University of South-Eastern Norway

www.usn.no

FM4017 Project 2020

Test system for testing ITS-services over 4G cellular network

MP-10-20

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FM4017 Project, 2020

Title: Test system for testing ITS-services over 4G cellular network

This report forms part of the basis for assessing the student's performance in the course.

Project group: MP-10-20 **Availability:** Open

Group participants: Alexander Svindseth

Supervisors: Hans-Petter Halvorsen

Tomas Levin

Project partner: Norwegian Public Road Administration

Summary:

In 2018 approximately 1.35 million people was killed in traffic, that is 3700 people a day[1]. Many of these deaths can be avoided by standard means like seat belts and motorcycle helmets, but for the last percentage of accidents, the vehicle driver needs smarter safety systems built into the vehicle. To implement these intelligent systems, much can be achieved with sensors in the vehicles. But even more can be achieved if the vehicles communicate and cooperates. The services needed for this is under development and standardization, and during these phases a test system is needed in order to assess these services. This report outlines the analysis, development and operation of a system that is a capable to implement a system for testing ITS-services over 4G. The cloud part of the system is based on microservice architecture, running in a Kubernetes cluster, with Elasticsearch for log handling and dedicated containers for each service. An onboard unit for a vehicle that communicates with the cloud system is made with a Raspberry Pi, 4G router and GPS.

Preface

This report has been written during the fall semester in 2020 at University of South-Eastern Norway in the course FM 4017 Project. The task was formulated together with Tomas Levin, who also has been supervisor from NPRA for this project. A big thanks to Tomas for the help and support, and also a big thanks to our common colleague, Christian Berg Skjetne. I would thank Hans-Petter Halvorsen which have been supervisor from the university.

A reader of this report should have interest in software architecture and preferably experience with container-based software.

Bergen, 19.11.202

Alexander Svindseth

Contents

Nomenclature

4G	Fourth generation cellular network technology
AMQP	Advanced Message Queue Protocol
API	Application Programming Interface
CSV	Comma Separated Values
C-ITS	Cooperative Intelligent Transport System
CCAM	Cooperative, Connected and Automated Mobility
CI	Continuous Integration
CD	Continuous Delivery
Day 1 services	ITS-services that is to be deployed today or in relative near future
Day 1.5 services	ITS-services that is to be deployed
DNS	Dynamic Name Service
GB	GigaByte
GHz	Gigahertz
GCP	Google Cloud Platform
GPS	Global Positioning System
GUI	Graphical User Interface
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Enviroment
IOT	Internet of Things
IP	Internet Protocol
ITS	Intelligent Transport System
JSON	JavaScript Object Notation

MQTT	Message Queuing Telemetry Transport
NMEA	National Marine Electronics Association
NPRA	Norwegian Public Road Administration (in Norwegian: Statens vegvesen)
NTP	Network Time Protocol
OEM	Original Equipment Manufacturer
OS	Operating System
PKI	Public Key Infrastructure
REST	Representational state transfer
RPC	Remote Procedure Call
RSRP	Reference Signal Received Power
RSRQ	Reference Signal Received Quality
SINR	Signal to Interference & Noise Ratio
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UML	Unified Modelling Language
USB	Universal Serial Bus
URL	Uniform Resource Locator
V2I	Vehicle to Infrastructure
V2V	Vehicle to Vehicle
V2X	Vehicle to everything
vCPU	Virtual Central Processing Unit
YAML	a recursive acronym for "YAML Ain't Markup Language"

1 Introduction

1.1 Background

The vehicles of today are trending from being individual vehicles towards becoming connected and cooperative with other road users, owners and authorities. The most modern vehicles of today, is able to perform simple autonomous driving based on sensors installed in the car. To be able to automate the driving even more, it is commonly acknowledged that the vehicles can't rely on onboard sensors only, but also needs to communicate with the road infrastructure and other road users. It is expected that this will contribute to safer and more efficient mobility solutions [2], and will accordingly be a key for reaching NPRAs vision zero[1]. When vehicles are connected and cooperative, they will be part of a traffic system that is known under the acronym CCAM.

NPRA is contributing to the development of CCAM by participating in the Nordic Way 3 project in cooperation with Finland, Sweden and Denmark. The project focuses on developing and testing Day 1 and Day 1.5 services and developing an interchange system for exchanging information related to these services. A dedicated interchange system has through the Nordic Way 1 and 2 projects been developed for exchanging information, and in the Nordic Way 3 project, this system will be developed further to bring it close to a production ready system.

The Nordic Way 3 project period is from 2019 to 2023 and is co-financed by the European Commission through the Innovation and Networks Executive Agency (INEA).

In Figure 1, a simplified topology of the communication within CCAM is shown, with the Interchange entity in the top, communicating with vehicle OEM-clouds and other service providers through a Basic Interface-interface, denoted BI. The Improved Interface (II) is designated to communicate with other Interchange entities within other regions or countries. A key feature in this system is that information can flow freely across borders, so the driver can utilize services when travelling abroad.

The vehicles can then communicate through cellular network or ITS-G5 to exchange information and work smoothly across borders.

[1] The vision zero is that no persons shall be killed or severely injured in road traffic

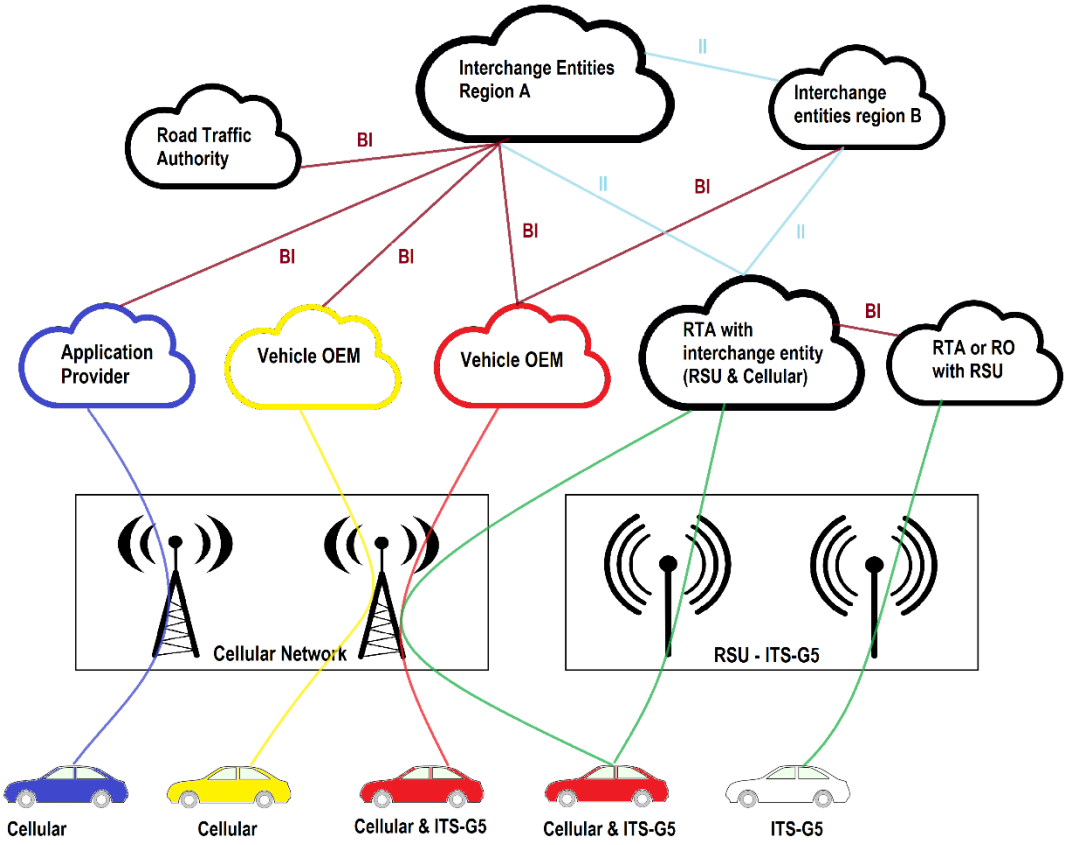

Figure 1 Communication between vehicles and backend[3]

There are two alternatives as data carriers for V2X; cellular network and ITS-G5. ITS-G5 standard is loosely spoken "WiFi for cars" which is reserved in the 5.9 GHz band and thus short-range communication. Cellular network is the same mobile network as used by phones and other mobile devices. When these two technologies are combined, it is hybrid communication.

As a part of the Nordic Way 3 project, a series of services will be chosen to be developed and tested during the project period. These services can for example be Road Works Warning, that can warn drivers that road work is ongoing so the driver can adopt the driving or take an alternative route to avoid the road work.

When the services are deployed and tested. In order to test these services, a test system is which can implement these services are needed.

1.2 Objective

The objective is to make a system for testing these services over 4G cellular networks. The system is to be used for collecting data from test runs and analyze the results to figure out how good some of the different services works over 4G cellular network. In order to analyze the results, a special focus will be the logs that the system generates.

The scope in this project is to make the basis for a system that can replicate the key functionality for a system that serves as an "Vehicle OEM" or "Application Provider" and is

able to test and potentially identify shortcomings of implementing over 4G cellular network. The task description is in Appendix A.

1.3 Scope

In this project, the services will not be implemented in this system, as the decision for which services is going be developed and tested in the Nordic Way 3 project have not been agreed upon within the start of this project.

A GUI will not be made for this system, as the basis for a GUI is heavily connected with the services.

A short analyze will be made to figure out the key elements needed.

1.4 Methods

The de-facto standard method for software development is a variant of the Agile methodology. While many different methods are umbrellaed under the Agile, DevOps is the one method chosen for this system.

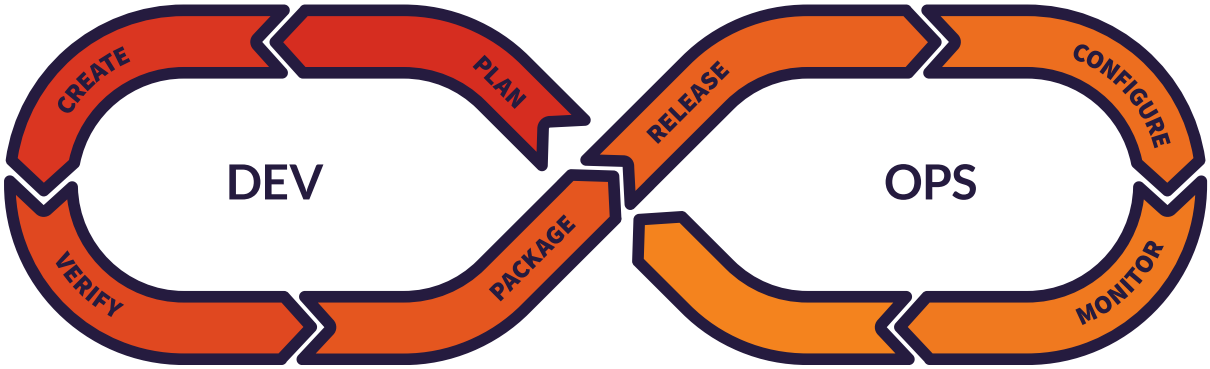

Figure 2 DevOps process[4]

The main difference between DevOps and traditional Agile methods, is that the operation of system is a part of the methodology. In the Dev-part (development), as seem on left hand side of Figure 2, it uses Agile methodology for developing the software. While for the Ops-part (operation), the same team or persons is participating in deploying and keeping the system running. This methodology has been widely adopted in order to remove silos between developers and operational teams. A motto with DevOps can be shortened into: “you build it, you run it”, referring to a motivation, that a team don’t want to make trouble for them self, but will deliver good quality software to avoid getting problems in operation.

1.5 Tools and software used in relation to the DevOps process

Looking at the overall perspective, Git is used as version control system of software together with Gitlab as remote repository. Git is involved in most of the stages and is to be looked on a as a “single source of truth” for the system.

I chapter 1.5.1 to 1.5.7, the tools used related to the seven different stages in Figure 2 is elaborated shortly.

1.5.1 Plan

- Star UML for visualizing results from analyze

1.5.2 Create

- VScode with Python extension as IDE
- Star UML to visualize the software

1.5.3 Verify

- Manual exploratory testing
- Integrated testing in Gitlab CI/CD tool

1.5.4 Package

- Gitlab's built in CI/CD tool that builds and package docker containers
- Gitlab registry for storing containers

1.5.5 Release

- Gitlab CI/CD with Helm to generate helm charts and perform deployment of containers

1.5.6 Configure

- The configuration is stored in the Gitlab repository and deployed with configfiles

1.5.7 Monitor

- Elasticsearch Metricbeat with Kibana dashboard
- Helm and Kubectl to generate helm charts and perform deployment for containers

2 Analysis of system requirements

From Figure 1, the system will at least need two parts, a backend system that communicates with the interchange system and an onboard unit (OBU) in a vehicle with connectivity to cellular network that is able to communicate with the backend system.

2.1 Use cases

2.1.1 Backend system

The backend system has following functional requirements:

- Transmit messages with interchange node
- Transmit messages with OBUs
- Log all transmitted messages
- Filter messages

2.1.2 Onboard unit

The functional requirements for the OBU:

- Transmit messages with central system
- Connect to cellular network
- Transmit vehicles positional data
- Log data

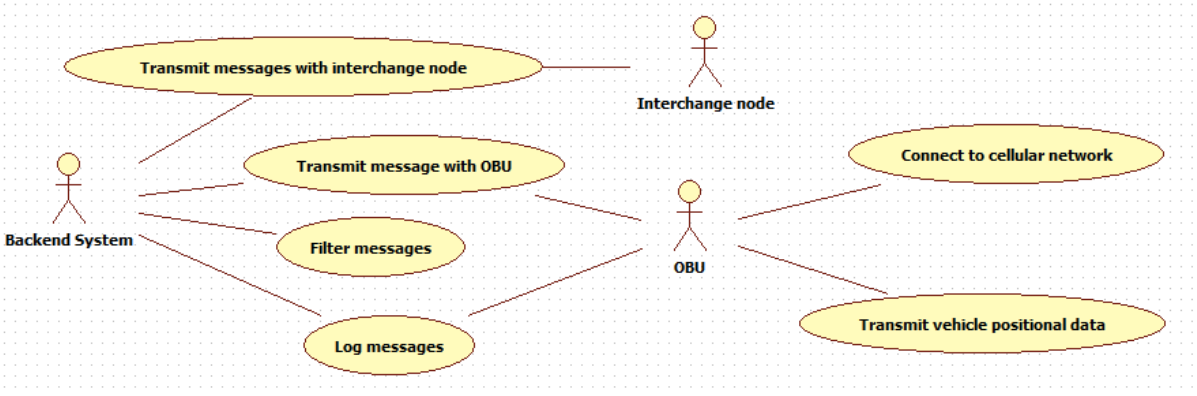

Figure 3 Use case diagram

2.2 Business logic

The overall business logic for the system is shown in Figure 4, where the key functionality is that the message flow can be bi-directional between the Interchange node, backend system and OBU. For the messages in the backend and OBU, all messages will be logged to a log system, as the logs will be subject to analyses.

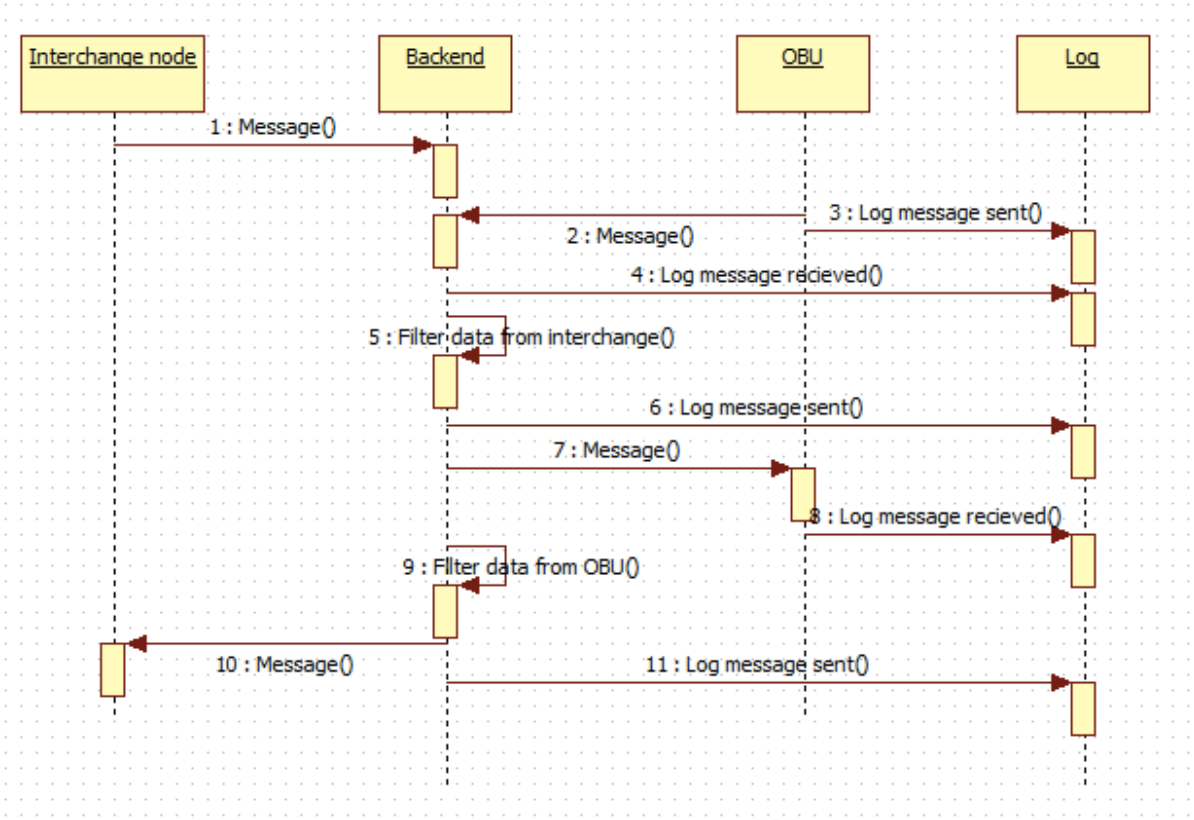

Figure 4 Business logic

2.3 System architecture

With the nature of this system, the OBU will be placed in a vehicle and the backend will be in a cloud platform[2], since cloud platforms have become a de facto standard for most central applications. A parameter to take into consideration when choosing platform, is the location of the datacenter, since a datacenter on the other side of the world is more prune to be affected by latency then a datacenter in a region relatively closer.

The requirements for the backend system, where the transmittal of messages is the most important parts, makes it suitable for a microservice architecture as it has rather clear boundaries and a dedicated purpose. It is then easy to foresee that for each new service implemented, implementing these in parallel, without needing to maintain the other services would improve serviceability, and make it easier to implement more services without changing the other implemented services.

[2] A cloud platform can be public cloud (e.g. Google Cloud), hybrid cloud (Google Anthos) or private cloud

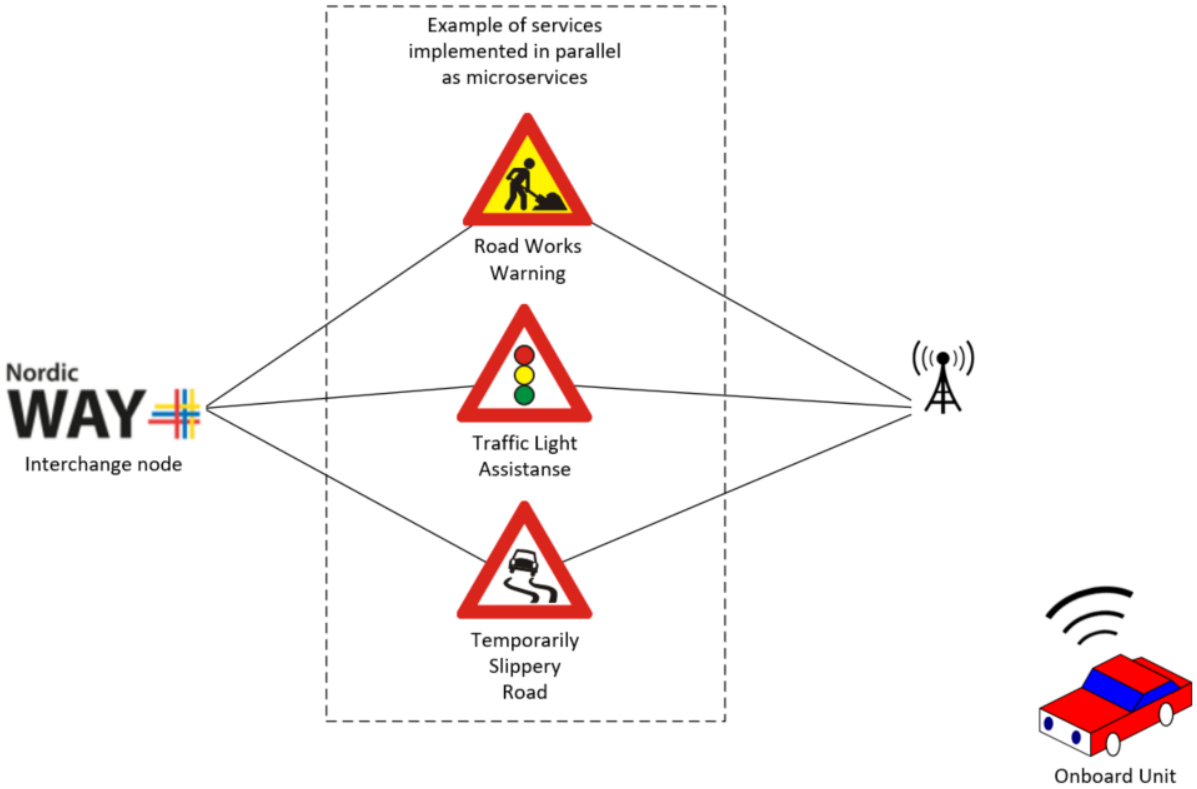

Figure 5 Principle of system architecture

2.4 Communication between backend and OBU

The communication between the backend and OBU will naturally rely on internet as a data carrier with TCP/IP in layer 3 and 4 in the OSI-model.

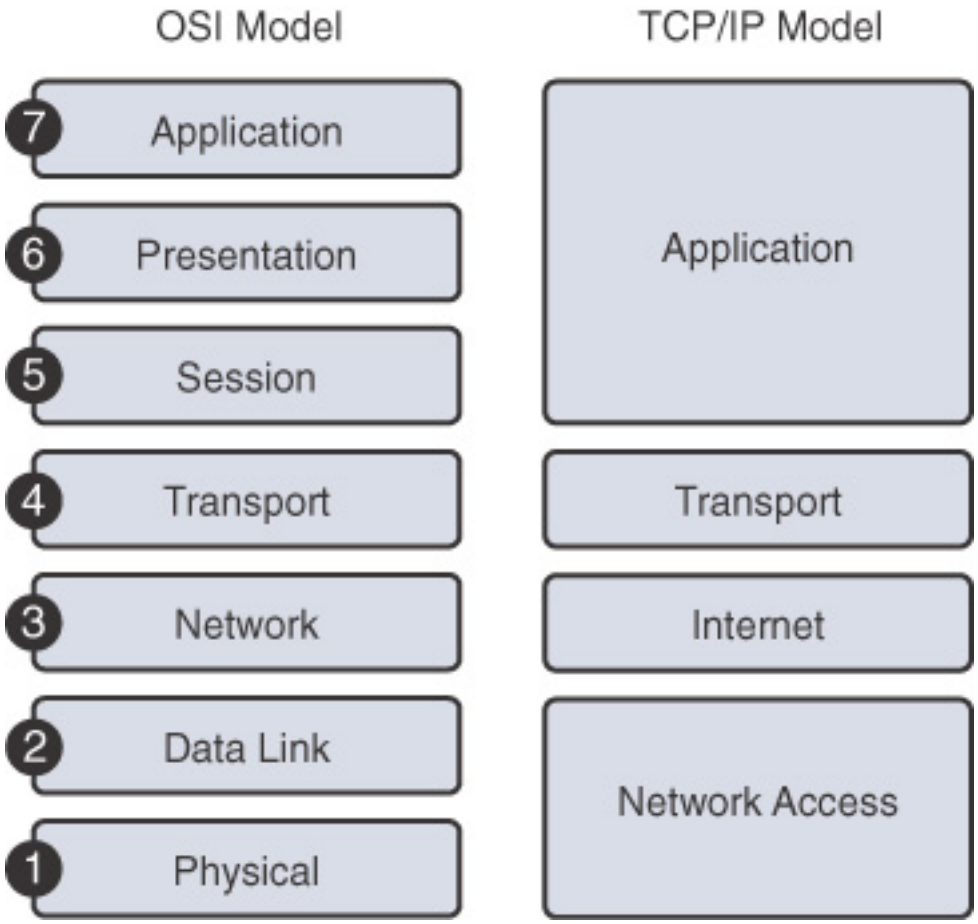

Figure 6 OSI-model with a 7-layer and 4-layer model[5]

When choosing a protocol in the application layer, many considerations can and should be made, but a key element would be to replicate what the OEMs are doing. According to HiveMQ[6], they support 2 of the top 3 German car manufacturers with connected cars platform. HiveMQ is a vendor of MQTT ecosystems and among others, delivers a MQTT-broker.

MQTT is a lightweight publish/subscribe protocol that uses a broker to transmit the messages. Using a broker, makes it easy to implement several services in parallel since the services publishes and subscribes to the broker through different topics.

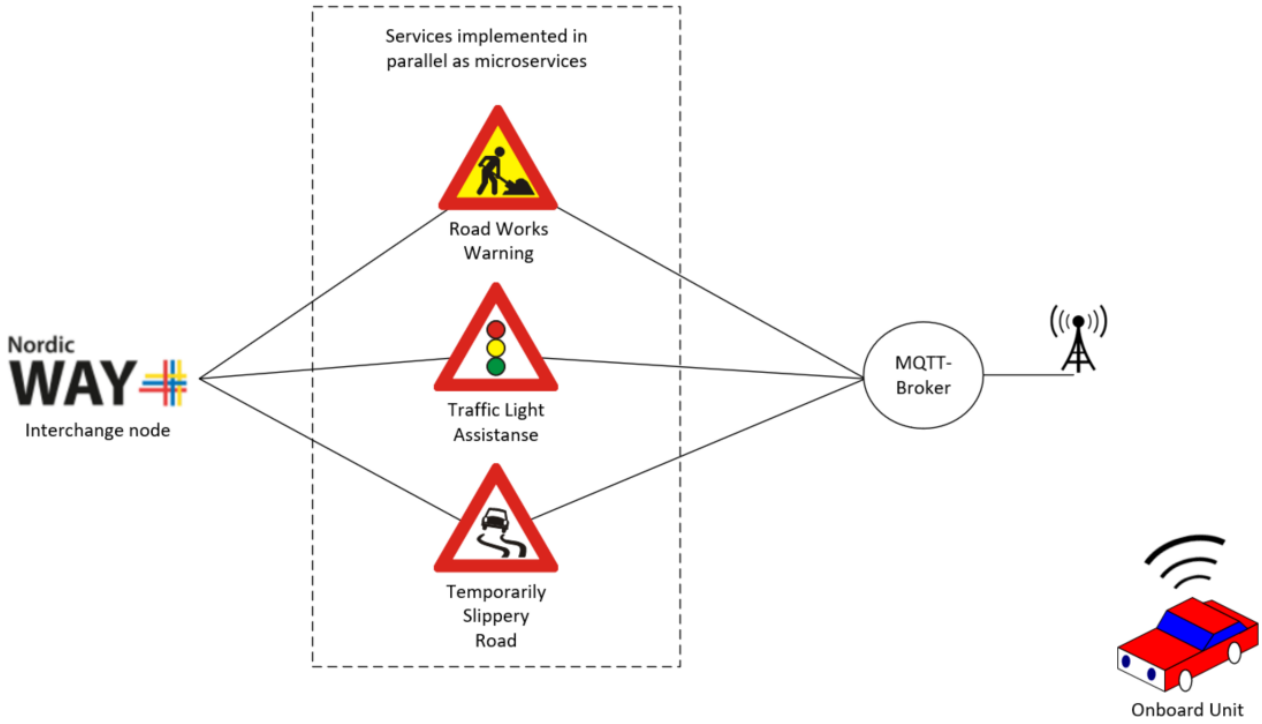

Figure 7 Architecture with MQTT-broker

2.5 Log system

A log system can be as easy as writing logs to a CSV-file, but as the number of logs grows, keeping track of them will become a hassle. A dedicated system to handle the logs is therefore desirable. An important feature is thus to use a system that is easy to integrate with the business logic, where logs will be retrieved from a cloud system, and also from OBU which can be situated in places with no or limited internet connection.

3 Cloud system components

3.1 Logging system

A much used and popular choice for storing and searching logs is the elastic eco-system. The main module, Elasticsearch[7], is built upon Apache Lucene and is a NoSQL search engine which is possible to interface with the Kibana web based graphical interface. Kibana[8] is a tool for visualization and searching in logs with a user-friendly interface.

To ingest logs into Elasticsearch, it can either be done through its REST API, but also through separate modules with a designated purpose, like Filebeat[9] that harvest logs from containers running in Kubernetes and ship them to Elasticsearch without the need for extra configuration in the containers deployed. This feature in Filebeat makes it a really powerful tool since it is less error prone than to configure log handling in each deployment.

For collecting metrics from the Kubernetes cluster, Metricbeat[10] is able to collect metrics and comes with a pre-defined dashboard for Kibana to visualize the state in the Kubernetes cluster. An example of the dashboard is shown in Appendix B.

In Figure 8, the modules mentioned, and their context is shown together with modules not used in this system.

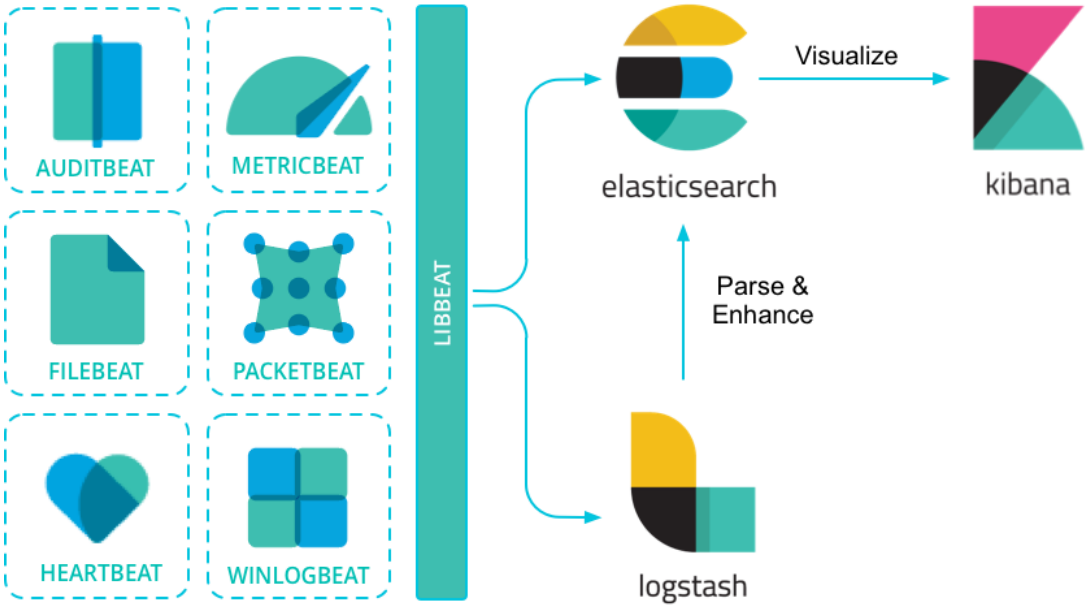

Figure 8 Elastic eco-system with most used components[11]

By default, Elasticsearch comes without any security enabled and since this will be exposed publicly on the internet, a minimum of security measures needs to be enabled. Between the modules in the Elastic eco-system, SSL-encryption is enabled, in addition to username and password for authentication. Kibana and Elasticsearch is facing public internet, and the communication is protected over HTTPS and authentication is done by username and password.

3.2 MQTT-broker

An open-source and popular MQTT-broker is Mosquitto. It supports the most basic features necessary for this system, with TLS-communication and PKI-authentication for clients connecting to the broker. Using PKI-authentication means that the device connecting to the MQTT-broker needs to be in possession of a public certificate and a private key to authenticate. Using this method, username and password is avoided.

3.3 Backend service

The program that will serve as the backend service, will be running in a Python docker container. This will be further elaborated in chapter 6.

3.4 Ingress and Cert-manager

Since Elasticsearch, Kibana and the MQTT-broker will be exposed to the public, they need a public IP-address. In order to avoid using one IP-address each, which will incur cost, a dedicated reverse proxy is desirable. Using Ingress Nginx, this reverse proxy will perform path-based routing to different internal IP-addresses within the cluster. For the MQTT-broker, which relies on a connection on a TCP-port, Ingress Nginx will perform dedicated port passthrough to the MQTT-broker[12].

For web-based services, most browser require HTTPS with a certificate signed by a trusted authority. To fetch certificates from Let's Encrypt, which is a trusted authority, cert-manager will be used and then issue the certificate to the ingress controller[13]. Cert-manager will then also renew the certificates before they expire.

3.5 Helm

To generate the deployment files for Kubernetes, Helm is used as tool. With this tool, default Helm charts provided from the software provider is configured through a values.yaml file and Helm then generates the necessary deployment files and deploys the application to Kubernetes.

The following applications will be based on a pre-defined Helm chart:

- Elasticsearch[14]
- Kibana[15]
- Filebeat[16]
- Metricbeat[17]
- Ingress-Nginx[18]
- Cert-manager[19]
- Mosquitto[20]

3.6 Cloud system with components

One of the cloud systems NPRA uses, is the Google Cloud Platform. It is among others used for hosting the NordicWay interchange node and access have been provided to a sandbox-project within GCP for this project. To host the containers within GCP, a Kubernetes cluster is used for orchestration of containers.

There are other possibilities to run containers in cloud, for example in Google Cloud Run and it would probably be an easier solution then using Kubernetes. Since Kubernetes is an open source system and the other major cloud providers also deliver Kubernetes as a service, and it is possible to install it on bare-metal and running it locally. By using Kubernetes, the system developed in this project can be moved to other clouds with simple means and using the same tools.

The Kubernetes cluster used in this project consists of a two-node cluster, with possibility to automatically scale up to three nodes. It resides in one of GCP's datacenters in Europe-north-1 region, which is known to be some place in Finland. With two nodes running, it has 4 vCPUs and 16 GB RAM. Even though this isn't much more power than a decent laptop, it is sufficient for now. Should it be a need for more power, it can be scaled up to several nodes and clusters as scalability is one of the key advantages with Kubernetes.

Looking at Figure 9, the components in chapter 3 are visualized to give an overview and to set it in a context of how the system is built.

Figure 9 Cloud components for the total system

3.7 Communications

The communications in the system are shown in Figure 10. The MQTT-communication will run over TLS to ensure that the data transmitted can't be sniffed up by others. This is an important feature of the system, as OEMs will be using some sort of protection of the data transmitted between a vehicle and OEM backend system.

Sending logs from the OBU to Elasticsearch, the built in REST API of Elasticsearch is to be used. This communicates over HTTPS and authentication can be done with username and password, API-key or certificates.

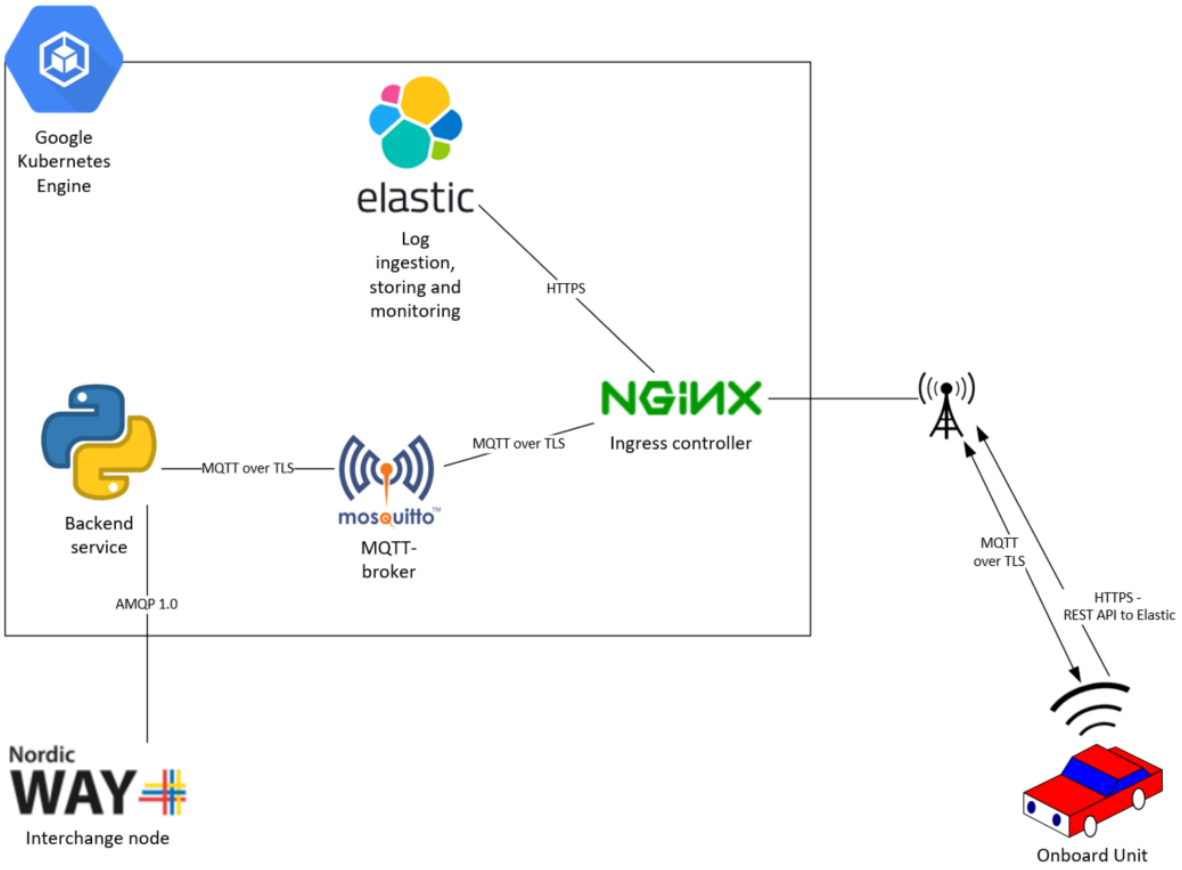

Figure 10 Communications in system

4 Onboard unit - physicals

A factor for the OBU is the size and portability, needing to support connection to a 4G router and retrieval of positional data. On the application layer, the device needs to support MQTT-over-TLS and HTTPS REST API to communicate with the backend- and log system. Although these requirements potentially can be handled by a microcontroller, e.g. an ESP32, choosing a microcomputer assures greater flexibility. A microcomputer can run different OSes and possible also run containerized applications if the supported OS and hardware support it.

The choice is therefore a Raspberry Pi 3 Model B since it is of small size, but still offers great functionality with ethernet-port to interface a 4G router, USB-ports to interface a GPS and can run different OSes if necessary. It is based on an Arm 32-bit processor and should be sufficient for the tasks outlined in chapter 2.1.2. which needs to be considered if demanding tasks is to be run on it.

4.1 4G-router

When selecting 4G router, several factors come into account:

To avoid using Wi-fi between router and Raspberry Pi, the router must have physical ports. Using Wi-fi shouldn't be a problem, given the short distance between the components, but experience tells that Wi-fi is more error prone than ethernet, and to avoid potential mistrials the safer solution is chosen.

Another important feature for the router is to use one with an operating system that is possible to extract the signal quality in an automated way[3]. The signal quality will be subject to logging, as this will have effect on the message throughput for the system.

The choice for router, resulted in a Teltonika RUT955 industrial router. This was reasonably priced and comes in an industrial grade encapsulation. It runs a variant of the OpenWRT-linux distribution, which gives possibility to access the router through JSON-RPC to retrieve signal data[21]. Another benefit is that it can use supply voltage in a range between 9-30V, making it well suited for connecting to a cars 12V-grid.

4.2 Positional data

Although the router selected have GPS built in, and it is possible to retrieve the positional data with the same method as retrieving signal strength, it will not be good enough due to a max sampling rate of 1 Hz within the router. When the car is running 80 km/h, it moves 22 m/s, and with 1 Hz resolution on the GPS, it will be 22 m between each registration.

A dedicated GPS, with a higher resolution is therefore desirable, and a Columbus V-800 GPS is supplied from NPRA. This GPS is based on an MTK-3329-chipset and has a resolution up to 5 Hz, it connects to the Raspberry Pi over USB and is an all-in-one unit without the need

[3] As per Tomas Levin recommendation at meeting 28.09.2020

of extra antennas. It communicates over a serial-bus with the NMEA 0183 protocol, which is common for many external GPS'.

4.3 Physical setup

The component setup with wiring is shown in Figure 11

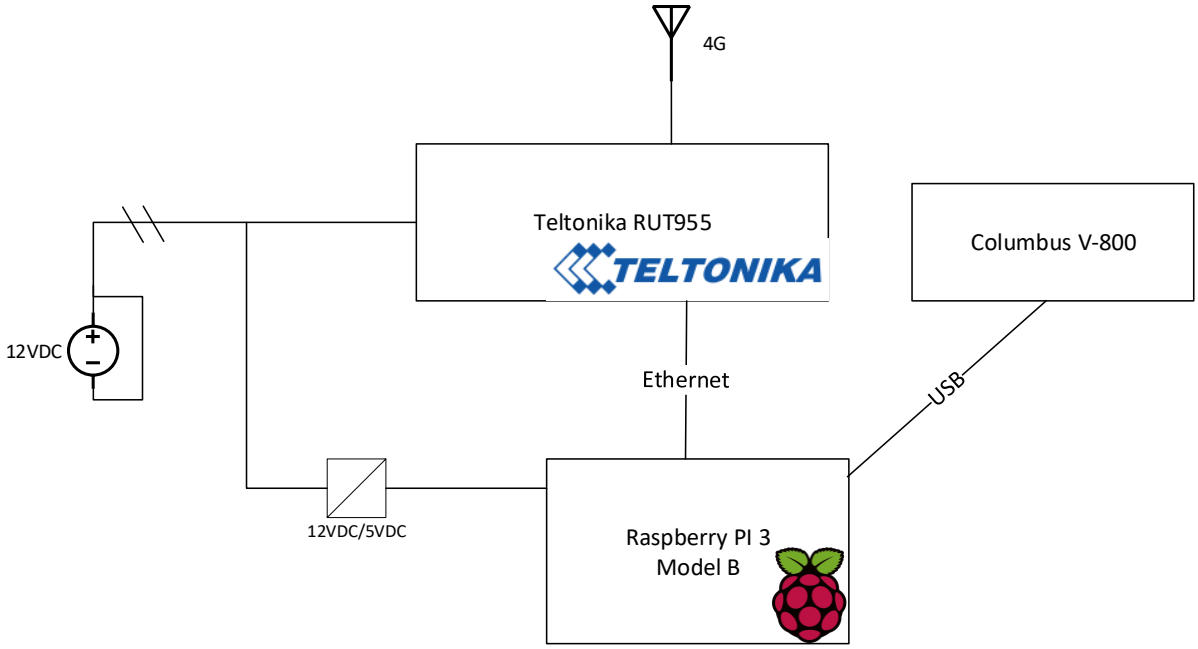

Figure 11 Component diagram

5 Onboard unit - software

5.1 Operating system and OS-installed software

For the Raspberry Pi, the operating system is the Raspberry Pi OS Lite, previously known as Raspbian Lite, and is the officially supported OS from the Raspberry Pi Foundation[22]. This is a minimal operating system, without a GUI.

Even though the software for the OBU runs in a container, it is hard to abstract everything into a container when dealing with physical components. Therefor the GPS is connected to a GPS service daemon, gpsd in this case, on the host and consumers can then query gpsd for GPS-signals on the host computer[23]. The gpsd can then be queried from a client to the IP of the host.

Another case is that the Docker engine uses the time of the host-machine. When comparing different timestamps of the logs collected, it is of uttermost importance that they are synchronized, and the easiest way to ensure that they are on the same state is to use the same NTP-server. By recommendation from Google, it is advised to use the same NTP-server as GCP uses on the Kubernetes cluster, the reason for using same NTP-server is due to how Google handles leap-seconds[24]. By using the time.google.com NTP-server both the containers running in the cloud and on the OBU is synchronized to the same time reference.

Since containers are ephemeral by design, the generated logs and certificates used to authenticate with the MQTT-broker, will be stored on the host instead of inside the container. This way they persist even if a container crashes and needs restarting.

Regarding the network connection, the container is connected to the network of the host, and network vise, acts as if the container is running on the host.

To start the container, docker-compose is used. This ensures that network and volume mapping is consistent each time, and also restarts the container if it crashes. Since the only task to the container is to run the OBU-software, a failure in the OBU-software will result in that the container crashes and that docker-compose restarts it.

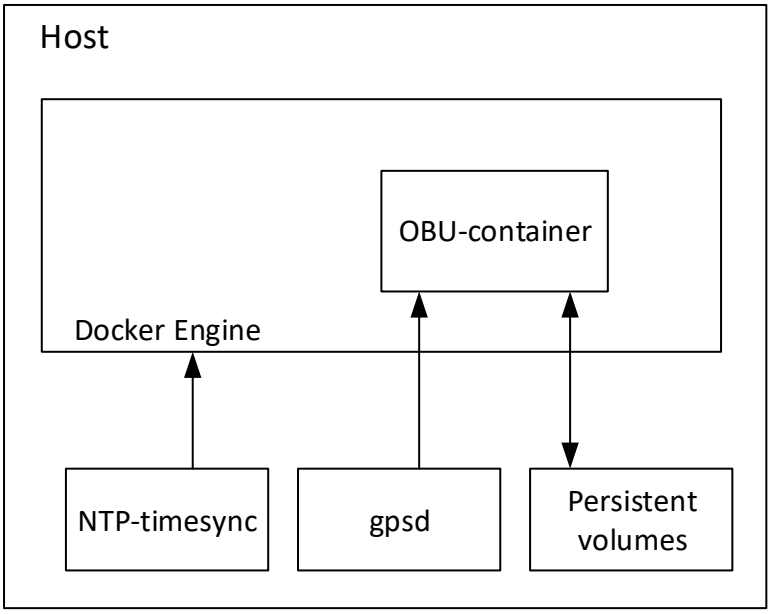

Figure 12 System abstraction

5.2 OBU-container

The base image for the OBU-container will be an official python docker image, and fortunately, the docker-community maintains a small-sized python docker image based upon Alpine Linux. The version made for Arm-processor is about 17 MB, compared to the image based upon Debian Buster which is approximately 300 MB. This offers a nice reduction in image size, which is great suited for small, resource limited devices.

As part of the DevOps-methodology, the container is built with a CI/CD-pipeline. This is done within Gitlab's CI/CD tool, but out of the box, this tool only builds containers to run on either x86 or AMD64-architecture. Trying to run a non-Arm-container on an Arm-device will simply fail. In order to be able to build container images for Arm-architecture, this needs to be specified explicitly in the .gitlab-ci.yml to be able to perform multi-arch-builds[25]. By performing a multi-arch build, it is possible to build for several architectures in the same pipeline. It is therefore possible to build images for AMD64 and Arm-architecture in the same pipeline, which is a benefit if testing on an AMD64 device or changing hardware in the future.

5.3 OBU-software

The client software for the OBU is built with Python 3.9. All the necessary configurations are stored in a config-file in order to make the configuration easily manageable, but also to have a history of the file in a git-repository. The configfile is in YAML-format which contains all the configurational details needed, this includes; paths to certificates, MQTT-topics, URLs and port numbers.

In addition to the python 3.9 standard libraries[26], the following have been used:

- asyncclick[27] for passing the configfile to client.py on startup

- PyYAML [28] for parsing and converting the configfile to type dict
- Paho-mqtt[29], from this library, the TLScontext.py function have been cherry-picked to make a convenience-function for making a TLS-context used for authenticating to the MQTT-broker
- Python-json-logger[30] for making the logs in JSON format
- Asyncio_mqtt[31], and the advanced example in this library have been used as basis for the for MQTT-client
- Requests[32] for sending JSON-RPC requests to the 4G-router
- Gpsd-py[33] for retrieving GPS-data from gpsd
- Elasticsearch-py[34] for interfacing the Elasticsearch cluster with the built in REST API of Elasticsearch

In Figure 14, a sequence diagram shows the basic functionality of the software that reads from gpsd and the 4G-router. The positional data is logged and sent to the MQTT-broker, while the signal strength is only logged to file.

Beside from the functional part of the software, some special considerations have been made for the software:

Although it is possible to parse the NMEA 0183-messages from the GPS directly, using the gpsd-py library[33] connects to gpsd on the host, and the GPS information is easily parsed using the functions in the library.

The MQTT-messages is published to the MQTT-broker on a two-component topic, with the first component of the topic, being the “car chassis number” as this is a unique number for each vehicle. The second component of the topic is made up with a semantic reference to the information published, for example “location” or “altitude”. This makes an easy reference system of meaning of the message, that is both machine and human understandable.

Each message is published with QOS 1, meaning that for each message that is sent, a confirmation from the MQTT-broker is required. This is used to be able to get a notification if the MQTT-broker is out of reach, and a driver can’t rely on getting reliable information from a service.

To get the sending of MQTT-messages as effective as possible, it is crucial that a potential limited capacity is utilized. If the transmittal of messages between the OBU and MQTT-broker is synchronous, the OBU will be sitting waiting for the confirmation message from MQTT-broker, before it can progress with the next message. If communicating on a network with limited capacity, this response can take some time. To avoid that this waiting time is blocking other messages, the program is made asynchronous. In short terms, this means that the program let other functions run when waiting for response from e.g. an MQTT-broker. This is particular useful when working with network connections where the estimated time of a response is unknown[35].

Since the logs is to be parsed to Elasticsearch, it is important that they are machine readable in order to get consistent log ingestion. JSON is a format that is easily understandable for both human and machines, and therefore all the logs are globally enabled[36], and are saved to a file in JSON format by using the Python-json-logger library[30]. To avoid the logs being sent to Elasticsearch while being out driving, and thus occupying bandwidth that should be

used for services, the logs are temporarily stored on the host. In order to get the logs into Elasticsearch, a separate python program have been made to upload the logs in bulks of 500 logs. This upload needs to be initiated manually to ensure it is uploaded at a convenient time.

5.4 CI/CD pipeline

The pipeline consists of three stages, one for checking the linting of the Dockerfile, one for building the container and pushing it to Gitlab registry and one for checking the linting of the python files in the newly built container. This is visualized in Figure 13.

Since the Raspberry Pi can be situated on different networks, with different IP's and without DNS, a Continuous Delivery will be hard to achieve. It is therefore necessary to pull the container manually from Gitlab instead of an automatic deployment with tools like Helm.

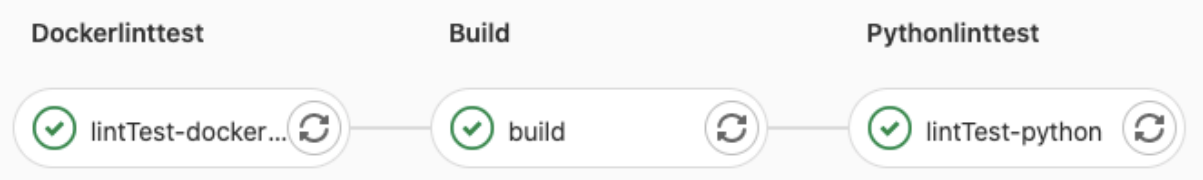

Figure 13 Pipeline for OBU

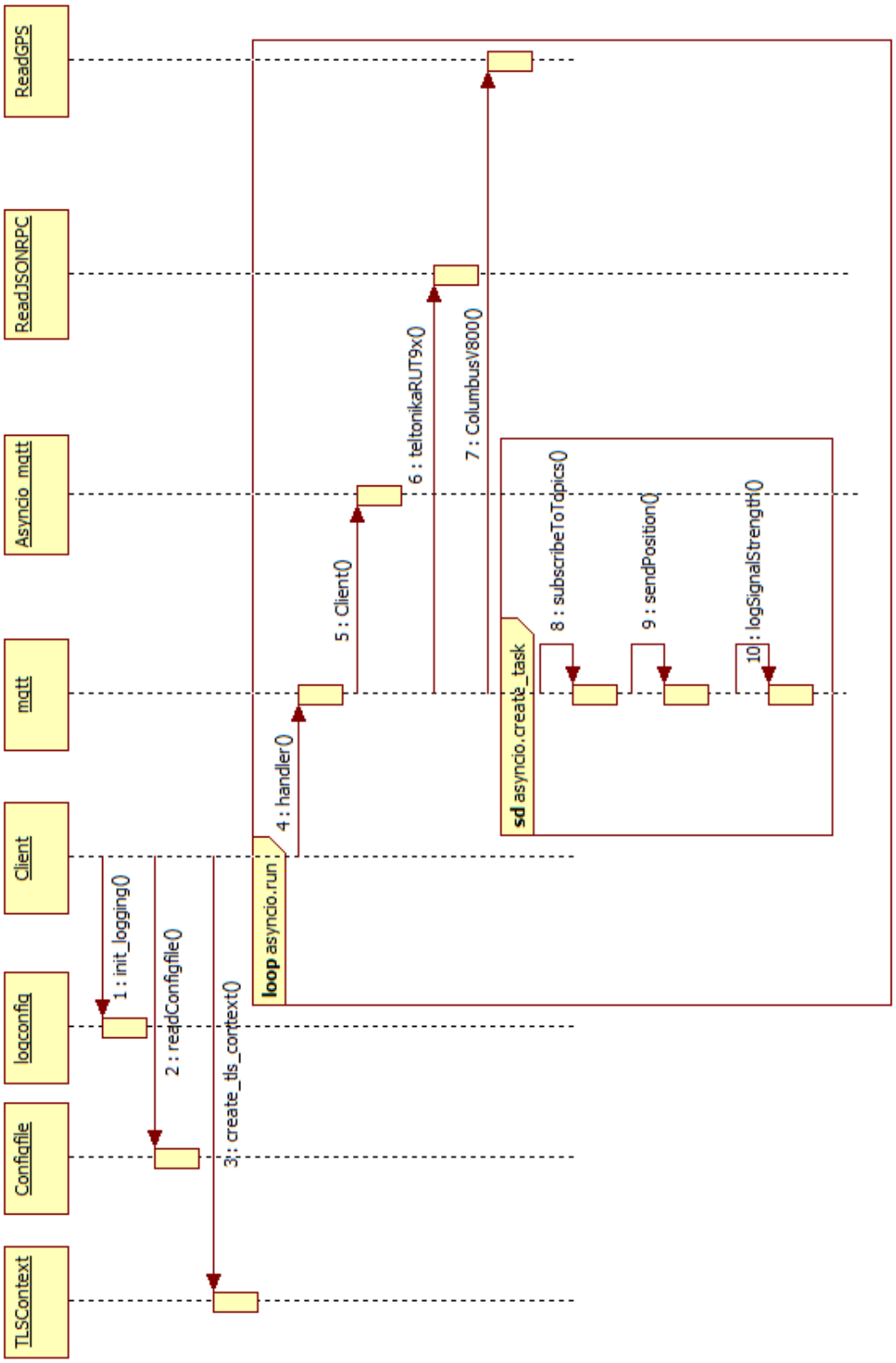

Figure 14 Sequence diagram for OBU software

6 Backend module

The standard Python container is based upon Debian Buster and is running Python 3.9

6.1 Backend software

The backend software uses, in addition to the standard Python 3.9 library[26]:

- CLICK[37] for passing the configfile to the main.py during startup
- Pyngus[38] for connecting to the Nordic Way 2 interchange node. Pyngus is a wrapper around the Apache Qpid python library[39]. Apache Qpid is used in development for the Nordic Way interchange node, but this uses the Java library.
- Paho-mqtt [29] for connecting to the MQTT-broker.
- Python-json-logger [30] for formatting the logs into JSON format.

Due to some problems with the Nordic Way 2 interchange node, it wasn't possible to retrieve messages from the interchange node[4]. The Nordic Way 3 interchange node has a different message format and is under development. Since the Nordic Way 3-project is still in an initial phase, there is no messages flowing on the deployed version.

As per now, this limits the functionality of the backend module to only subscribe to messages from the MQTT-broker and streaming the logs to stdout. As noted in 3.1, the logs written to stdout, will be fetched by Filebeat, and parsed into Elasticsearch.

The setup of logging is the same as with the OBU in chapter 5.3, except for the writing to file.

6.2 CI/CD pipeline

The pipeline for this module contains at most 4 stages, as shown in Figure 15. First stage checks if there are errors in the docker file. This is only run if there have been changes to the dockerfile. Build stage is as the name implies, the stage where the docker container is built and pushed to gitlabs container registry. Python lint test checks the python files in the newly built container for import errors, syntax error etc. DeployTest deploys the module to Kubernetes and is semi-automatically deployed. A dedicated Helm chart is used for deploying the module to Kubernetes.

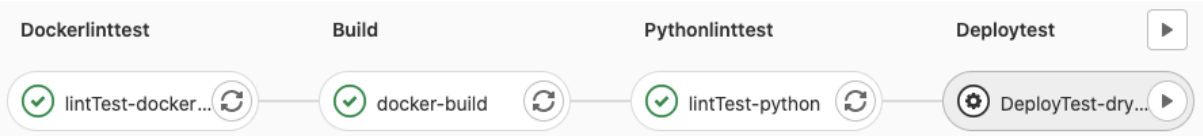

Figure 15 CI/CD pipeline for Backend

[4] As per dialog with Christian Berg Skjetne on Teams, 12.11.2020

7 Test of system

The test of the system was performed during a drive, starting at "Ålesund trafikkstasjon, Vestre Olsvikveg 13, Ålesund" and ended at Horgheim in Rauma municipality, close to the famous "Troll wall". The GPS-antenna was placed on the roof of the car, while the 4G-antenna was placed inside on the dashboard of the car.

The main success criteria were:

- Backend module logged all received messages
- Onboard unit sending positional messages to MQTT-broker
- Onboard unit logging to file locally on host
- File from Onboard unit parsed to Elasticsearch manually with a dedicated script.

A downside with this test, is that no messages was transmitted from the backend to the OBU. This would of course give the test an extra dimension, but with the use of MQTT-messages that uses QOS 1, the OBU will both send and receive messages from the MQTT-broker.

7.1 Test results

When comparing the amount of positional data logs in Elasticsearch, the OBU have logged 25 084 to file and later parsed into Elasticsearch. The backed module has logged 24 628. This is a difference of 456, meaning that not all messages were received by the backend module. One of the main success criteria is therefore not fulfilled.

The OBU logged a total of 25 084 positions to file and sent 25 083 positions to the MQTT-broker. This one message in difference, is acceptable and is probably due to a message that weren't sent in the end after the OBU container was stopped.

The total number of logs stored in file locally was 230 767 which is the same as number of logs stored in Elasticsearch.

To visualize the data collected, the positional data is plotted in Figure 16, this shows the route travelled, except for the tunnels where there are no GPS signals. In Figure 17, the signal strength along the route is shown. Code for generating plots is uploaded on github[40]

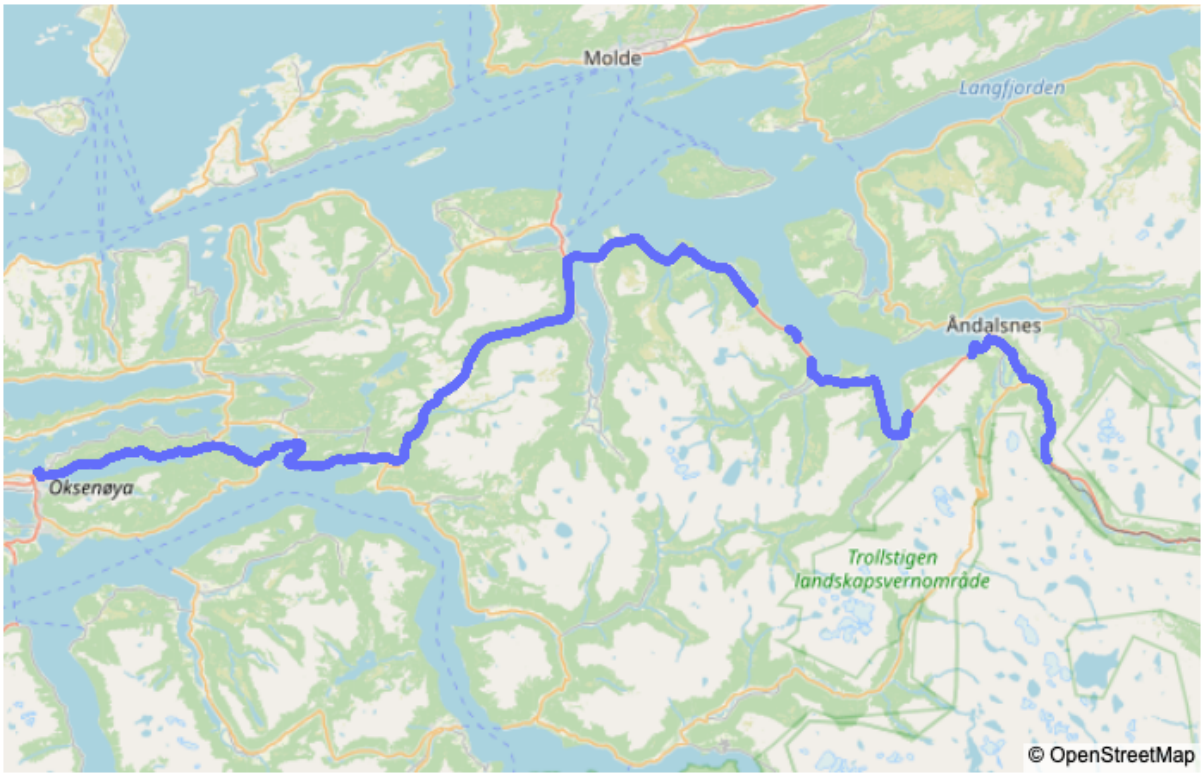

Figure 16 Positional data during test of system. Plot generated with plotly.py[41]

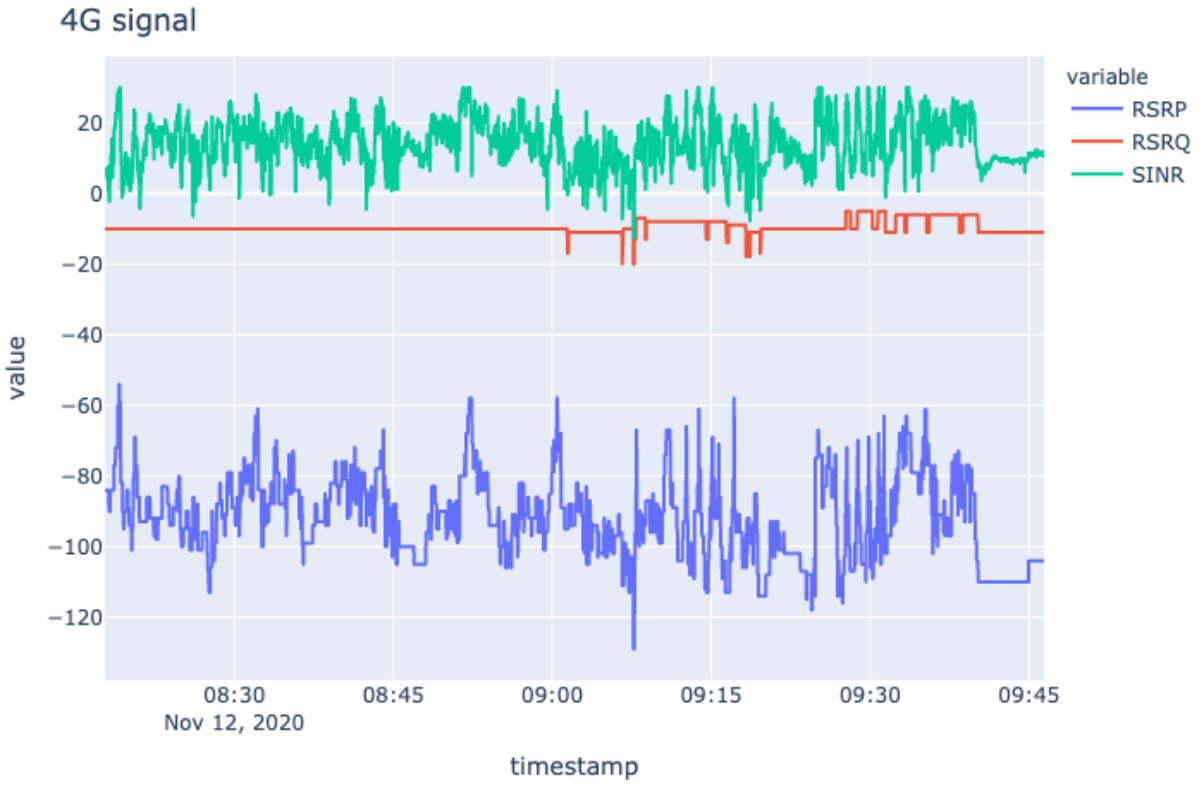

Figure 17 Signal strength during testrun. Plot generated with plotly.py[41]

8 Conclusion and discussion

As this report shows, it is possible to develop a flexible test system to test different V2X-services over 4G cellular network. The important feature of transmitting positional data from the OBU to the backend module is per now a clear drawback, but it is assumed that this is bug that is fixable.

A part of the system requirement is a good logging system, and exemplified with the fault found during testing, this shows that a good logging system will ease the work of finding these faults. If an alternative by maintaining CSV files had been used, these probably be imported to Excel or similar tool for analysis, and for each new CSV file, the logs would be imported. Using Elasticsearch and Filebeat, it collects and stores the logs in a database and Kibana holds tools for searching logs in a simple manner, reducing the work for log handling.

The system is not only the communication between the OBU and backend module. In total a system has been set up to accommodate for further testing, that is flexible to use in future use cases and easily adopted. The fact that it is running in Kubernetes, makes it easy to move to different cloud vendors, giving it a possibility to assess the cloud vendors and possible test a data center in a region far away.

The CI/CD pipelines ensures that building of the backend module and onboard unit is concise every time and reduces chance for bad quality of the software. Although the two pipelines used here have included some tests, it would be wise to include more tests in the pipeline, for example unit tests and integration tests to ensure that nothing breaks after changes have been done.

9 Remaining work

As the test results show, the received messages weren't consistent with what was sent from the OBU, this clearly needs to be sorted out and a new test needs to be performed in order to verify that the system functions properly. It is thus assumed to an r

For testing of the services that probably is to be implemented in Nordic Way 3, it is necessary to connect the backend module to Nordic Way 3 interchange node when this is up and running. The NW3 will have a different message format than the messages used in the Nordic Way 2 interchange.

References

[1] W. H. Organization., "Global status report on road safety 2018," 2018.

[2] "Intelligent transport systems - Cooperative, connected and automated mobility (CCAM)." https://ec.europa.eu/transport/themes/its/c-its_en#:~:text=All%20members%20of%20the%20C,of%20the%20digitisation%20of%20transport. (accessed 28.08, 2020).

[3] J. Sundberg, "NordicWay 2 Architecture, Draft," 22.02 2019.

[4] G. Gruver. "DevOps explained." https://about.gitlab.com/topics/devops/ (accessed 24.09, 2020).

[5] S. K. G. "MQTT for IoT Applications : Introduction Series- PART 1." https://nexiot.com/mqtt-for-iot-applications-introduction-series-part-1/ (accessed 20.09, 2020).

[6] "Enabling the Connected Car with HiveMQ." https://www.hivemq.com/solutions/iot/enabling-the-connected-car/ (accessed 20.09, 2020).

[7] "What is Elasticsearch?" https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html (accessed 26.09, 2020).

[8] "Kibana — your window into the Elastic Stack." https://www.elastic.co/guide/en/kibana/current/introduction.html (accessed 26.09, 2020).

[9] "Filebeat overview." https://www.elastic.co/guide/en/beats/filebeat/current/filebeat-overview.html (accessed 26.09, 2020).

[10] "Lightweight shipper for metrics." https://www.elastic.co/beats/metricbeat (accessed 1.11, 2020).

[11] "What are Beats?" https://www.elastic.co/guide/en/beats/libbeat/current/beats-reference.html (accessed 26.09, 2020).

[12] "Exposing TCP and UDP services." https://kubernetes.github.io/ingress-nginx/user-guide/exposing-tcp-udp-services/ (accessed.

[13] "Creating a Basic ACME Issuer " https://cert-manager.io/docs/configuration/acme/#creating-a-basic-acme-issuer (accessed.

[14] "Elasticsearch Helm Chart." https://github.com/elastic/helm-charts/tree/master/elasticsearch (accessed 27.09, 2020).

[15] "Kibana Helm Chart." https://github.com/elastic/helm-charts/tree/master/kibana (accessed 27.09, 2020).

[16] "Filebeat Helm Chart." https://github.com/elastic/helm-charts/tree/master/filebeat (accessed 27.09, 2020).

[17] "Metricbeat Helm Chart." https://artifacthub.io/packages/helm/elastic/metricbeat/7.8.0 (accessed 31.10, 2020).

[18] "ingress-nginx." https://github.com/kubernetes/ingress-nginx/tree/master/charts/ingress-nginx (accessed 27.09, 2020).

[19] "Jetstack Helm charts." https://hub.helm.sh/charts/jetstack (accessed 27.09, 2020).

[20] "Mosquitto Helm Chart." https://github.com/halkeye-helm-charts/mosquitto (accessed 27.09, 2020).

[21] "RUT955 Monitoring via JSON-RPC linux." https://wiki.teltonika-networks.com/view/RUT955_Monitoring_via_JSON-RPC_linux (accessed 24.10, 2020).

[22] "Raspberry Pi OS (previously called Raspbian)." https://www.raspberrypi.org/downloads/raspberry-pi-os/ (accessed 26.10, 2020).

[23] "gpsd — a GPS service daemon." https://gpsd.gitlab.io/gpsd/index.html (accessed 26.10, 2020).

[24] "Configure NTP for your instances." https://cloud.google.com/compute/docs/instances/managing-instances#configure_ntp_for_your_instances (accessed 28.09, 2020).

[25] J. DROUET. "Multi-arch build, what about GitLab CI?" https://www.docker.com/blog/multi-arch-build-what-about-gitlab-ci/ (accessed 26.10, 2020).

[26] "The Python Standard Library." https://docs.python.org/3/library/ (accessed 1.11, 2020).

[27] Smurfix. "asyncclick." https://github.com/python-trio/asyncclick (accessed 31.10, 2020).

[28] K. Simonov. "PyYAML." https://github.com/yaml/pyyaml (accessed.

[29] R. Light. "paho-mqtt." https://pypi.org/project/paho-mqtt/ (accessed.

[30] madzak. "python-json-logger." https://github.com/madzak/python-json-logger (accessed 31.10, 2020).

[31] F. Aalund. "MQTT client with idiomatic asyncio interface " https://github.com/sbtinstruments/asyncio-mqtt (accessed 1.11, 2020).

[32] K. Reitz. "Requests: HTTP for Humans™." https://requests.readthedocs.io/en/master/ (accessed 1.11, 2020).

[33] M. Breem. "Python3 GPSD client." https://github.com/MartijnBraam/gpsd-py3 (accessed 31.10, 2020).

[34] "Python Elasticsearch Client." https://elasticsearch-py.readthedocs.io/en/7.10.0/ (accessed 15.11, 2020).

[35] B. Solomon. "Async IO in Python: A Complete Walkthrough." https://realpython.com/async-io-python/ (accessed 15.11, 2020).

[36] koehlma and Z. Gates. "Python: logging module - globally." https://stackoverflow.com/a/7622029 (accessed 18.11, 2020).

[37] "Click." https://click.palletsprojects.com/en/7.x/ (accessed 15.11, 2020).

[38] K. Giusti, F. Percoco, and J. O. Robles. "Pyngus." https://github.com/kgiusti/pyngus (accessed 15.11, 2020).

[39] "Apache Qpid Proton." http://qpid.apache.org/proton/ (accessed 15.11, 2020).

[40] A. Svindseth. "NWIXN-maps_and_plots." https://github.com/svinz/nwixn-maps_and_plots (accessed.

[41] "Plotly Python Open Source Graphing Library." https://plotly.com/python/ (accessed 18.11, 2020).

Appendices

Appendix A Task description

FM4017 Project

Title: Test system for testing the Nordic Way Interchange over existing cellular network

USN supervisor: Hans-Petter Halvorsen

External Partner: Statens Vegvesen / Norwegian Public Road Administration (NPRA)

Task Background: The road vehicles of today are trending towards becoming more and more cooperative, connected and automated. To be able to achieve this, the vehicles needs to rely on more than sensors installed in the vehicle. A key component in the vehicle of the future is communication from vehicle to vehicle (V2V) and vehicle to infrastructure (V2I). The V2I services can typical be “Geo fence”, “Road Works Warning”, “Traffic Ahead Warning”. As a road operator and road traffic authority, Statens vegvesen needs to start planning for implementing these services, and as a part of this, Statens Vegvesen participates in the NordicWay3 project. This project is a collaboration between Finland, Sweden, Denmark and Norway, and is partly financed by the Innovation and Networks Executive Agency (INEA) through the Connecting Europe Facility (CEF) program.

As a basis for some of these services, there is a need for a backend system that performs the exchange of this information, as visualized in Figure 18. In the earlier NordicWay projects, a prototype of the interchange has been developed and used by the pilots in the projects. This interchange will now be furthered developed in NordicWay3 and be brought closed to production and pilots in the NordicWay project will use the interchange for testing. In order to test this interchange before the pilot are put into action, a mobile test device for testing the interchange over cellular network from a car is wanted. This test device will be used later in the NordicWay3 project for testing of which services can be relevant to deliver over the interchange.

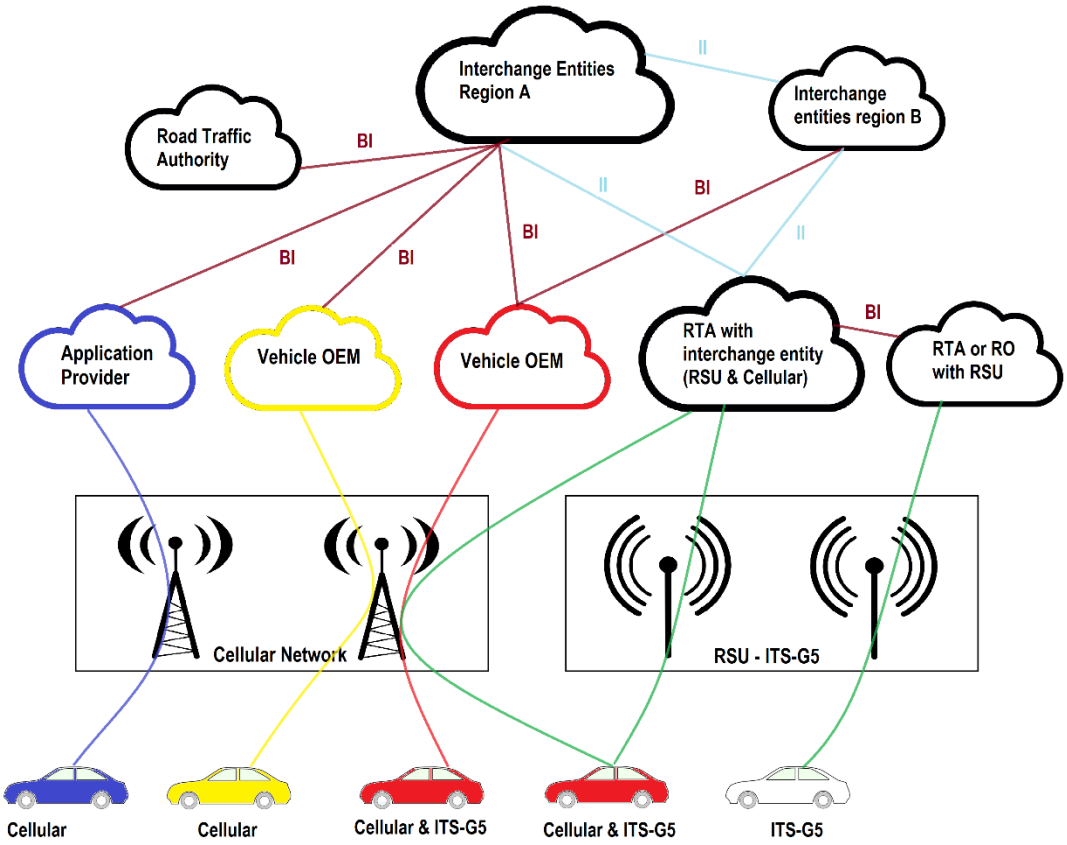

Figure 18 Communication between cars and backend

Suggested Project Activities:

- Find suitable equipment for this test system
- Connect the system to the interchange
- Make a plan for testing the system
- Perform some simple tests to verify the functionality

Student category: This project is reserved for Alexander Svindseth (IIA student employed at Statens vegvesen)

Practical Arrangement:

Statens vegvesen will give access to interchange and needed equipment for the device.

Statens vegvesens supervisor will be Senior Principal Engineer Ph.d Tomas Levin

Statens vegvesen will be responsible for providing a sensor for the project that will grade the work in collaboration with the supervisor from USN.

The resulting report should be public available.

Signatures:

Supervisor (date and signature):

Student (write clearly in all capitalized letters):

Student (date and signature):

Appendix B


Dashboard / [Metricbeat Kubernetes] Overview ECS
Full screen
Share
Clone
Edit
Search
KQL
Last 15 minutes
Show dates
Refresh
+ Add filter
Nodes [Metricbeat Kubernetes] ECS
Nodes
2
Deployments [Metricbeat Kubernetes] ECS
Deployments
14
Available pods per deployment [Metricbeat Kubernetes] ECS
cert-manager
cert-manager-cainjector
cert-manager-webhook
event-exporter-gke
ingress-nginx-controller
kibana-kibana
per 10 seconds
Desired pods [Metricbeat Kuber...
Desired Pods
15
Available pods [Metricbeat Kub...
Available Pods
15
Unavailable pods [Metricbeat K...
Unavailable Pods
0
Unavailable pods per deployment [Metricbeat Kubernetes] ECS
per 10 seconds
CPU usage by node [Metricbeat Kubernetes] ECS
gke-alexander-def... 618.7m nanocores
gke-alexander-def... 242.8m nanocores
Memory usage by node [Metricbeat Kubernetes] ECS
gke-alexander-default-... 5.2GB
gke-alexander-default-... 3.8GB