDD to implement new features, and once all features are implemented move away from TDD to another strategy for bug fixing.

And while the roles were very clear, the ‘Head of Report’ role did not have much to do until the last two weeks. But overall we were very satisfied with these roles, and it ensured that nobody ended up doing things twice.

In terms of communication, we are also very satisfied with how it went. All communication was clear and professional, but in the final couple of weeks it sometimes turned out to be more of a challenge to plan meetings. This was because the team, coach and client were busier than the first couple of weeks, or there was just nothing to discuss.

10 Ethical Implications

When developing software it is important to keep an eye on the ethical implications the software might have. During the development of our software, we were not directly faced with many explicit ethical challenges. We did, however, make some implicit ethical decisions, mainly to do with data and accessibility.

Firstly, the measurement may collect sensitive and/or personal data. This is because some assignments capture all traffic of the user network. When the data is compromised, this may expose the sensitive/personal data to the outside world. However, taking into consideration that most measurement endpoints will not be run on personal computers, but mainly on enterprise servers and virtual machines, and the fact that users are provided with the option to opt-out of this feature (by blocking the specific assignments), the ethical implication is minimized.

Secondly, our product does not delete any data after a period of time. The data stored is going to be used for analytic purposes in the future, as an expansion to our current project. There are two possibilities that we can consider for the storage of data. Either all data is anonymized (but as far as we know that is not possible for data that we upload) or data is deleted after a specified period. This was, however, not a task for this project and was thus considered or even thought about. This issue is left as future work and for research in the best possibilities to minimize any ethical implications for the users.

11 Conclusion

At the end of the project, we can conclude that it was a success. Internet monitor allows an admin to create new assignments, which are then picked by measurement endpoints and the results of these are stored in the database. Anybody can view the real-time traffic on the map screen and see what is currently happening.

Despite the changes that were made during the project by our client and us, the final product suits the requirement of the client and meets the expectations that we set ourselves at the beginning of the project. And even though we are satisfied with the result, there is still room left for improvements in the future. There are also always things that, in hindsight, should be done differently. But, all in all, we are very satisfied with the result of the project and the approach we took to develop it.

A MoSCoW

This appendix contains the MoSCoW list as given to us by the client. We will conclude with some extra features that were implemented, which were not originally on the MoSCoW list.

A.1. Measurement Endpoint

A.1.1. Must Have

- Connect to a control endpoint in regular intervals (heartbeat) and get a list of tasks via an HTTP API endpoint to process, execute them and report the results back.
- Perform DNS and whois lookup of a domain
- Run a traceroute using ICMP or TCP to a destination
- Open a local port and forward incoming traffic to the server, if responses are given by the server, send them out using the client IP address

A.1.2. Should Have

- Include authentication when connecting to the control endpoint (through an API token for example)
- Sign the results sent back as being authentic
- Connect to standard ports such as HTTP, SSH, Telnet to see that the service is up and upload the banner

A.1.3. Could Have

- Open the port through a firewall/router via a UPnP command and close it afterwards
- Download the certificate in the case for an HTTPs connection and send it to the control point

A.2. Backend

A.2.1. Must Have

- Download BGP datasets from RIPE and CAIDA using HTTP or their common API
- Store the results reported by the client in a database
- Offer a possibility to query the database for paths
- Admin config: provide an overview of connected clients, allow to whitelist/block clients, whitelist/block IP range to not participate and send results Admin config: configure which measurement tasks are running; tasks can be set to individual endpoints, groups (by country, tag etc.) or to all endpoints (once / periodically)
- Client config: user can see active endpoints and the data they have sent, configure own clients to participate in certain measurements or not (e.g. open local port and say which ones are allowed)
- API endpoint to make, edit, change, measurement

A.2.2. Should/Could Have

- Different users can “own”, create different measurements, possibly with quotas
- Jobs should be randomly handed out so that we don’t overwhelm destinations by sending out the same job to all endpoints

A.3. Visualisation

A.3.1. Must Have

- Provide a real-time view of the incoming traffic to the clients (e.g. https://www.fireeye.com/cyber-map/threat-map.html, https://threatmap.checkpoint.com/, https://cybermap.kaspersky.com/)
- Provide an AS-level view of interconnections on the Internet, and show which ones are currently deviating
- A slider to scroll through historical data and update the AS-level view and the connection patterns

A.4. Extra Features

A.4.1. Measurement Endpoint

- Option for DNS to query via remote resolver
- Option for traceroute via TCP or UDP on Linux
- Option to inject packets on measurement endpoint or not for port forwarding
- Server should save packets send from and to the measurement endpoint
- Server should communicate with ’glue’ that keeps track of connections and determines the packets to inject by creating the IP and TCP or UDP headers and communicating with a docker container for the actual payload
- New assignment: Measurement endpoint should capture all traffic that comes in (excluding local area network traffic and traffic to and from the backend) and report them back to the server in a pcap file
- Option to define custom capture filter for the new assignment from above

A.4.2. Backend

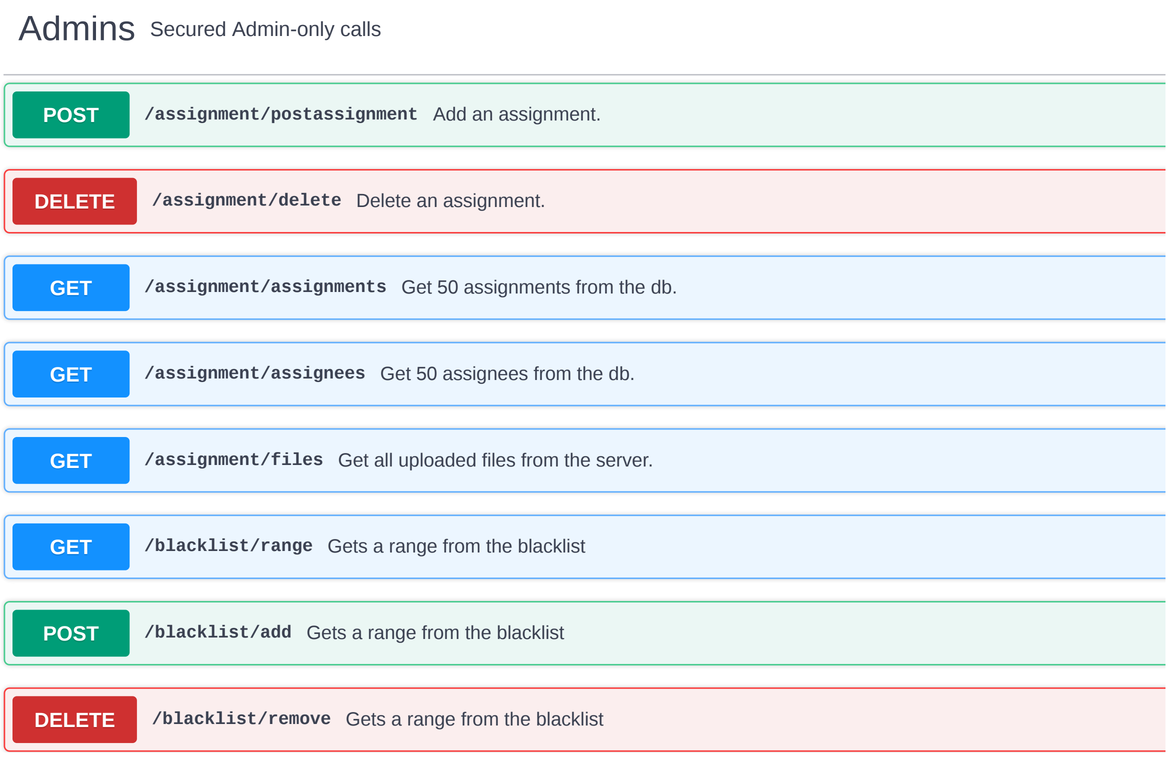
- All additional features from the ME are supported by the Backend
- Api key refreshing
- Automatic region code determining
- Status tracking of assignments
- Status tracking of nodes which are doing tasks
- A local serverside cache
- Kafka caching before writing to Hadoop
- Prevent MEs from doing assignments multiple times
- Database filtering on all assignments for a search bar in the visualisation
- Result downloading for PCAP capture and port forwarding
- Delete an assignment

A.4.3. Visualisation

- All extra features from the Backend and ME are able to work with the Visualisation
- Admin can see additional information regarding how many nodes got the assignment assigned, failed it and completed it
- Besides incoming traffic to Clients, display real-time view of the results send from the Clients to the backend

B
API Documentation

The full API documentation can be found here[1].



Admins Secured Admin-only calls

POST /assignment/postassignment Add an assignment.

DELETE /assignment/delete Delete an assignment.

GET /assignment/assignments Get 50 assignments from the db.

GET /assignment/assignees Get 50 assignees from the db.

GET /assignment/files Get all uploaded files from the server.

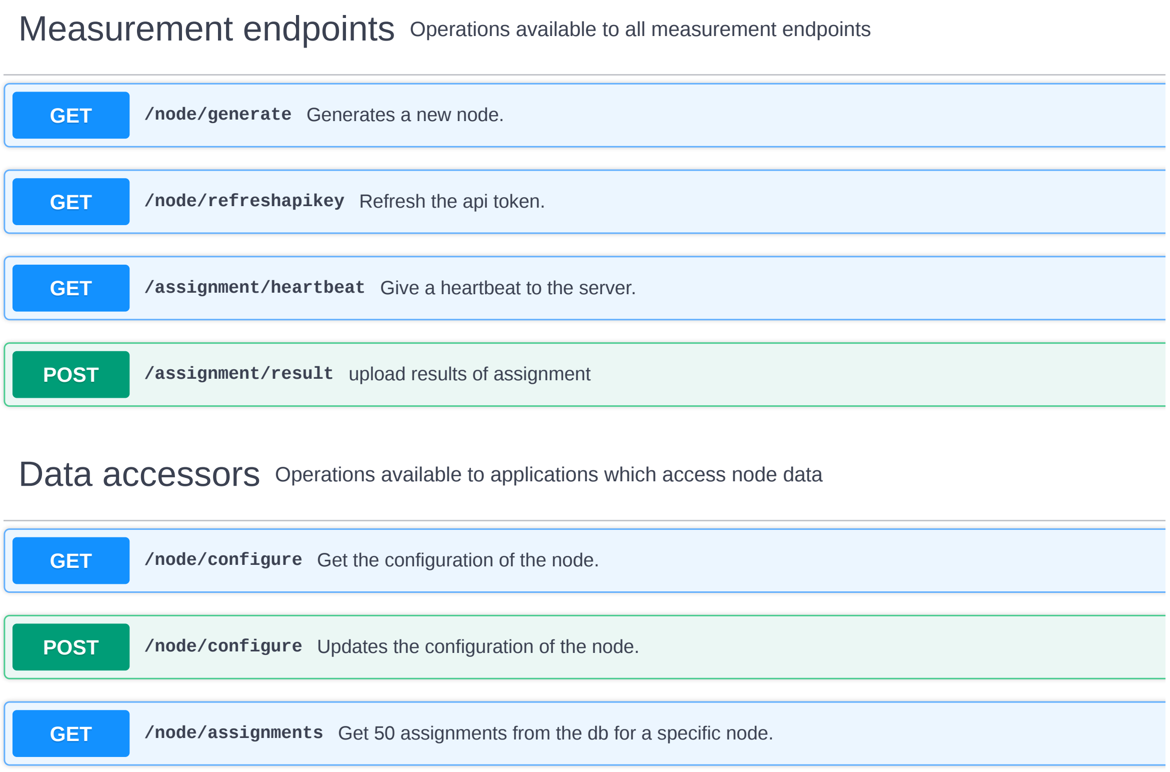
GET /blacklist/range Gets a range from the blacklist

POST /blacklist/add Gets a range from the blacklist

DELETE /blacklist/remove Gets a range from the blacklist

Figure B.1: The Admin calls to the API

[1]`https://app.swaggerhub.com/apis/alexj123/internetmonitor-backend_api/1.0.0`



Measurement endpoints Operations available to all measurement endpoints

GET `/node/generate` Generates a new node.

GET `/node/refreshapikey` Refresh the api token.

GET `/assignment/heartbeat` Give a heartbeat to the server.

POST `/assignment/result` upload results of assignment

Data accessors Operations available to applications which access node data

GET `/node/configure` Get the configuration of the node.

POST `/node/configure` Updates the configuration of the node.

GET `/node/assignments` Get 50 assignments from the db for a specific node.

Figure B.2: The other calls to the API

C
Original Project Description

The Internet is not a homogeneous infrastructure, but it is formed by linking up tens of thousands of networks through a protocol named BGP. Designed in the 1990s, this protocol contains no security protections, and essentially every network owner in the world can impersonate other companies and reroute their network traffic.

While most of the time this happens due to misconfigurations, this weaknesses has been exploited to steal money from bank accounts (e.g. the Route53 MyEtherWallet incident), for industrial espionage or even cyber warfare. On a typical day in the Internet, dozens of such hijackings occur, but can usually be resolved within minutes or hours of being detected. One of the detection tools is BGPMon, which provides a real-time map of BGP hijacks and route leaks on the Internet: https://www.bgpstream.com/. BGPMon is however end-of-life and will shut down at the end of 2019, and the Internet community will loose an important tool to monitor the health of the Internet and detect incidents.

In this project, we will develop a platform to perform a global monitoring and visualization of the state of the Internet. It will ingest data from public BGP routers, accumulate and visualize traceroute measurements and visualize where currently traffic flows across the Internet. The system will be built on a Java platform (Java EE / Vaadin), with a distributed database (MySQL cluster, or Hadoop/Impala) in the backend to enable custom querying on the accumulated dataset.

Global Monitoring and Visualization of the Internet

Cyber Threat Intelligence Lab **Presentation date:** July 2, 2020

Description

Internet Monitor provides a real-time overview of traffic from its measurement endpoints that are running across the world. An admin can enter specific assignments, like traceroute and DNS lookups, that a group of or all measurement endpoints need to execute. They report their results back, where it can be analysed for anomalies against data reported by BGP servers and historical data. The client can use this product in his field of research to perform analysis on the reported results or create honey pots for potential attackers.

The challenge for this project was developing a scalable and reliable backend, that can handle thousands of measurement endpoints uploading results at the same time, and having the measurement endpoints work on both Windows and Linux while supporting many different assignments that they can execute. In addition, everything needs to be coordinated from an admin page.

During the research phase, we performed a competitor analysis, made architecture designs, sequence diagrams and researched the languages to use. This gave us more insight in how the product will look like, how the different components should interact and what the best approach is. Lastly, a protocol design was made to perform the port forward assignment between a measurement endpoint and the server securely.

The process started with gathering a list of requirements from the client. Afterwards, we divided the team in multiple groups to work separately on the different components of the product. We held regular meetings and used the SCRUM methodology throughout the duration of the project. A few problems we encountered where issues due to cross-platform support, problems with port forwarding and scaling the backend efficiently. We were able to resolve most issues by simply helping each other.

The product is tested using unit tests, integration tests and system tests, automatically run by GitLab CI on each commit. The interaction between the backend, visualisation and measurement endpoints was tested manually throughout the project.

There are several things for this project that need to be implemented in the future. This is mainly storing the BGP data in the backend, implementing historical comparisons in the backend and visualisation, and performing the analysis on the stored results. Not all of the above were in the initial requirements and some requirements were slightly changed during the project that both the client and team agreed upon.

The final product is ready to be used, but might not be of much value yet. A few meetings have been planned with the client to help him and his team get started with the product we developed and how they can best extend and maintain it. They will develop the product further so that it contains aforementioned features. After that, the product can be used to its full potential.

Members of the Project Team

Rick Huisman
Role: Secretary
Contributions: Measurement Endpoint Developer
Interests: Cyber Security and System Programming

Alex Jeleniewski
Role: SCRUM Master
Contributions: Backend Developer
Interests: Artificial Intelligence and Big Data

Stefan Weegink
Role: Chairman
Contributions: Measurement Endpoint Developer
Interests: Image Processing and Concurrent Programming

Stef Kaptein
Role: SCRUM Master
Contributions: Backend Developer
Interests: System programming and Artificial Intelligence

Tom Saveur
Role: Head of Report
Contributions: Front-end Developer
Interests: UI Design and Machine Learning

Client: Christian Doerr - Cyber Security and Enterprise Security - University of Potsdam
Coach: Stjepan Picek - Cyber Security Research Group - Delft University of Technology

Contact Persons
Rick Huisman, rick.huisman@planet.nl
Stef Kaptein, stefkapte@gmail.com

The final report for this project can be found at: http://repository.tudelft.nl

Bibliography

[1] .org agreement appendix 5 whois specifications, 2006. URL `https://www.icann.org/resources/unthemed-pages/appendix-05-7e-2006-12-08-en`.

[2] Aug 2018. URL `https://support.microsoft.com/en-us/help/314868/how-to-use-tracert-to-troubleshoot-tcp-ip-problems-in-windows`.

[3] Elaine Barker. Recommendation for key management:. Technical report, May 2020. URL `https://doi.org/10.6028/nist.sp.800-57pt1r5`.

[4] Russell Brandom. Hackers emptied ethereum wallets by breaking the basic infrastructure of the internet. *The Verge*, April 2018. URL `https://www.theverge.com/2018/4/24/17275982/myetherwallet-hack-bgp-dns-hijacking-stolen-ethereum`.

[5] C. Doerr. Global monitoring and visualization of the internet. *BEPSys*, 2019. URL `https://projectforum.tudelft.nl/course_editions/19/projects/684`.

[6] J. Hawkinson and T. Bates. Guidelines for creation, selection, and registration of an autonomous system (as). BCP 6, RFC Editor, March 1996.

[7] Reid Kaufmann and David Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299, 2003.

[8] Stephen Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (s-bgp). *IEEE Journal on Selected areas in Communications*, 18(4):582–592, 2000.

[9] Q. Li, J. Liu, Y. Hu, M. Xu, and J. Wu. Bgp with bgpsec: Attacks and countermeasures. *IEEE Network*, 33(4):194–200, 2019.

[10] S. Murphy. BGP security vulnerabilities analysis. RFC 4272, RFC Editor, January 2006.

[11] A. Newton and S. Hollenbeck. Registration data access protocol (rdap) query format. RFC 7482, RFC Editor, March 2015. URL `http://www.rfc-editor.org/rfc/rfc7482.txt`. `http://www.rfc-editor.org/rfc/rfc7482.txt`.

[12] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL `http://www.rfc-editor.org/rfc/rfc793.txt`.

[13] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). RFC 4271, RFC Editor, January 2006. URL `http://www.rfc-editor.org/rfc/rfc4271.txt`.

[14] Ryan Singel. Pakistan's accidental youtube re-routing exposes trust flaw in net. *Wired*, June 2017. URL `https://www.wired.com/2008/02/pakistans-accid/`.

[15] Andree Toonk. Bgpstream and the curious case of as12389. *BGPMon*, April 2017. URL `https://bgpmon.net/bgpstream-and-the-curious-case-of-as12389/`.

[16] Q. Vohra and E. Chen. Bgp support for four-octet as number space. RFC 4893, RFC Editor, May 2007.