HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

CISCO

Streaming Telemetry

Study Thesis

Department Computer Science
HSR University of Applied Sciences Rapperswil

Author(s): Matthias Dunkel, Raffael Vögeli
Advisor: Prof. Laurent Metzger
Project partner: Cisco System (Switzerland) GmbH, Marcel Witmer

Contents

Part I

Exercise description

This is the exercise description we received from Prof. Laurent Metzger:

Classical network monitoring tools mostly manage the physical and data layer, and some network monitoring tools are able to draw all the links where a protocol is running. The management protocol used is SNMP, with traps and polling.

On the one hand, with the introduction of technologies like segment routing, the path that a flow takes in the network can depend on the service that the flow is part of. A monitoring should be able to show those paths per service.

On the other hand, streaming telemetry opens the door to a near real-time monitoring in a very efficient way.

The goal of this thesis is to innovate the way routing is monitored. The use case for this thesis is going to be based on segment routing, even if other routing technologies could be the streaming telemetry data source for future use cases. The technology involved is streaming telemetry.

The configuration of the test network and the analysis of the relevant sensors is not part of the SA. This falls under the responsibility of the Institute for Networked Solutions (INS). The infrastructure and this information will be given to the students. The purpose of the SA is to develop a WebUI. The main focus of this SA is on the frontend. Nevertheless, some backend functionality will be required too. The WebUI is going to display paths of services which are provided by the network. The solution architecture should scale well (100 routers and more).

About the backend:

- The backend should be able to receive the streaming telemetry data and store the information in an efficient way.
- Furthermore, it should be able to detect changes and store streaming telemetry data diffs.
- To provide traceability of network changes a history should be maintained.
- As an optional part, it should be possible to trigger events in case of a detected change.

About the frontend:

- The network topology should be displayed. Each router should be represented with an icon and a connection between two routers with a line.
- By hovering over a link/device, the most relevant information should be displayed (e.g. metrics). For example, the information with CDP/LLDP Yang Models received from streaming telemetry could be used. A change in the physical topology should be recognized and the display refreshed as real time as possible.

- On that dynamic topology, the path that the flow is following between routers should be indicated with a change of color. To select a path for a flow,
  1. A specific router has to be selected.
  2. Based on the selected router, the services which are hosted on this router should be displayed.
  3. Select the date and time (e.g. now, 4 hours ago, 1 day ago or the exact date and time)
  4. Optionally, there should be the possibility to select a destination router. If no destination router is selected, all destinations of this service are shown (if possible).
- Besides the above-mentioned flow, there could also be another way to choose what is displayed:
  1. Select a specific service.
  2. Based on the selected service, all routers which are hosting this service are shown. Next the router can be selected.
  3. Based on the already selected router, all possible destination routers are shown and one or more can be selected.
  4. Select the date and time (e.g. now, 4 hours ago, 1 day ago or the exact date and time)

The information from different Yang models is received via streaming telemetry and the information displayed is constantly refreshed.

Prof. Laurent Metzger

Part II

Abstract

When running a network, it is important for the operator to gain insight. They need to know which routers are online, and which network routes are taken. Additionally, they need to know how the network behaved in the past.

In order to obtain data from a network, Cisco added streaming telemetry to their routers. This is an approach in which data is not polled from the routers, but is sent as a continuous stream to a server.

Although network monitoring tools already exist, they mostly display the physical and data layer of the network. By introducing technologies such as segment routing into a network, the paths taken by packets in the network may depend on the service these packets are part of.

In this thesis we developed a collection of services which consume the streaming telemetry data from the routers. This data is then used to display a graph which represents the network with its routers and neighbors. Moreover, the application displays information about the individual routers. However, focus was placed on analyzing the segment routing paths. As a result the user can display the path packets take, in conjunction with a service.

As the networks can get very big, emphasis was placed on scalability of the solution. This was achieved by using a very scalable data stream-processing software, and by splitting the application into separate stateless services.

Part III

Management Summary

Initial Situation When maintaining a network as an operator, it is important to know which routers are available and how the data packets move through the network. It also gives them the advantage of being able to access past network topologies in case of problems and thus perform an error analysis. In recent years, a new way of sending data packets has emerged: Segment Routing. With the help of this new technology, packets are labeled with their forwarding path before they depart.

Existing monitoring systems currently mostly map the physical and data layers of a network. However, this new application enables the operator to also display segment routing paths of several services.

Furthermore, one can move through time and get information about the network in the past. Moreover, the network path that packets use between network nodes can display.

Procedure & Technologies The scope of this project was to receive data from the network, then process it, store it and display it when needed.

To receive data we used an already existing project from Cisco called "Bigmuddy Network Telemetry Pipeline" (or just "pipeline"). This pipelines job is to receive the streaming telemetry data and forwarding it. In our case to an Apache Kafka cluster.

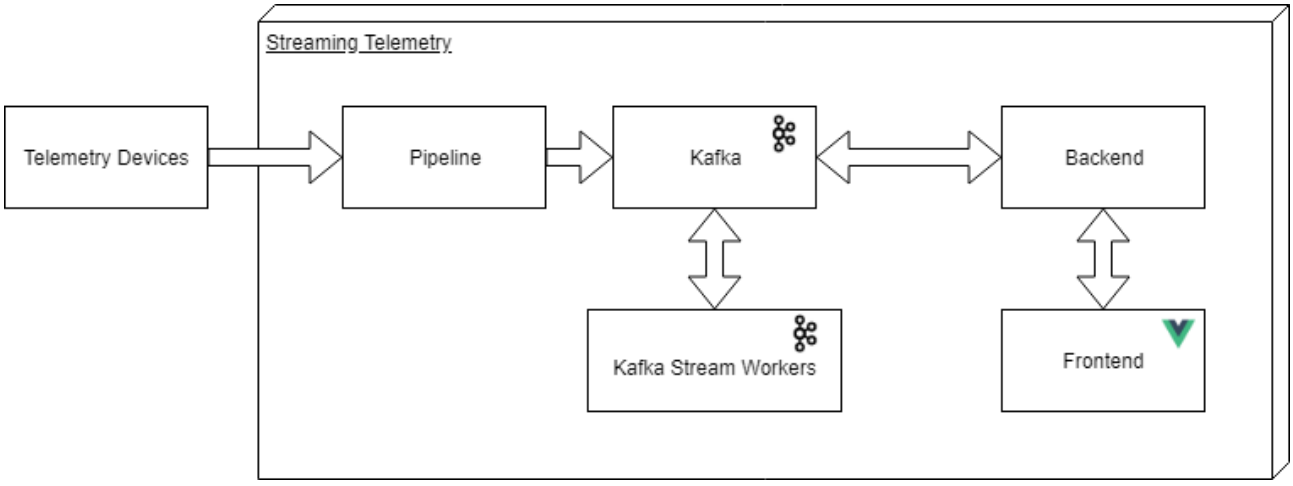

Figure 1: Architecture diagram

We use Apache Kafka not only as a message que but also as a data storage. All messages from the network are processed and distributed in corresponding topics.

This job is done by the streaming worker, which uses the Apache Kafka Streaming API to receive, process and produce messages.

After the data is 0processed, the backend can display the network. To do so, the Spring Boot framework is used. The backend provides an API to provide the data needed by the frontend.

The frontend is a single page web application which uses Vue.js as a framework. We use single file components, which makes it very easy to change or extend the frontend. To display the network as a graph, we use the D3.js library.

Furthermore, we use GitLab's CI to create Docker containers out of all the above mentioned services.

Result The result is a very scalable application which can be used to display very big networks that use segment routing.

Thanks to the stateless applications like the Pipeline or the Kafka Streaming Worker, which are available as Docker containers, one can easily start new instances if needed. Also the Kafka cluster is made for a lot of data, and is also very scalable. Of course the same is true for the frontend and backend.

Outlook While developing this application, we made sure that it is easily extendible.

This is beneficial because there is a lot of room for new functionalities. One could display more useful data per network node. Aggregating data for this, maybe to display a graph, can be done in the Streaming Worker.

Furthermore, we were not able to implement one optional requirement "trigger event based changes". The idea is to change the network when certain changes are detected. Apache Kafka streaming API has a lot of tools to detect changes and aggregate data. So one would need to implement a way to communicate back to the network, then define rules and aggregate data to check those rules. If they are broken, changes in the network could then be triggered.

Part IV

Technical Report

1 Problem Analysis

This section of the technical report deals with the problem the system has to solve.

1.1 Requirement Analysis

The requirements described below are categorized into MUST and CAN, whereas MUST is the highest priority and has to be fulfilled by the end of the project scope.

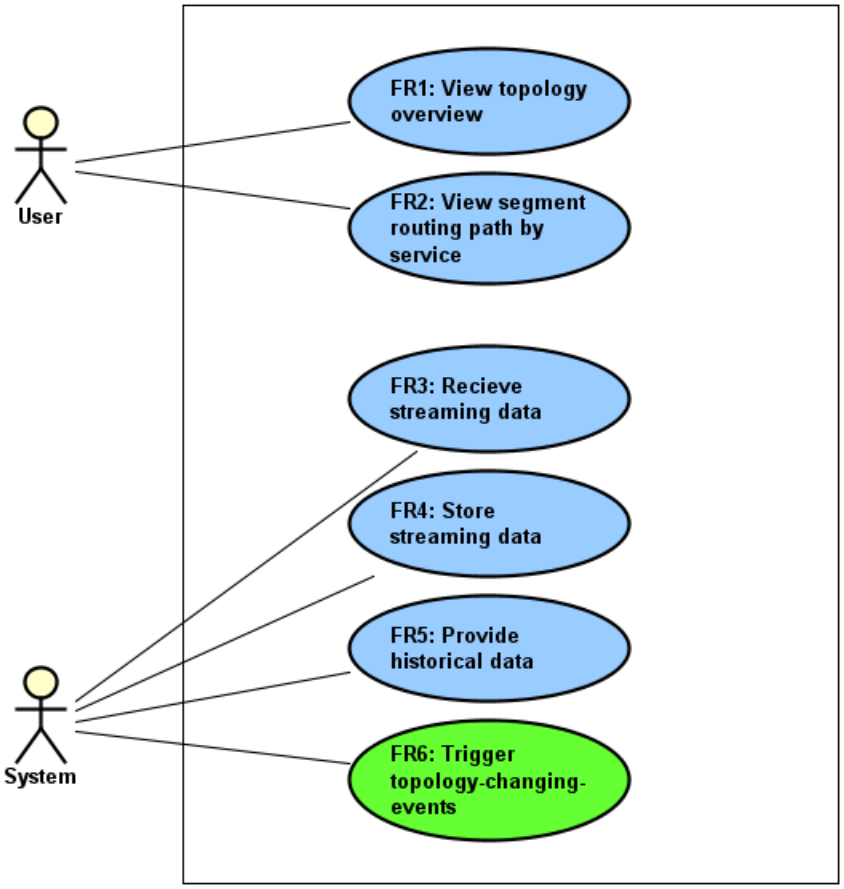

Figure 2: Function requirements model

Legend *Blue: Prio 1 | Green: Prio 2*

1.1.1 Actors

User	Interacts over the frontend with the system
System	Backend services

Table 1: Actors

1.1.2 Functional Requirements

ID	FR1
Title	View topology overview
Priority	MUST
Description	The user is able to view the topology overview in a graphical way.

Table 2: FR1 - View topology overview

ID	FR2
Title	View segment routing path by service
Priority	MUST
Description	A user selects the nodes and the service he wants to show the path by a specific time.

Table 3: FR2 - Select service to display

ID	FR3
Title	Receive Streaming Data
Priority	MUST
Description	The system receives streaming data of different routers.

Table 4: FR3 - Receive Streaming Data

ID	FR4
Title	Store streaming data
Priority	MUST
Description	The system stores the received streaming data in an efficient way.

Table 5: FR4 - Store Streaming Data

ID	FR5
Title	Provide historical data
Priority	MUST
Description	The system provides traceability of network changes within a history.

Table 6: FR5 - Provide historical data

ID	FR6
Title	Trigger topology-changing-events
Priority	CAN
Description	The system triggers events based on changes in the topology.

Table 7: FR6 - Trigger topology-changing-events

1.1.3 Non-Functional Requirements

ID	NFR1
Title	Scalability
Description	The system must be able to track 100 routers.
Measurability	Set up a system with 100 routers to check the scalability of the system

Table 8: NFR1 - Scalability

ID	NFR2
Title	Usability
Description	The system provides options to adjust the visualization of the topology in a self-explanatory way. In addition, there are usability hints for the user to give support.
Measurability	Usability tests

Table 9: NFR2 - Usability

ID	NFR3
Title	Performance
Description	The system should visualize the topology and the path of a selected service within 5 seconds.
Measurability	Performance tests

Table 10: NFR3 - Performance

1.2 Domain Analysis

The following part describes the domain analysis. It supports the understanding of the problem domain.

1.2.1 Domain Model

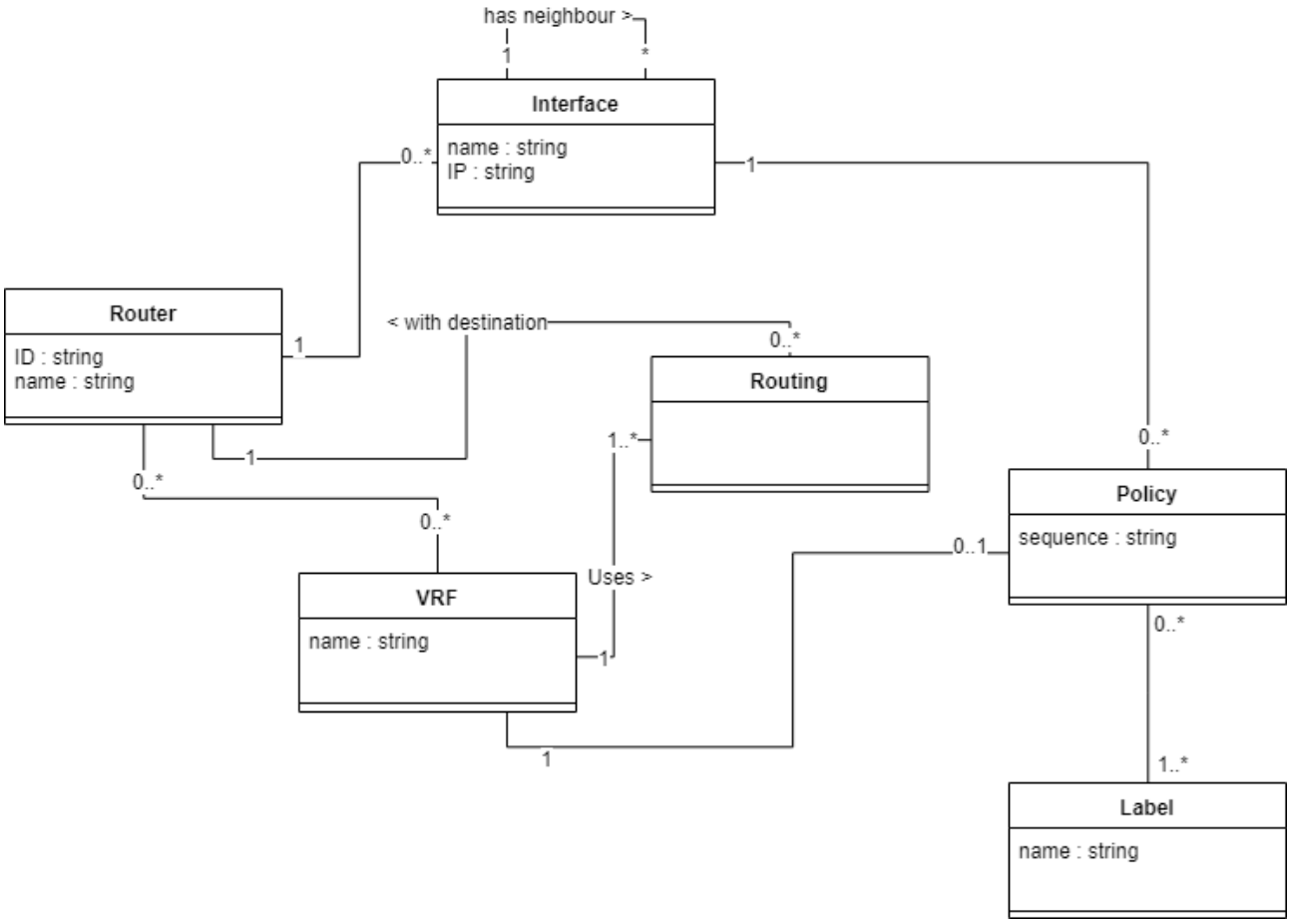

Figure 3: Domain model

1.2.2 Domain Description

Router

A router is a physical device, which should be displayed in the topology of the network.

Interface

The interface is the point of interconnection to another router. The interface defines what neighbours a router has.

VRF

Virtual routing and forwarding (VRF) is a technology that allows multiple customers to use the same router without interrupting each other.

Policy

The policy describes which route through the network a VRF takes to an other router.

Label

A label represents a segment in the segment routing path. They are provided by the IS-IS protocol.

Routing

The routing describes the path packets take when there is no policy defined.

2 Evaluation

During the evaluation phase in our project, we decided on which systems and technologies to use. In the following section we outline what alternatives we evaluated, and which decision we made.

2.1 Streaming data receiver

Cisco already developed a piece of software which parses telemetry data and pushes the packages to Kafka or databases. Because it was made by Cisco and was recommended to us by the supervisor of this thesis, we decided to use this Pipeline[1] for the system. Therefore, no further evaluation had to be made on this component.

The job of this pipeline is to receive data from the router, and then to forward it to one or multiple downstream consumers. Furthermore, it includes practical features for debugging and developing. For example, one can tap into the data that passes through the pipeline to see what is send by the routers.

Moreover, the pipeline can be easily scaled because it does not depend on other components. So one could deploy multiple instances, and then use a load balancer to distribute the load to multiple pipelines.

An other advantage of the cisco pipeline was the support for UDP, because in our meetings our supervisors decided to send the data over UDP to decrease the load on the network.

2.2 Database

The receiving data, that the system has to store, is a never-ending stream of data. So the system must have the ability to handle streams in an efficient way.

2.2.1 Apache Kafka

Apache Kafka is basically a message broker, which describes itself as follows: [2]

> Kafka® is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.

It is very fast and fault-tolerant and, more importantly, scalable to thousands of instances. The next thing we thought we needed was a database. We use Kafka to filter, transform and aggregate the data we get, but where do we put it? Also we needed to access the history of the data. How did the network look two months ago? We looked into different databases. We thought a graph database would make sense because we want to store a network graph. But we did not find an efficient way to deal with the history. An other option was a classic relational database. But then we stumbled over an article called

[1] `https://github.com/cisco/bigmuddy-network-telemetry-pipeline`
[2] https://kafka.apache.org/

"It's Okay To Store Data In Apache Kafka"[3] and we discovered that we could just leave the data on Kafka.

In the beginning we only wanted to use it for transforming and filtering the streams. But then we decided that we can also store our data directly on Kafka. Kafka is usually deployed as a cluster of many Kafka nodes, and the data is distributed between them. So the data is always redundantly stored.

A big advantage for us is also the integrated log compaction. If this is activated, Kafka automatically thins out old messages. This saves storage space, but is still retaining historical data. Although the resolution of this data will be less.

Historical data With the Stream API that Kafka provides, there is a function called "Windowing"[4], which allows the system to group the data into time windows with a specified time period (f.e. five minutes, one hour).

This makes it easy to use history data out of the telemetry.

2.3 Backend

The backend is responsible to offer a API for the frontend.

There are a lot of programming languages and frameworks which we could have used for the backend. We discussed Node.js and PHP. But based on our knowledge, we decided to use Java as the main programming language. This is because Java is one of the few languages that we are both very familiar with. And because Java is still very popular and is most likely still supported in the future.

Spring.io Because we knew we would have to consume Kafka streams, we looked for a framework that can natively consume them in an easy way. What we found was the Spring framework. There is a project called spring-kafka[5] which implements core functions to interact with a Kafka-Stream from a Spring application. However, we decided on that framework not only because of the easy integration with Kafka, but also because Spring-Boot makes it very easy and fast to develop a REST API.

Furthermore, Spring exists for 16 years (2002)[6], and is very well tested. Also Spring is very popular and well documented. This makes it easy for new developers, if it is ever decided to extend the application in the future.

[3] https://www.confluent.io/blog/okay-store-data-apache-kafka/
[4] https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html#windowing
[5] https://spring.io/projects/spring-kafka
[6] https://en.wikipedia.org/wiki/Spring_Framework

2.4 Frontend

2.4.1 Web server

The web server has to handle user requests and provide static files as a response.

Apache Apache HTTP Server is around for 23 years. It is very well tested and used by a lot of applications.

NGINX NGINX is a bit younger than apache (14 years old), but is also well tested and very powerful.

There were no real advantages or disadvantages from one over the other. In the end we decided to use NGINX, because it was very easy to use in an docker container, and needed very little configuration.

2.4.2 Frontend Application

Because we could display all the functionality of the app on one page, we wanted to create a single-page application. It was obvious that we had to use JavaScript for this. But we had the choice of a lot of JavaScript frameworks.

Vue.js Vue.js is a very intuitive framework. It is young, but very popular and used by many people. Therefore we could assume that the application is well supported. Also in Vue.js it is possible to develop components which are reusable.

AngularJS AngularJS is a bit longer on the market than Vue.js. It is also very popular and very well supported. AngularJS inspired the developers of Vue.js. But the API of AngularJS is a bit more complex.

In the end we decided to use Vue.js. This was because Matthias Dunkel was already familiar with it, and because the API of Vue.js is easier to learn. It also allows us to structure the application into components, which makes the application extendible, and components reusable.

2.4.3 Visualization

Because we need to visualize a graph, we wanted to use a pre made JavaScript library which can handle this task.

d3.js d3.js binds any desired data to the DOM of the web page, and applies data-driven transformations to the document. It is widely used, and well tested.

There are a lot of examples, including several topology and network graphs that can be reused and customized.

neXt UI neXt UI is a JavaScript/CSS toolkit centered around displaying networks, created by Cisco.

We decided to use d3.js. It is used by far more people, resulting in a well tested code and a lot of examples. Also it integrated very well with vue.js.

3 Conception and Design

One requirement of this project was the scalability, and that it should work with 100 routers. Additionally the telemetry data is potentially sent quite often (at least every five minutes).

Based on these requirements we decided to use different micro-services which can be scaled individually if more processing power is needed.

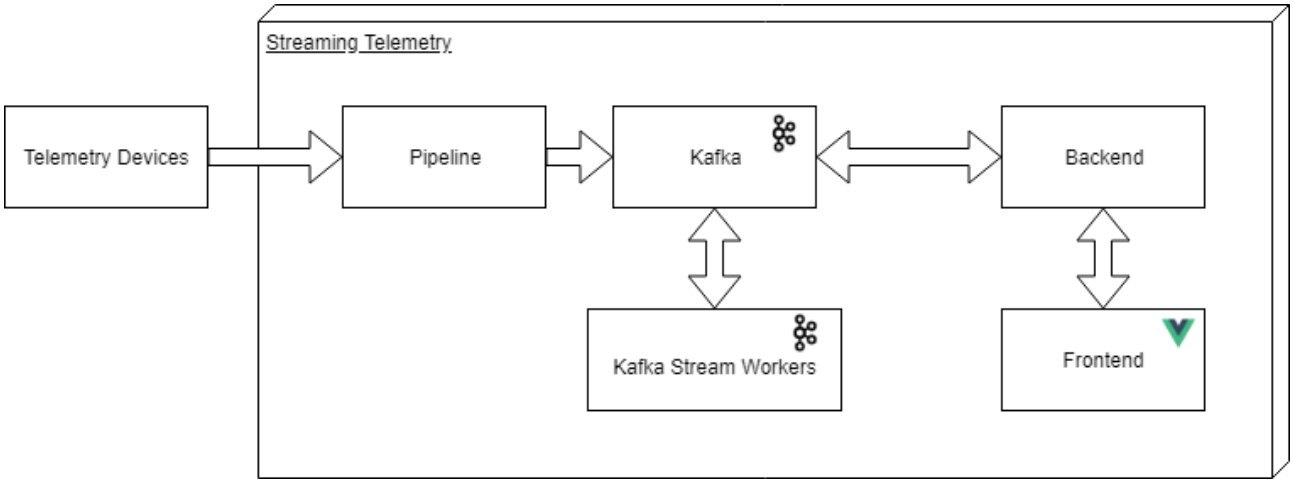

Figure 4: Architecture diagram

3.1 Pipeline

Pipeline is a piece of software developed by Cisco [7]. The use of this code is under free license. The Pipeline consumes directly the telemetry data from the routers. It could transform the data, but we use it to push the telemetry data directly and unmodified to a Apache Kafka topic named *telemetry*.

The pipeline is used within a Docker container, with a simple configuration file which manages the settings. This allows us to spin up multiple more instances if they are needed. Henceforth, the pipeline is very scalable.

3.1.1 Yang-Models

Yang-Models are defined on the routers and they determine what data is sent over streaming telemetry. The following Yang-Models[8] are used to get the relevant streaming data for the application:

Cisco-IOS-XR-ethernet-lldp-oper:lldp Get the network topology out of the LLDP-neighbours information

[7] https://github.com/cisco/bigmuddy-network-telemetry-pipeline
[8] https://github.com/YangModels/yang/tree/master/vendor/cisco/xr/651

openconfig-network-instance:network-instances Vendor independent information of VRF and the segment routing path.

Cisco-IOS-XR-fib-common-oper:mpls-forwarding Get the global routing table information

Cisco-IOS-XR-ipv4-io-oper:ipv4-network Get the different VRF information of every router

3.2 Apache Kafka

The system uses Docker to run the individual instances of Kafka. This allows the administrator to easily extend the Kafka cluster when needed.

A Kafka instance receives the streaming data from the pipeline over the topic *telemetry*.

3.2.1 Kafka Stream Workers

The Kafka Stream Workers component acts as a micro-service and uses the Stream API provided by Kafka to filter and aggregate the main stream of data into specific topics. The aggregation of data is needed, because the data packages from the pipeline are split and sent within multiple messages, which all share a common ID. Furthermore, this separation makes using the data later more efficient and easier.

See figure 5 for the dataflow diagram.

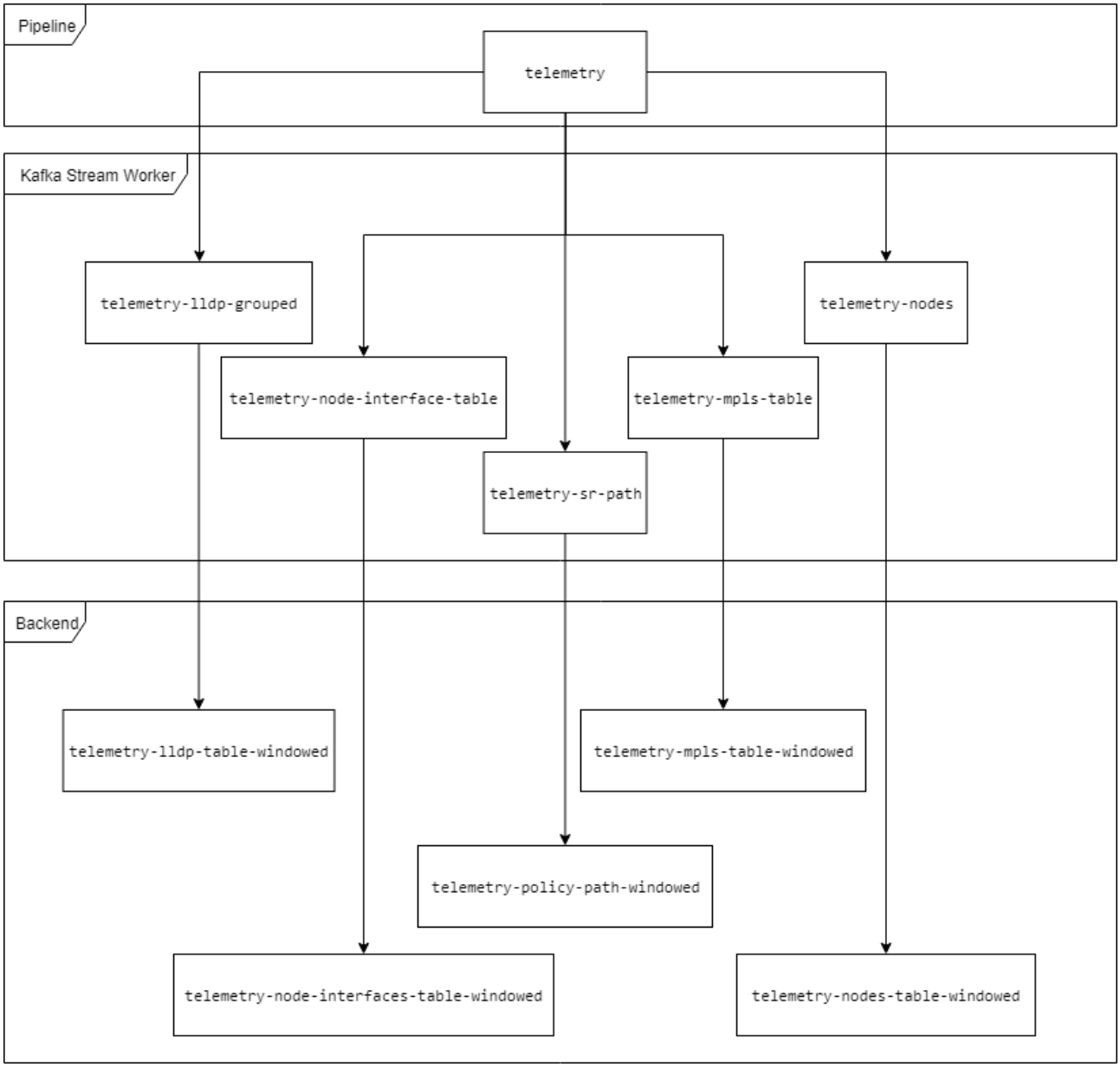

Figure 5: Dataflow between Kafka topics

3.2.2 Topics

These topics are produced by the Streaming Worker service:

telemetry-lldp-grouped This topic includes streaming data of the Yang Model Cisco-IOS-XR-ethernet-lldp-oper:lldp and contains data used to determine the neighbours of a node.

telemetry-nodes This topic includes streaming data of the Yang Model Cisco-IOS-XR-ethernet-lldp-oper:lldp and contains information about the nodes (routers).

telemetry-sr-path This topic includes streaming data of the Yang Model openconfig-network-instance:network-instances and contains the policy label stack.

telemetry-mpls-table This topic includes streaming data of the Yang Model Cisco-IOS-XR-fib-common-oper:mpls-forwarding and contains global MPLS routing table data.

telemetry-node-interface-table This topic includes streaming data of the Yang Model Cisco-IOS-XR-ipv4-io-oper:ipv4-network and contains the interfaces of each node.

3.3 Backend

To separate and aggregate the streaming data received in Kafka, and transform them for the needs of the frontend, there is a backend application. The backend application provides a RESTful interface for the frontend application to get the data.

3.3.1 Data transformation

The backend receives the data from Kafka via the Stream API (which also runs on the backend) and accesses the topics, which have been created by the Kafka Stream Workers. Every topic is windowed by 1 hour. For every hour, the newest record entry is taken and stored in a KTable. That allows the application to access the data by date and time.

A KTable (abstraction of the Kafka Stream itself) ensures that every record is always up-to-date. KTables always contain the newest record per key for a stream of data.

The topics created by the Kafka Stream Workers will be transformed as follows:

Nodes (Topic: telemetry-nodes-table-windowed) The node information (Name, Global IP Address, supported VRFs) is created as a KTable for every node.

Topology (Topic: telemetry-lldp-table-windowed) To display the topology, the topic *telemetry-nodes* will be transformed with the relevant neighbour information.

Global Routing Table (Topic: telemetry-mpls-table-windowed) The global routing table is needed to discover node labels on routers. They show the relations between nodes over these node labels.

Node Interfaces Table (Topic: telemetry-node-interfaces-table-windowed) Every node holds several physical and virtual interfaces. Any interface has a unique IP-address in the system, therefore each interface ip-address can be mapped to a specific node.

Policy Label Stack Table (Topic: telemetry-policy-path-windowed) Every path has its own policy on the departure node to get to a defined destination node over segment routing. This policy information is separated into node & adjadency labels.

To get to this label stack, there are three different paths involved on the openconfig-network-instance:network-instances sensor path. The way to get through these paths is as follows:

1.) Find a specific unique policy label.

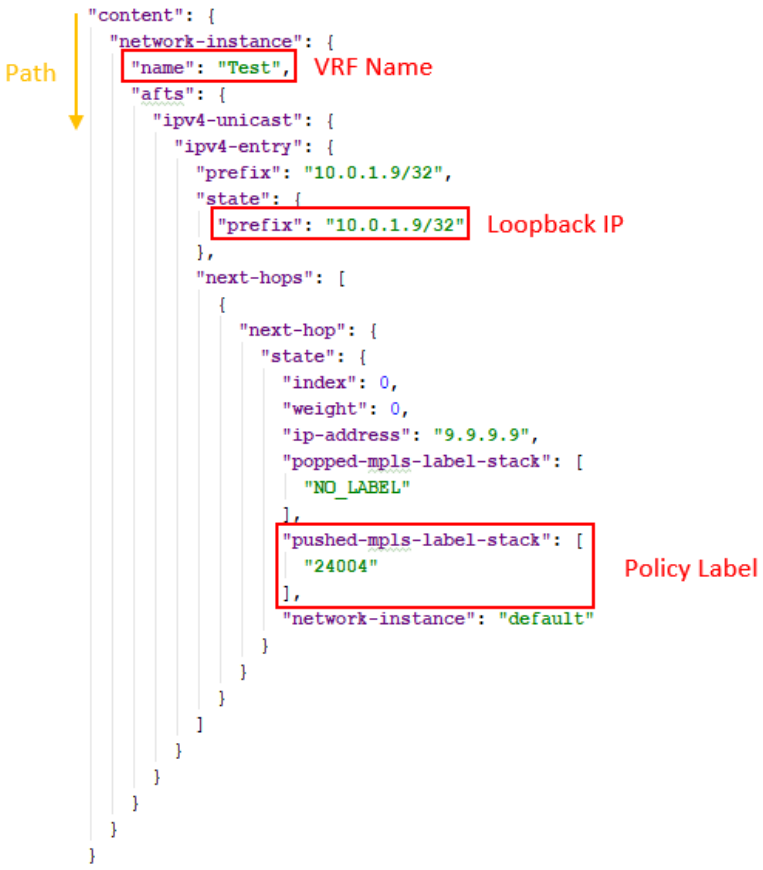

Figure 6: Find policy label

2.) Get the BGP tunnel interface with this policy label as key

Figure 7: Find tunnel interface

3.) Find the shortest label stack for this tunnel interface. The next-label-information-hop-string gives us a hint, which is the next node to follow.

Figure 8: Find label stack

With the logic described above, every configured policy in the network can be accessed.

3.3.2 Find Segment Routing Path

The goal is to find the segment routing path for a specific VRF.

Every segment is encoded as an MPLS label and the label stack is a ordered collection of segments, which creates the full segment routing path. There are two different types of MPLS labels we distinguish:

- 16xxx (Node label): This is a global node label. The information of the next node can be found in the Global Routing Table (Topic: telemetry-mpls-table-windowed). It can happen, that the following label isn't the label referenced on the node label. Then the whole process has to be repeated until the referenced node label has been reached.
- 24xxx (Adjacency Label): This is a local label for the adjacencies on the node itself. With a combination of current node name (e.g. XR-01) and adjacency label (e.g. 24002), concatenated as a key with a divider ":", the interface of the next node can be accessed. With this interface, it is possible to get the node name on the Node Interfaces Table (Topic: telemetry-node-interfaces-table-windowed)

With the help of the above-described created KTables it is possible to find the path as follows:

1. Start with the next hop interface as the first node of the path. The interface is a ip-address whose node can be found with the help of the Node Interfaces Table (Topic: telemetry-node-interfaces-table-windowed).
2. Process every label in the label stack received by the policy.
3. If the last visited node is the destination node, the path has been successfully created. Otherwise some router or segment routing configurations are missing.

Example The user has selected the departure node **XR-01** and wants to know, how the network traffic of the VRF ***Test*** is routed to the destination node **XR-09** at the moment.

Step 1: The system retrieves the following policy information based on the user selection:

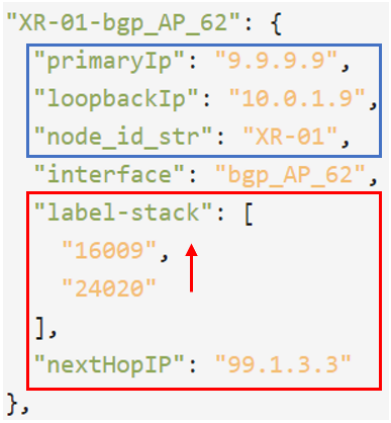

Figure 9: Path finding example: policy

The data in the blue bracket is to identify the exact needed policy. The *node_id_str* field is the departure node name, the *primaryIp* field is the global IP-Address of the destination node in the network and the loopbackIp is the loopback ip-address of the in the specific service.

The data in the red bracket is used to determine the path. The *nextHopIP* field delivers the next node of after the start and the label-stack includes a array of labels in reverse direction (read it from bottom to top).

Step 2: Get the node for the next hop IP of the policy received, in this example **99.1.3.3**. The information is stored in the Node Interfaces Table (Topic: telemetry-node-interfaces-table-windowed):

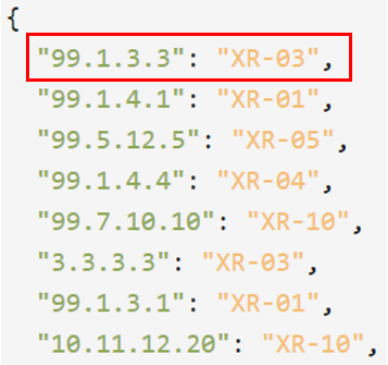

Figure 10: Path finding example: Find next hop interface node

We know that the path moves from our starting node XR-01 to node XR-03 and the process continues there.

Step 3: Process the label stack and start with the first label **24020**. This label is an adjacency label (label description can be found in the beginning of this section) which means, the node is directly connected over one of his own interfaces with this label. The system determines the interface name with the help of the Global Routing Table (Topic: telemetry-mpls-table-windowed):

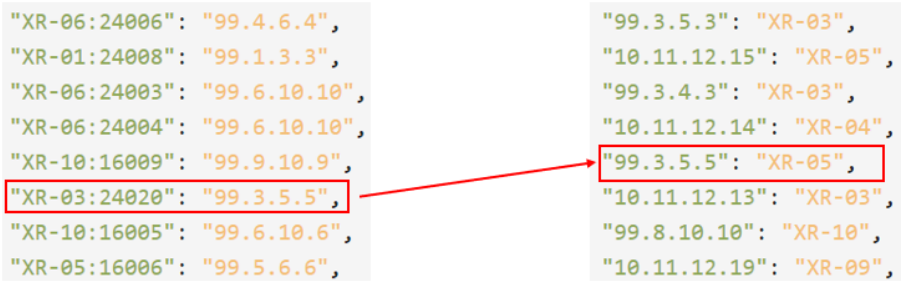

Figure 11: Path finding example: Process 24*** label

Step 4: Process the next label in the label stack, **16009**. This is a node label, therefore the system has to go over every connected node with the label attached, until no routing logic with this label can be found anymore (node with this global label is reached).

First the system finds the following entry on XR-05:

Figure 12: Path finding example: Process 16*** label on XR-05

Continue with the search on the discovered node XR-06:

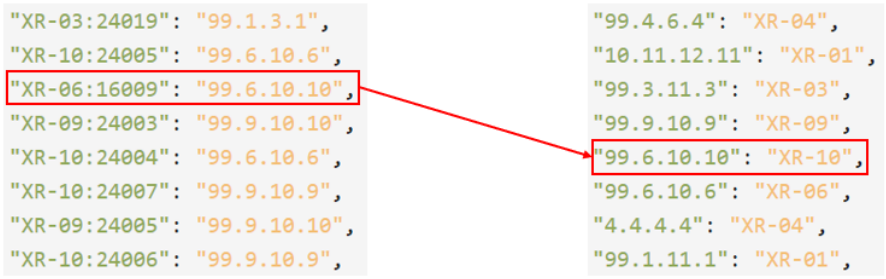

Figure 13: Path finding example: Process 16*** label on XR-06

Once more, continue on the new discovered node XR-10:

Figure 14: Path finding example: Process 16*** label on XR-10

No further entry can be found now with the XR-09 node. In conclusion, the global label has been reached.

Step 5: Since there are no further labels in the label stack, the destination node has been reached.

3.3.3 RESTful API

This is the description of the API the backend provides for the frontend.

Title: Get all nodes
URL: /nodes
Method: GET
URL Params:
time=[long]
(required)
The time as milliseconds since unix epoch, at which the state of the nodes should be replied.
Success Response:
Status: 200
Response:

```
{
    "XR-09": {
        "node_id_str": "XR-09",
        "systemID": "0000.0000.9999",
        "label": "16009",
        "vrfs": [
            "default",
            "VPN_FOO",
            "HSR-INS",
            "Test",
            "SA"
        ],
        "vrf-loopbacks": {
            "default": "9.9.9.9",
            "VPN_FOO": "172.16.2.1",
            "HSR-INS": "10.20.0.111",
            "Test": "10.0.1.9",
            "SA": "20.0.1.9"
        },
        "primaryIp": "9.9.9.9"
    },
    "XR-05": {
        "node_id_str": "XR-05",
        "systemID": "0000.0000.5555",
        "label": "16005",
        "vrfs": [
            "default",
            "HSR-INS",
            "VPN_BAA",
        ],
        "vrf-loopbacks": {
            "default": "99.5.6.5",
            "HSR-INS": "10.20.0.107",
            "VPN_BAA": "33.0.0.1"
        },
        "primaryIp": "5.5.5.5"
    }
}
```

Error Response: A Statuscode that is not 200.

Title: Get graph
URL: /historicalGraph
Method: GET
URL Params:
time=[long]
(required)
The time as milliseconds since unix epoch, at which the graph should be replied.
Success Response:
Status: 200
Response:

```
{
    "nodes": [
        {
            "name": "XR-01",
            "hasInfo": true
        },
        {
            "name": "XR-02",
            "hasInfo": true
        },
        {
            "name": "CE-01",
            "hasInfo": false
        }
    ],
    "links": [
        {
            "source": "XR-01",
            "target": "CE-01"
        },
        {
            "source": "XR-01",
            "target": "XR-02"
        },
        {
            "source": "XR-01",
            "target": "XR-03"
        }
    ]
}
```

Error Response: A Statuscode that is not 200.

Title: Get path
URL: /path
Method: GET
URL Params:
departureNodeName=[string]
(required)
The name of the node from where the path should be drawn.

destinationNodeName=[string]
(required)
The name of the node at which the path should end.

destinationGlobalIp=[string]
(required)
Primary Ip of destination node.

loopbackIp=[string]
(required)
Loopback IP of the vrf for which the path should be drawn.

time=[long]
(required)
The time as milliseconds since unix epoch, at which the path should be replied.

Success Response:
Status: 200
Response:

```
[
    {
        "source": "XR-01",
        "target": "XR-03"
    },
    {
        "source": "XR-03",
        "target": "XR-05"
    },
    {
        "source": "XR-05",
        "target": "XR-06"
    },
    {
        "source": "XR-06",
        "target": "XR-10"
    },
    {
        "source": "XR-10",
        "target": "XR-09"
    }
]
```

Error Response: A Statuscode that is not 200.

3.4 Frontend

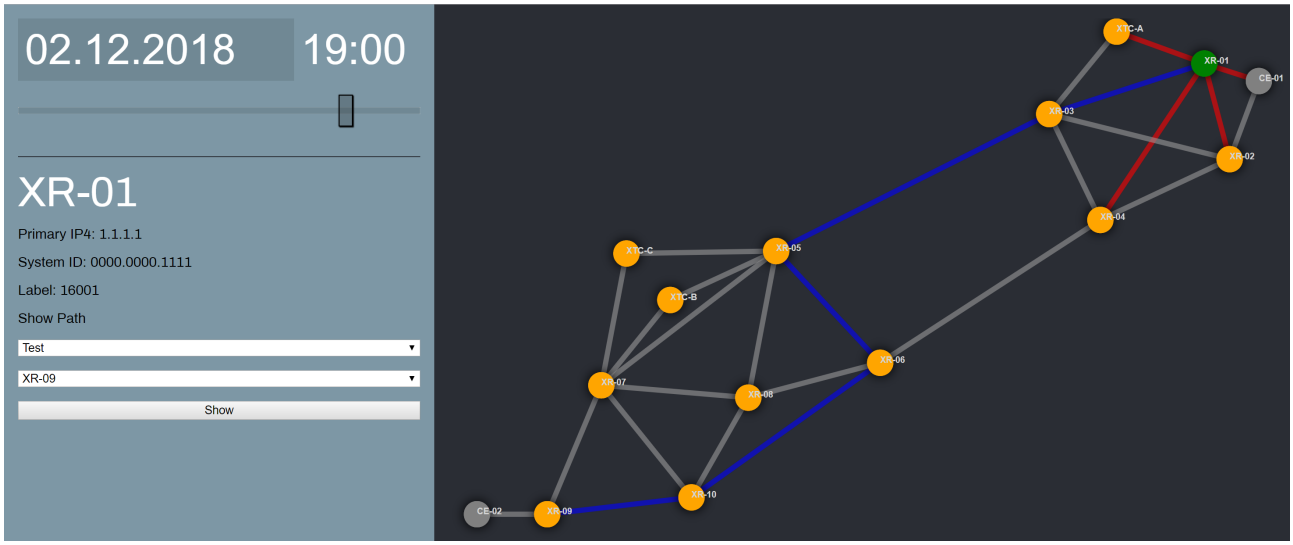

Figure 15: Frontend

We used a component based approach in the frontend. This means that the frontend consists of individual components. We used 4 of those components:

The App component is responsible of arranging the other three components. Those child components are marked with a red border in figure 16. Furthermore, it manages events from its child components, and makes API calls when appropriate.

In the top left there is a DateTimePicker component with which one can control the date and time at which the data should be displayed. On change, it communicates the new date and time to its parent.

The whole right part of the website contains the Topology component which displays the network graph. It contains all the logic to draw the graph. When clicking a node, the clicked node is communicated to it's parent. The parent (the App component) then displays the node information.

The NodeDetails component is responsible for displaying the node information for a single node. It also contains a form with which the user can choose which network path should be drawn.

All those last three components are completely separate. That results in the big advantage that they can easily be reused, in this project or an other. Moreover, they can be used multiple times in a project in case one would like to add new features that use the same functionality.

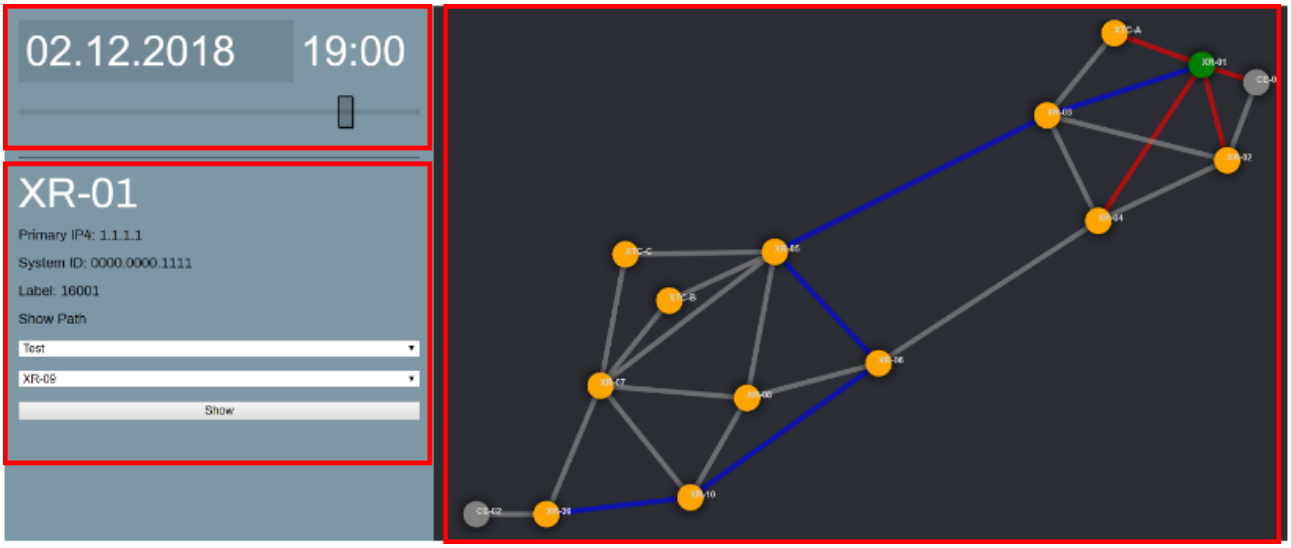

Figure 16: Frontend where Vue.js components are marked

3.5 Deployment & Installation

3.5.1 Continuous Integration

We use GitLab's Continuous Integration (CI) to build the projects every time when there is a push to the master branch. The CI then create a Docker container, and publishes it in the GitLab registry.

This has the advantage that one can always get the latest version as a Docker container. This makes deployment very easy.

3.5.2 Deployment

For easy deployment we decided to use Docker Compose, which describes all the containers and their configuration.

Before you can pull these containers, you need to log in to the GitLab registry. This has only to be done once on the host machine:

```
docker login registry.gitlab.com
```

Installation The containers will write their volumes to "/opt/data", so make sure this directory exists and is writable.

Then, once you have checked out the "sa-streaming-telemetry" project, you can start up the containers:

```
docker-compose up -d
```

Now make sure that the routers are sending the telemetry data to the host machine on port 5432 using UDP. To check if data is arriving you can look at the GUI of this application. Or directly tap into the Kafka topic to look at the raw data, by using the commands described later on.

Update To update the containers with the new version from the registry, execute either the 'update.sh' file, or run:

```
docker-compose rm -s -f
docker-compose pull
docker-compose up -d
```

If you would like to update just one container, just append its name to the commands:

```
docker-compose rm -s -f sa-backend
docker-compose pull sa-backend
docker-compose up -d sa-backend
```

3.5.3 Maintenance

Debugging There are a few practical commands that can be used to debug the application. To **inspect the data from a Kafka topic** (in this example the topic "telemetry"), run this command on the host machine:

```
clear && docker run \
--net=sa-streaming-telemetry_default \
--rm confluentinc/cp-kafka:5.0.0 \
kafka-console-consumer --bootstrap-server kafka1:29092 --topic telemetry \
--property print.key=true --property print.timestamp=true
```

This command only show new incoming messages. If you want to **look at older messages**, you first have to get the current offset:

```
docker run \
--net=sa-streaming-telemetry_default \
--rm confluentinc/cp-kafka:5.0.0 \
kafka-run-class kafka.tools.GetOffsetShell \
--broker-list kafka1:29092 --topic telemetry
```

This will give you a number after the second column, like this: telemetry:0:10725705

The offset here is 10725705, now just subtract how many messages back you wanna go, and use this new number in this command (here I used 10725000 to go 705 messages back):

```
  docker run \
--net=sa-streaming-telemetry_default \
--rm confluentinc/cp-kafka:5.0.0 \
kafka-console-consumer --bootstrap-server kafka1:29092 --topic telemetry --partition 0 \
--offset 10725000 --property print.key=true --property print.timestamp=true
```

To **check the configuration** of a Kafka topic, use:

```
docker run \
--net=sa-streaming-telemetry_default \
--rm confluentinc/cp-kafka:5.0.0 \
kafka-configs --zookeeper zookeeper:32181 --entity-type topics \
--entity-name telemetry --describe
```

3.6 Testing

3.6.1 System Tests

The system consists of several different components that are working with each other. This collaboration must work to provide a visualization for the user at the end. The communication between the individual components and the correct execution of their tasks is tested in this section.

Pipeline This test uses a dump file to analyze whether any data from the routers reaches the pipeline.

ID	**ST1 - System receives telemetry data**
Test objective	The system receives telemetry data via the Pipeline
Preconditions	• Pipeline-Docker is set up
Execution	1. Start docker service Pipeline 2. Track the dump-file: `tail -f dumpFile.txt`
Expection	Data in dump file existing and file enlarges over time
Result	**Fulfilled**

Table 11: ST1 - System receives telemetry data

Kafka This test cases are responsible for the connection of Kafka with the Pipeline itself and to test the filter functions.

The commands described in section **Maintenance** can be used to check the topics data.

Name	**ST2 - Kafka receives Pipeline data**
Test objective	Kafka receives telemetry data over the Pipeline
Preconditions	• ST1 sucessfully executed • Kafka-Docker is set up (Zookeeper as well)
Execution	1. Start docker services Zookeeper & Kafka 2. Run maintenance command "Debugging" with topic *telemetry*
Expection	Topic *telemetry* receives data over time
Result	**Fulfilled**

Table 12: ST2 - Kafka receives Pipeline data

Name	**ST3 - Kafka-Stream-Workers running**
Test objective	Kafka-Stream-Workers filtering packets and creating new topics
Preconditions	• ST2 sucessfully executed • Stream-Workers-Docker set up as customized
Execution	1. Start docker services Stream-Workers
Expection	Docker Kafka-Stream-Workers running and consuming Kafka data-stream
Result	**Fulfilled**

Table 13: ST3 - Kafka-Stream-Workers running

Name	**ST4 - Filter LLDP packets**
Test objective	Kafka-Stream-Workers filtering LLDP packets into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Start docker services Stream-Workers 2. Run maintenance command "Debugging" with topic *telemetry-lldp-grouped*
Expection	Topic *telemetry-lldp-grouped* created and receives data over time
Result	**Fulfilled**

Table 14: ST4 - Kafka-Stream-Workers filtering LLDP packets

Name	**ST5 - Filter node information packets**
Test objective	Kafka-Stream-Workers filtering node information packets into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-node-table*
Expection	Topic *telemetry-node-table* created and receives data over time
Result	**Fulfilled**

Table 15: ST5 - Filter node information packets

Name	**ST6 - Filter segment routing policy packets**
Test objective	Kafka-Stream-Workers filtering segment routing policy packets into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-sr-path*
Expection	Topic *telemetry-sr-path* created and receives data over time
Result	**Fulfilled**

Table 16: ST6 - Filter segment routing policy packets

Name	**ST7 - Filter global routing table packets**
Test objective	Kafka-Stream-Workers filtering global routing table patckets into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-mpls-table*
Expection	Topic *telemetry-mpls-table* created and receives data from time to time
Result	**Fulfilled**

Table 17: ST7 - Filter global routing table packages

Name	**ST8 - Filter node interfaces packets**
Test objective	Kafka-Stream-Workers filtering packets information about node interfaces into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-node-interface-table*
Expection	Topic *telemetry-node-interface-table* created and receives data over time
Result	**Fulfilled**

Table 18: ST8 - Filter node interfaces packages

Backend These tests are important to ensure that the data is made available to the frontend and that historical data storage is introduced.

Name	**ST9 - Backend consuming Kafka data**
Test objective	Backend running and consuming Kafka data
Preconditions	• ST2 sucessfully executed
Execution	1. Start docker service SA-Backend
Expection	Docker service SA-Backend is up and running
Result	**Fulfilled**

Table 19: ST9 - Backend consuming Kafka data

Name	**ST10 - Window LLDP-Packages**
Test objective	Backend windowing topic *telemetry-lldp-grouped*
Preconditions	• ST3 sucessfully executed • ST9 sucessfully
Execution	1. Run maintenance command "Debugging" with topic *telemetry-lldp-table-windowed*
Expection	Topic *telemetry-lldp-table-windowed* created and receives data over time
Result	**Fulfilled**

Table 20: ST10 - Window LLDP packets

Name	**ST11 - Window node information packets**
Test objective	Backend windowing topic *telemetry-nodes-table* into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-node-table-windowed*
Expection	Topic *telemetry-nodes-table-windowed* created and receives data over time
Result	**Fulfilled**

Table 21: ST11 - Window node information packets

Name	**ST12 - Windowing segment routing policy packets**
Test objective	Backend windowing topic *telemetry-sr-path* into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-policy-path-windowed*
Expection	Topic *telemetry-policy-path-windowed* created and receives data over time
Result	**Fulfilled**

Table 22: ST12 - Window segment routing policy packets

Name	**ST13 - Window global routing table packets**
Test objective	Backend windowing topic *telemetry-mpls-table* into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-mpls-table-windowed*
Expection	Topic *telemetry-mpls-table-windowed* created and receives data over time
Result	**Fulfilled**

Table 23: ST13 - Window global routing table packets

Name	**ST14 - Window node interfaces packets**
Test objective	Backend windowing topic *telemetry-node-interface-table* into new topic
Preconditions	• ST3 sucessfully executed
Execution	1. Run maintenance command "Debugging" with topic *telemetry-node-interface-table-windowed*
Expection	Topic *telemetry-node-interface-table-windowed* created and receives data over time
Result	**Fulfilled**

Table 24: ST14 - Window node interfaces packets

Frontend These tests are to ensure, that the frontend is accessible and usable for the user.

Name	**ST15 - Frontend running and usable**
Test objective	Frontend running and accessible
Preconditions	• ST9 sucessfully executed
Execution	1. Start docker service SA-Frontend 2. Open web-browser and call ip-address, where the Docker Service are hosted
Expection	Docker service SA-Frontend is running and web page can be opened
Result	**Fulfilled**

Table 25: ST15 - Frontend running and usable

Name	**ST16 - Frontend displays topology**
Test objective	The user can view a topology with node information
Preconditions	• ST10 sucessfully executed
Execution	1. Open web-browser and call ip-address, where the Docker-Services are hosted
Expection	A network topology is displayed
Result	**Fulfilled**

Table 26: ST16 - Frontend displays topology

Name	**ST17 - Frontend displays historical topology**
Test objective	The user can adjust the date and time to display historical topologies and paths
Preconditions	• ST10 sucessfully executed
Execution	1. Open web-browser and call ip-address, where the Docker Service are hosted 2. Use the date-time text-box or the slider to change the date and time of the topology
Expection	A network topology is always displayed. In case of network changes the viewed topology gets adjusted
Result	**Fulfilled**

Table 27: ST17 - Frontend displays historical topology

3.6.2 Performance Test

This test is to ensure, that the quality requirement of the frontend performance can be fulfilled.

Name	**PT1 - Frontend displays topology within the specified time**
Test objective	The frontend displays the topology within 5 seconds after start.
Preconditions	• System is set up and running
Execution	1. Open web-browser and call ip-address, where the Docker Service are hosted
Expection	A network topology is displayed within 5 seconds.
Result	**Fulfilled**

Table 28: PT1 - Frontend displays topology within 5 seconds

4 Results & Conclusion

4.1 Target achievement

The developed components meet the requirements expected by the customer.

The following points were achieved:

- Reception and storage of streaming data over Pipeline
- Visualization of the network topology
- Visualization of the segment routing path between two routers
- The ability of historical data tracking within a topology or a segment routing path to detect changes

Unfortunately there was no time left to implement the optional requirements:

- Trigger events in case of a detected change

On the basis of this work, the customer can easily extend the application according to his requirements, because we made sure that the components are ready for this and well separated.

4.2 Team retrospective

In the following section we would like to discuss the problems we had during this project and how we solved it.

To begin with, we are satisfied with the achieved result, as we both immersed ourselves in a new world and have never worked with the processing of streaming data or segment routing. Therefore, we had to deal with several difficulties during this project.

The specification of the architecture turned out to be the biggest hurdle. Without the necessary prior knowledge, it is difficult to find the most suitable components to solve the problem. The project managers were able to support us with their knowledge and recommend the best tools from their experience with data streaming processing.

Once the architecture was evaluated, there were troubles with the integration of the Kafka component into our solution. First, we had to understand, how Kafka is installed within a docker container, how it can receive the data, what a topic is, how they can be created and how we can filter the data received by the system. Fortunately, there is a documentation[9] which helped us a lot.

Another problem was the space consumption of the Pipeline and Kafka docker services on the server. The Docker file system was initially full for inexplicable reasons, interfering with the execution of Docker services and resulted in not processing any more streaming data. The analysis of this problem

[9] `http://kafka.apache.org/documentation.html`

showed that the volume provided by the Docker file system was not large enough. In addition, the pipeline created a dump file of all incoming data from the network devices, which can be used for debugging or data analysis purposes. But it filled the disk space very fast. With the help of an additional, larger volume (capacity: 200 GB) we mounted the data volumes of the Docker Services to this drive. Additionally, we terminated the creation of the pipeline dump file.

In the end it turned out that we suddenly weren't able to display the path in the network any more. That was very frustrating because we wanted to show that it works in our final presentation. Because we did not change our code, it had to be the data we received. So we analyzed it, and wrote a document outlining which data is missing and what does not match anymore.

To conclude, we had some difficulties but we managed to solve all of them and learned a lot in the process. And we would happily do it again.

5 Outlook

5.1 Trigger event based changes

Unfortunately, we did not have enough time to implement this optional requirement.

The idea is to change the network when certain changes are detected. Apache Kafka streaming API has a lot of tools to detect changes and aggregate data, so this feature can be implemented in little time in the future when needed.

So one would need to implement a way to communicate back to the network, then define rules and aggregate data to check those rules. If they are broken, changes in the network can be triggered.

5.2 Extension of the architecture

To ensure the stability of the architecture in case the network grows, there are a couple of points to introduce:

Multiple Kafka Producers In case there are more than 100 routers and the processing of the streaming data need more capability, there is the possibility to start more instances of the Kafka-Streaming-Workers and therefore have multiple Kafka workers at one time.

Backend with Load Balancer At the moment it is assumed that only a few users access the application at the same time. In the event of an increase, the system should still be performant. As a bottleneck, the backend would be overwhelmed by the number of requests and slow down the system accordingly. As a solution, you can aim for load balancing on several backend instances. However, you need to synchronize the data that the backend instances process for Apache Kafka.

Change historical data granularity The ability to track historical data is one strength of the application. There is a one hour window created for every topic. Over time, i.e. over several months or years, this implementation logic leads to a high demand for data storage and could lead to a capacity problem. It would be possible to combine individual windows, which are further back in time, no longer hourly, but daily into one window. To do this, a worker would have to be implemented that would combine all windows of one day into one separate window.

5.3 Frontend extensions

The topology is currently based on the width of the topology component and the screen size. For a smaller number of nodes, the network graph is clearly understandable. But a network with a lot network devices may look a bit overloaded. Therefore the following points could present a solution:

- Dynamically fade out the settings component and enlarge the topology component when needed
- Adding the ability to zoom the graph
- Adding the ability to move the nodes within the component

Additionally, the information shown per router or a path could be extended, e.g. by next hop time, tunnel interface name, processed label names and many more.

5.4 Security

The project has not yet addressed the issue of security.

On the one hand, access to the application itself could be restricted. This can be achieved with the help of a login with user name and password or a connection to an external login service (e.g. Cisco login).

Another important point would be the integration of security on Apache Kafka. By default, Kafka is used without any security mechanisms. This means that any consumer who has the Kafka Docker network information could read or write data and manipulate them.

Kafka Security provides the following three components[10]:

Data encryption using SSL/TLS The data flow between Kafka and its producers and consumers can be encrypted.

Authentication Client authentication can be used to restrict access to Kafka to individual applications.

Authorization with Access Control Lists (ACL) With the help of Access Control List you can distribute read & write permissions to the individual authenticated applications. These rights are set to individual Kafka topics.

[10] `https://kafka.apache.org/10/documentation/streams/developer-guide/security.html`

List of Figures

List of Tables

Acronyms

INS Institute for Networked Solutions. 4

IS-IS Intermediate System to Intermediate System. 15

LLDP Link Layer Discovery Protocol. 4, 20

MPLS Multiprotocol Label Switching. 23, 27

SA Studienarbeit. 4

SNMP Simple Network Management Protocol. 4

VRF Virtual routing and forwarding. 15

Glossary

Apache Kafka topic Is a stream of messages. A Kafka cluster can have multiple topics, which a distinguished by name. 20

Yang Is a data modeling language. 4, 5, 22, 23, 56

Part V

Attachments

A Personal reflection

A.1 Matthias Dunkel

In our first meeting we realized that we had the wrong knowledge about the task that was given to us. We never applied to this project, because we wanted something that was focused on software developing. And this project was intended to be about segment routing, which is network technology. Additionally we both had spent very little time on this in our studies.

Luckily, we could mention that in our first meeting when we realized what the project is about. Our expert reacted very positively, and promised to create a new assignment which is focused on software developing. Consequently we got an assignment that we are very happy about. The result was that we had to start two weeks later than expected, but we made the best out of it.

I found it very interesting to work with people who specialize in network technologies, and I got to see a part of informatics that I was not very familiar with.

The most interesting part for me was to learn about Apache Kafka and using it in this project. I had heard about it before, but never knew what it does. Now I see use cases for it everywhere.

The construction phase was also very interesting. Although in the assignment it was specified that the focus was on the frontend, we spent most of the time developing the backend and the streaming worker. This because aggregating the data was not easy. But I loved to do this kind of work, so that was no issue at all for me. Because this was something new, even our expert and his team did not know exactly which data we were going to need, so we had meetings every week discussing what we needed and how to interpret the data. This was a very good experience, and the meetings were always very constructive.

Additionally, I was very happy about the teamwork with my team partner Raffael Vögeli. We are both on the same technical level, so discussions were always very constructive and efficient. Furthermore, Raffael always saw the big picture, and was excellent in planning this project.

In conclusion, I am very proud of the work we did. I think we created a great product for the time we had, and also made sure it is very scalable. I think we learned a lot, and are now very well prepared for the bachelor thesis next semester.

A.2 Raffael Vögeli

The start into the student research project proved to be quite bumpy for me. The work was assigned to us due to an error in the HSR internal job administration tool (AVT). We did not have the necessary knowledge for the original task, and we immediately admitted that at the beginning of the work.

However, the supervisor of the work, Prof. Laurent Metzger, was not aware of this and was surprised by it himself. Nevertheless, together with us and the industry partner, he was able to react flexibly and define a new task, which was more tailored to our abilities. This also greatly increased my motivation to successfully complete the project, which we could start with a delay of 2 weeks.

The elaboration phase with the evaluation of the individual architecture components represented the greatest difficulty for me. I had never worked with streaming data and its processing before and

therefore I found the familiarization with these components (Yang Models, Pipeline, Apache Kafka) very time-consuming and difficult.

The cooperation with the project supervisors was very constructive and we received great help with any problems that arose. The weekly meetings provided us with the information we needed to move the project forward. It also resulted in constructive discussions which could be considered useful for both sides.

I have already successfully worked alongside my project partner throughout my studies. Therefore I did not expect anything to the contrary for this project. Throughout the whole project, problems arose again and again, which we had to clear up, among other things, in the team. With the help of pair programming, we were able to identify and solve these problems quickly. We complemented each other with our individual know-how and both the motivation and incentive to work on this project were always at a high level. The communication and the distribution of the project also worked excellently throughout the entire project.

As a final conclusion, I can speak of a very successful and instructive work and am already looking forward to the bachelor thesis, which I will do again with Matthias Dunkel.

B Used time

In this section we analyse the time spent on this project.

In the first figure 17 we can see that we both spent almost the same time. Some weeks one person worked more, but we always made sure that the other person then did something more in the following weeks. This happened sometimes because it was easier when one person developed or learned something alone and then briefed the partner. For example when we had to learn how Apache Kafka worked.

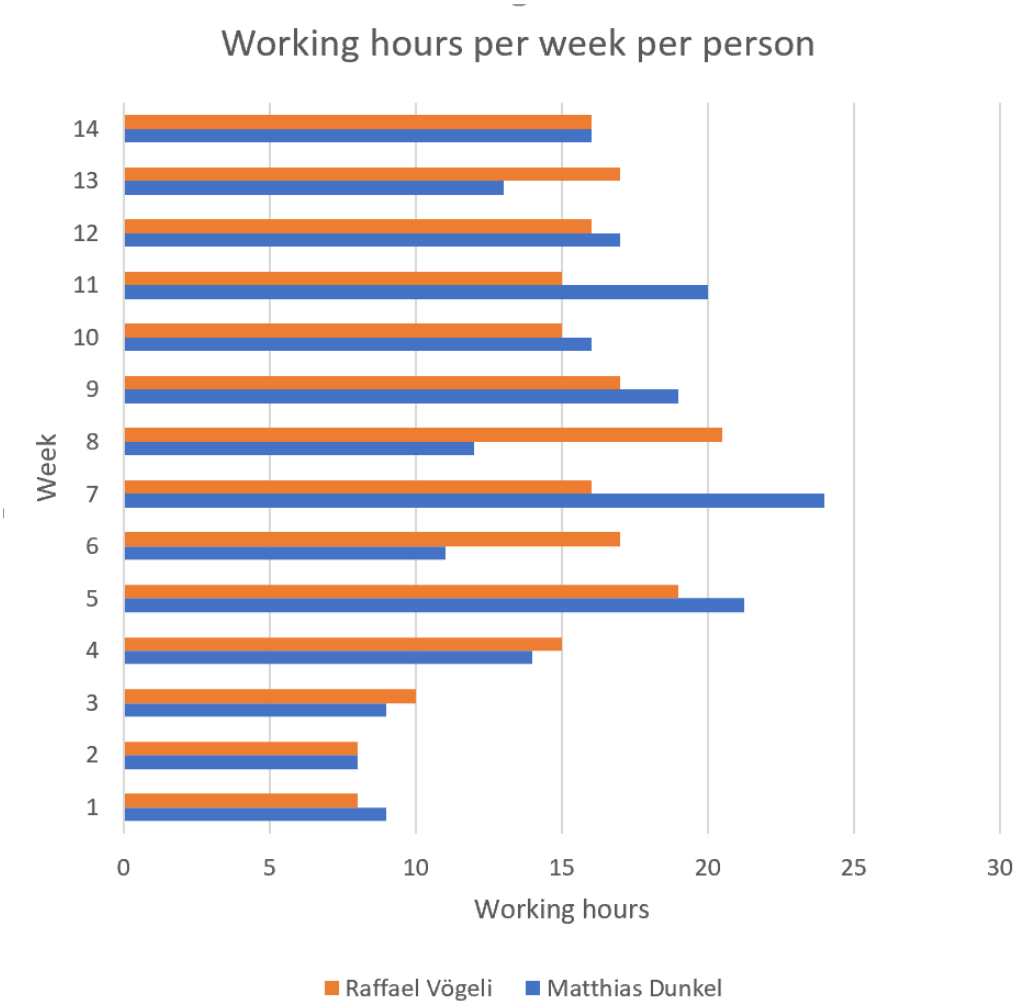

Figure 17: Working hours per week per person

Figure 18 shows what we expected. Most time was spent in the construction phase, where we developed the application and wrote this report.

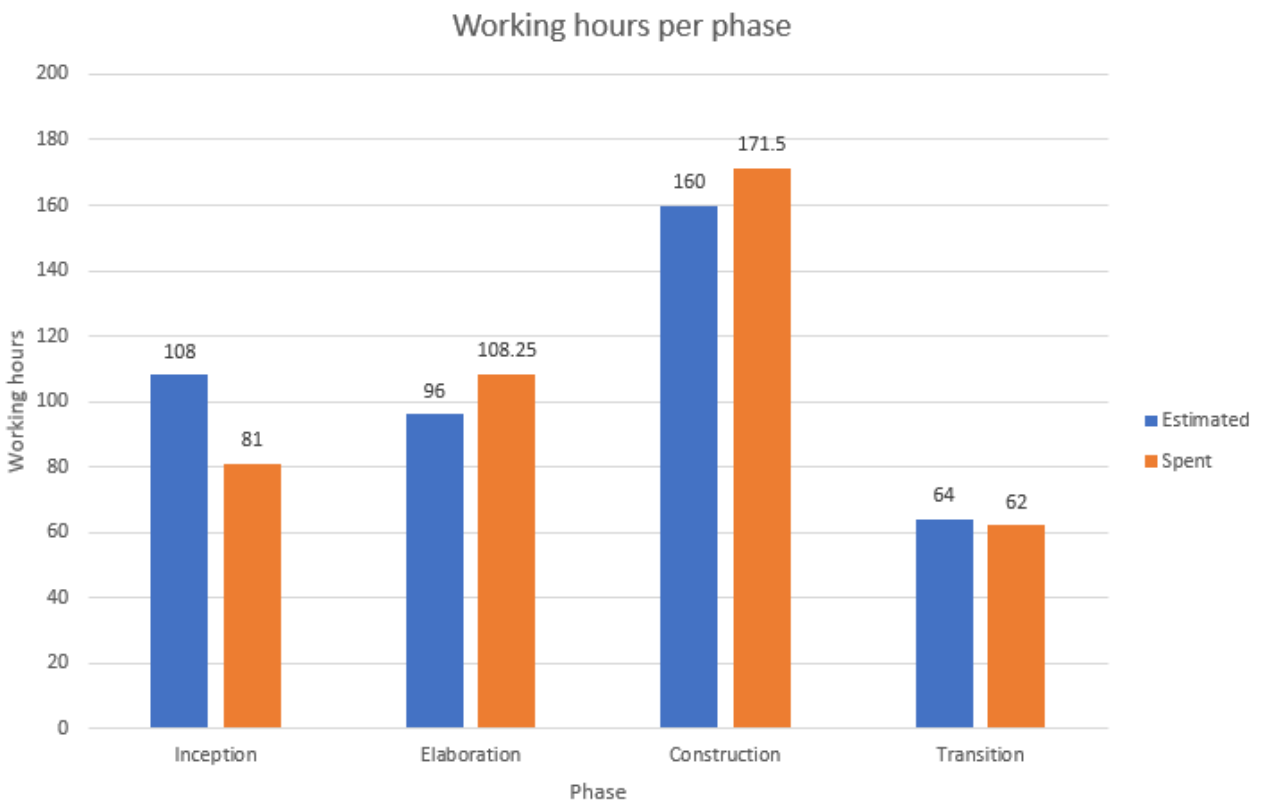

Figure 18: Working hours per phase

When we sum the time per person we get around 211 hours spent in total. This is a little bit under the amount that was forseen for this project. This is due to the fact that we received the assignment later than expected. We still managed to do some work without the assignment, such as setting up the documentation tools, discussing technologies and reading about them.

In this regard, we spent even more in the elaboration and construction phase, to get the project done within the estimated time.