CHALMERS
UNIVERSITY OF TECHNOLOGY

Integrated system enables remote vehicle monitoring

Development of a scalable telematics platform for electric scooters
Bachelors's thesis in Computer science and engineering

Eva Bergsten
Albin Brovina

Department of Computer science and engineering
Chalmers University of Technology
University of Gothenburg
Göteborg, Sweden 2020

BACHELOR'S THESIS 2020

Integrated system enables remote vehicle monitoring

Development of a scalable telematics platform for electric scooters

Eva Bergsten
Albin Brovina

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer science and engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2020

Integrated system enables remote vehicle monitoring
Development of a scalable telematics platform
Eva Bergsten, Albin Brovina

Supervisor: Pedro Petersen Moura Trancoso, Niels Boardman Jonsson
Examiner: Peter Lundin, Department of computer science and engineering

Bachelor's Thesis 2020
Department of computer science and engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Telephone +46 31 772 1000

Cover: A Xiaomi M365 Electric Scooter with the system mounted in a box.

Typeset in LaTeX
Printed by Chalmers Reproservice
Göteborg, Sweden 2020

Integrated system enables remote vehicle monitoring
Development of a scalable telematics platform for electric scooters
Eva Bergsten, Albin Brovina
Department of computer science and engineering
Chalmers University of Technology
University of Gothenburg

Abstract

In recent years it has become important to put more and more technology in vehicles in order to satisfy the user. When introducing all this technology in the vehicles it is important to be able to analyze the data that becomes available. In this thesis work it is investigated if it is possible to build a reliable platform that can extract data from multiple electric vehicles, in this case electric scooters, and send it wirelessly to a web client using existing technology. The possibilities such a platform opens up for and how it could be used in the future is discussed.

To make a platform functional with existing technology it was required to learn about and integrate different tools with each other. Which solution to be used for which part of the system and to investigate and combine them into a complete system.

The thesis work resulted in a fully functional telematic platform that can extract, decode, send and display data from a scooter. The platform consists of a Raspberry Pi equipped with 4G and GPS using MQTT, Telegraf, InfluxDB and Grafana. It is designed to be scalable for usage on multiple scooters at the same time. With a few minor changes, the platform enables to present data from different vehicles and not just an electric scooter, which means that this project opens up for many possibilities when analyzing data.

Keywords: IoT, MQTT, electric scooters, decoding, wireless transmission

Integrated system enables remote vehicle monitoring
Development of a scalable telematics platform for electric scooters
Eva Bergsten, Albin Brovina
Department of computer science and engineering
Chalmers University of Technology
University of Gothenburg

Sammanfattning

På senare år har vikten av att implementera mer och mer teknik i fordon för att göra användaren nöjd ökat. Det är även viktigt att analysera datan som blir tillgänglig på grund av all denna teknik. I detta examensarbete undersöks det om det är möjligt att, med existerande teknik, bygga en pålitlig platform som kan extrahera data ur elektriska fordon, i detta fall scootrar, och skicka datan trådlöst till en webbklient. Det diskuteras även vilka andra möjligheter en sådan plattform öppnar upp för samt vad den kan användas till i framtiden.

För att plattformen skulle fungera med existerande teknik var det nödvändigt att lära sig om och integrera olika verktyg med varandra. Bestämma vilken lösning som skulle användas till vilken del samt att undersöka och kombinera dem till ett komplett system.

Examensarbetet resulterade i en fullt fungerande telematikplattform som kan extrahera, avkoda, skicka och visa data från en scooter. Plattformen består av en Raspberry Pi utrustad med en 4G- och GPSmodul, den använder sig av MQTT, Telegraf, InfluxDB samt Grafana. Den är designad för att vara skalbar för att kunna användas på flera scootrar samtidigt. Med några små ändringar kan plattformn även visa data från andra fordon än scootrar vilket innebär att detta projekt öppnar upp för många möjligheter vad gäller analys av data.

Keywords: IoT, MQTT, electriska scootrar, avkodning, trådlös överföring

Acknowledgements

This thesis project was performed at Chalmers University of Technology at the department of computer science and engineering. The practical work was made at the company Infotiv in Göteborg.

First we want to thank our supervisor Pedro Petersen Moura Trancoso for his support throughout the project. We also want to thank Infotiv for believing in us and letting us do our thesis work there. We give a special thanks to our supervisors at Infotiv Niels Boardman Jonsson, Olle Norelius and Hugo Frost, for their help and expertise. Last but not least we want to thank all the other thesis workers at Infotiv for much needed moral support.

Personally I would like to thank my family and friends for the patience, love and support throughout this thesis work. You truly gave me much needed strength when things were being difficult. - *Albin Brovina*

I would like to thank my family and my partner Johan for standing by me and giving me love and support when things didn't go as planned. - *Eva Bergsten*

Eva Bergsten, Albin Brovina, Göteborg, June 2020

Terminology

- RPi - ’Raspberry Pi’, a small computer. The one used in this project is of type 4B and runs the operating system Rasbian Buster February 2020.
- 4G - The fourth generation of broadband cellular network technology, i.e a protocol for transferring data wirelessly.
- SD - ’Secure digital’, a non-volatile memory card format.
- PCB - ’Printed circuit board’
- Arduino - An electronics platform based on easy-to-use hardware and software.
- GPS - ’Global Positioning System’
- IoT - ’Internet of Things’, the concept of connecting electronic devices to the internet.
- MQTT - ’Message Queuing Telemetry Transport’, a protocol for sending messages between devices.
- TCP - ’Transmission Control Protocol’, protocol used for most transmission across the internet.

Contents

List of Figures

1 Introduction

In this chapter the background for the project will be presented as well as the purpose, limitations and goals.

1.1 Background

As we put more and more technology into our vehicles, we also connect them to the internet to enable remote assist to the user and run troubleshooting on demand. Data from vehicles can be useful to analyze. For example, to make further development of the vehicle more efficient or monitor environment variables.

One kind of vehicle that has recently become important when analyzing data is the electric scooter. For example when renting a scooter it is necessary to know where it can be picked up and how much battery is left. When buying a scooter from a store or a factory, the equipment for collecting this data is not included. This means that if a private person or a company wants to analyze their own usage of the scooter, for example how fast they are driving and a history of where they have been, they need a platform for sending and receiving data wirelessly.

This thesis work is an investigation in whether it is possible to integrate existing tools to assemble a scalable platform for vehicle fleets to produce useful information and centralize that in a meaningful way.

1.2 Purpose

The purpose of this thesis work is to explore the possibility of combining existing telematic technology with existing electric scooters to develop a platform for electric vehicle fleet monitoring.

1.3 Project questions

Therefore some questions are to be answered:

- Is it possible to build a platform, with available tools, that extracts information from an existing scooter and sends it to a web client?
- Can the platform be scalable and used on several scooters at the same time?

- Does the platform open up for any further possibilities?

1.4 Demarcations

Due to cost limitations on hardware, the platform will only be implemented on two scooters. Previous work done by Infotiv on how to implement 4G will be used. The main focus will be collecting, sending and receiving the data and not on the user interface. Finally there will be no focus on optimization or making the system energy efficient.

1.5 Goals and requirements

With the help of the company Infotiv with headquarters in Göteborg, the platform developed will be implemented as a system for monitoring their fleet of electric scooters.

The aim of this thesis work is therefore to deliver a system containing of:
Hardware:

- A processing module consisting of a Raspberry Pi or similar.
- Attached to the processing unit, a module for wireless communication through an internet connection.
- Attached to the processing unit, a module for retrieving location information through GPS satellites.

Software:

- Back end system for hardware to database communication.
- Implemented scalability to enable unlimited amount of scooters connected.

Web:

- Basic graphical user interface using Grafana or similar.

The requirements for the system were set together with Infotiv and are as follows:

1. Hardware consisting of a computer with communication modules should be installed in a scooter.
2. Data being displayed on the scooter display and GPS location should be obtainable by the hardware module through UART at an acceptable rate.
3. Software back end and front end in a database to visual presentation should be able to display data.
4. Hardware should communicate with the web server with a maximum latency of two seconds.
5. Duplicating the system should be demonstrated by installing the system in two scooters.
6. Retrieved data should be stored permanently on an SD-card.

7. Display of final data in the front end should be with a latency of under 10 seconds.
8. At power loss or reboot, the system shall be able to re-initiate without user intervention.

2
Technical background

The scooters communicate with their RPi on the UART bus. The RPi then sends data using the MQTT protocol to a scooter specific topic. The Telegraf tool listens to the topics and then inserts the data into the InfluxDB database where it is ordered chronologically. Grafana displays the data stored in InfluxDB. See Figure 2.1 below.

The receiving end consists of a server computer at the company office. It runs a linux based operating system called Ubuntu and has MQTT, Telegraf, InfluxDB and Grafana installed. It acts as the centralized "broker" in the MQTT[1] protocol and is configured according to company standards with TLS[21] encryption in order to receive the data in a secure way. With the data being on the server database it is easily accessible for presentation on a company display.

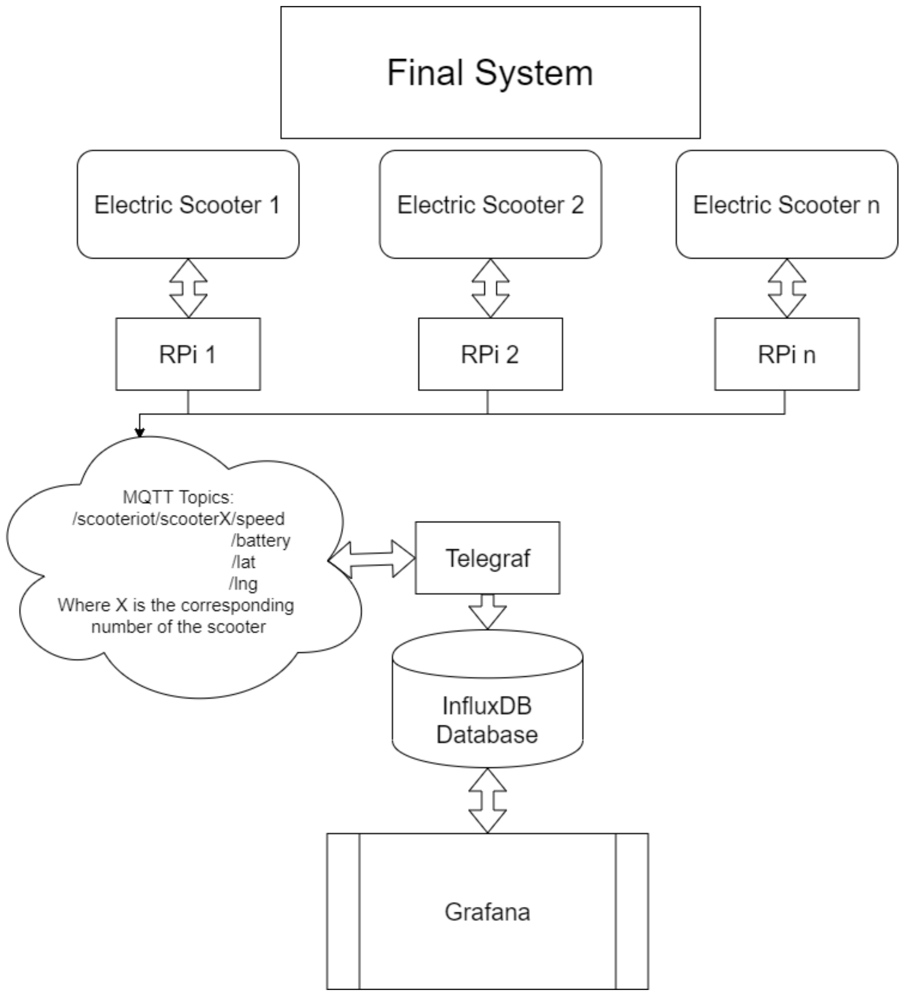


Figure 2.1: Flow chart of the system

2.1 Technical Description

Here follows the technical background and description of the software and hardware used in this project.

2.1.1 XIAOMI M365 Scooter

The electric scooters used in this project is of type XIAOMI M365, the reason for using these scooters is that they were the only ones available at Infotiv. The scooter consists of a system of three microcontrollers communicating with eachother on a UART half-duplex line. One of the microcontrollers is attached to the motor controller, another to the Battery Management System (BMS) and the third is on the bluetooth communication board.

A message on the bus always include a header indicating the beginning of a message, the length of the message and checksums to confirm message integrity. One byte in the message indicates destination of that message. Between which microcontrollers the message is being sent.

2.1.2 MQTT

Message Queuing Telemetry Transport, MQTT, is a lightweight network protocol for sending messages between devices [1] based on the TCP protocol. It is also designed for connections with remote locations where the network bandwidth is limited, which fits this project.

The protocol consists of three main parts, a broker, a publisher and a subscriber. A client can either be a subscriber, a publisher or both. The publisher publishes one or more topics to the broker, the subscriber can then listen to these topics and send a reply. The system can have multiple publishers and subscribers that communicates with each other.

The broker acts as the middle man in the ’conversation’ between the publisher and the subscriber. All messages from both sides go through the broker. The broker also monitors the status of all connected clients, eliminates vulnerable client connections and tracks all client connections.

2.1.3 Telegraf

Telegraf is a tool for collecting and sending metrics to and from different systems, databases or IoT-sensors [2]. It is a plugin-driven service that can collect wide arrays of inputs and write to outputs. Telegraf is a standalone service that can be used on any system and therefore has no need of other external packages.

2.1.4 GPS/4G module

Global positioning system is a navigation method that calculates a geographical position by using the satellites orbiting the earth [3]. When the GPS receives a signal from at least three satellites at the same time it can calculate its position on earth. The position is where the signals from all three satellites intersect, see Figure 2.1 below. The RPi is connected to a 4G module that also has a GPS antenna.

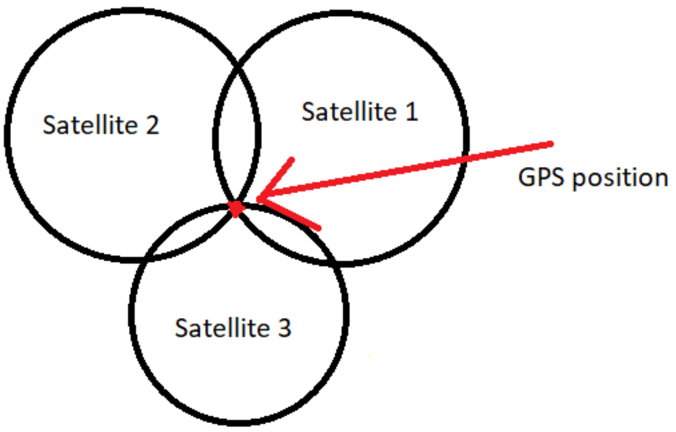


Figure 2.2: Simplified visualisation how a GPS position is obtained from satellites.

2.1.5 UART

UART stands for Universal Asynchronous Receiver/Transmitter [4] and is the bus the scooter is using to send data. The UART bus converts parallel data to serial data by taking one byte and sending it bit by bit. Another UART receives the data bits and puts them back together to one byte parallel data. The receiving UART knows which bits belong to which byte by looking at the start and stop bits that are sent with the data. The data can be sent both ways, meaning that one UART can both transmit and receive data.

Half-duplex UART [5] is when data is transmitted according to the UART procotol but on one wire and in one direction at a time.

2.1.6 InfluxDB

InfluxDB is an open source database that is specifically designed for time series data [6]. The database can handle high write rates and query loads which makes it ideal for storing large amounts of timestamped data. InfluxDB requires no external dependencies but supports plugins for other data protocols.

2.1.7 Grafana

Grafana is a graphical tool for displaying and analyzing metrics and other data [8]. Grafana works dynamically with a variety of databases to turn the data stored there into, for example, graphs for easier visualization. Grafana can show both live feed data from the database or all data from a chosen time span. There are different plugins that can be used to display data in different ways such as graphs, tables, GPS positions on a map and much more.

3 Methods

This chapter presents the methods used for accomplishing this project.

3.1 Different methods used

The Engineering Method [15] was implemented in a collaborative sense with Infotiv. The idea and the concept phase were discussed early on, before the thesis work started. Then followed the design process while research was being conducted on existing solutions and different communication protocols. The company aided the project team in the early stages with ideas about the system design and options for software/hardware but toward the design/development process of the prototype the work was more independently executed by the team.

Scrum [14] is "a simple framework for effective team collaboration on complex products". It was used in this thesis work as an iterative method with daily reflection and weekly planning/feedback. Every day, what was currently being worked on and what had been done the previous day was presented in a smaller group of people working on different projects at the company. Any obstacles preventing development could be addressed and discussed. The daily feedback was unfortunately not possible to maintain during the ongoing global pandemic but the weekly ones were continued. Every week there were also a short meeting where progress and current issues were documented.

Disciplinary method. The Pomodoro Technique [16] is a technique for working effectively. The basic concept is a strict working schedule with intervals of 25 minutes working, 5 minutes break and for every fourth working session there is a longer break. Often during, especially software development, there is a risk of losing focus due to long periods of sitting down and looking into a screen. This technique helped by promoting taking short breaks to stretch, drink some water and become motivated for another session of focusing on tasks.

3.2 Gates

The end goal was divided into 5 main parts, here called gates, to help structure the work. Each gate was set to a date where that part of the project had to be done. These gates were then divided further into smaller blocks to make the project more perspicuous. The gates were decided together with Infotiv and were as follows:

- **Gate 1: Project description approved**
  Developers, Infotiv and Chalmers have all agreed on a Project Description and the development can begin.
- **Gate 2: Research done and reused code tested**
  Already developed foundation for Telematic platform will have been reviewed and adjusted to suit the scooter which has been examined. Developers should have succeeded in extracting data from the scooter to present that the code works with the hardware. If there is unexpected delays on either part of the prerequisites for this gate to be reached in time, this will be communicated and Gate 3 postponed if need be.
- **Gate 3: Basic functions testing and data presentation**
  Developers have figured out how to retrieve, send and receive data from the scooter to the database for visualization and presentation. A functional front end has been developed and can display the data.
- **Gate 4: Final Presentation Chalmers**
  Final presentation at Chalmers for the developers.
- **Gate 5: Final Presentation Infotiv**
  All project goals have been met and a scooter is equipped with hardware and ready to go for a test ride. Developers demonstrate each part of the system for everyone and an in-depth presentation of the HW/SW is held.

A more detailed time plan can be found in appendix 1.

3.3 Research

To establish a good foundation for decision-making, the project started with a lot of research on which hardware and software to choose for implementation in the system.

Research was also conducted on previous Infotiv projects and open source public projects regarding parts of the system. All research is presented below.

3.3.1 Hardware

It was decided early in the project to use a RPi for extracting and sending the data. The RPi was chosen because it has on board memory, a great community and is rather fast which is important for further development. It also fit the workflow of Infotiv better than the other alternatives. The 4G and GPS module was chosen because it had been confirmed to work with the RPi and tests had been done on the same type of module earlier at Infotiv.

However, an investigation in hardware options before coming to a final decision was made. The research was based on the hardware requirements and the overall software functionality. One consideration that was taken into account when looking for suitable hardware option was the physical dimensions of the unit itself. It was desirable to have a device which could fit inside of the frame of the scooter.

The Icarus IoT board [9] turned out to be a very promising small size contestant since it provided a 4G internet connection, GPS and a fast processing unit.

Two of the drawbacks of this unit in particular were that it had to be programmed in ZephyrOS and that the community support was smaller compared to the RPi. Adding to the drawbacks was a steep learning curve which combined with the reasons for going with the RPi led to the conclusion that this was a good option and that fitting it inside of the frame would be desirable, however the time needed to learn a new programming language was estimated to not suffice. Lastly, the Icarus board was over budget.

Another option that was looked into was using a development board from Arduino family but since most of them required memory modules and operated at relatively low clock frequencies they had to be excluded. However, the Arduino was used for the initial serial bus readings coming from the scooter to confirm communication protocol functionality.

The final decision was to use a RPi[17] equipped with peripheral units for 4G and GPS features. A 4G "hat"[18], 4G module[19] with a tripple antenna[20] and a SIM-card. A cable for splitting the UART for half-duplex implementation was needed and was constructed with a high speed switching application signal diode for general purposes, the 1N4148 and a ceramic 1/4W resistor with a value of 120 or 220 ohms. Both values are viable options. See figure 4.9

3.3.2 Software

When choosing which method to use for sending the data from the scooter to the database, some different protocols were compared. The final choice, MQTT, was chosen because of its low bandwidth footprint which is good for minimizing data rates through 4G. Additionally because of being publish/subscribe based which is a messaging method developed from the IoT perspective with mobile light weight application in mind. It was seemingly rather easy to get up and running. Its' feature to handle multiple clients at the same time was crucial since the platform was required to be scalable.

A lot of research went into which database solution to use. InfluxDB was finally chosen primarily because it is open source, operates a time series model and is push based. Time series means it stores the data in the order it enters the database[7]. Therefore centralizing time stamping from the unit transmitting data, in this case the scooter. The timestamp is needed when displaying the data in chronological order to track certain parameters. Push based means data has to be pushed into the database unlike a pull based model works by pulling data to the database on set intervals or other events. Finally InfluxDB is very compatible with Grafana and therefore the two in combination were chosen.

Table 2.1 shows what the investigation of different databases led to. The database was required to be open source, be able to receive data frequently, have good documentation available and have known synergy with MQTT.

Database	**Most important notes**
InfluxDB	Open source, Written in GO, SQL-like queries, push based, time series model
Prometheus	Written in GO, pull based, time series model, XML support
CrateDB	No free version
MongoDB	No free version
RethinkDB	Seemed overall pretty good, good python integration and community
SQLite	Public domain, low memory usage, reliable, support available
Apache Cassandra	Only free trial

Table 3.1: Summary of investigation results

The small tool Telegraf is required as it pushes data from MQTT to InfluxDB and is how it is used in this project. InfluxDB cannot subscribe to MQTT topics by itself.

3.3.3 Previous work

During the research process, different existing solutions for decoding the scooter data stream were looked into to get an understanding of how they work and how they are structured. This research resulted in the finding of a protocol, see appendix 2, for what data could be extracted from a scooter and how to decode it. The protocol was made by Camilo Ruiz [10]. Arduino code was used to confirm the protocol as described in section 4.3.

4 Implementation

Here follows a description on how every part of the project was implemented, from start to finish, in order to reach the goals.

4.1 Software design

A software design was drafted to better understand and describe exactly how the platform was planned to work. This was done in order to speed up the programming process and to keep on track.

The main function, Figure 4.1, describes how the platform is supposed to work, from scooter to Grafana.

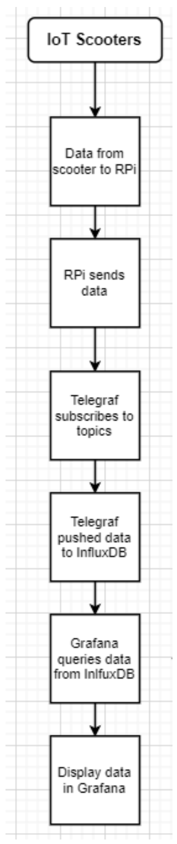

IoT Scooters
Data from scooter to RPi
RPi sends data
Telegraf subscribes to topics
Telegraf pushed data to InfluxDB
Grafana queries data from InlfuxDB
Display data in Grafana

Figure 4.1: Main function of the platform

The RPi extracts data from the scooter. This is accomplished as shown in Figure

4.2. The RPi sends a request to the scooter to send a message including the desired data. The scooter then sends a message back containing that data, in this case speed and battery.

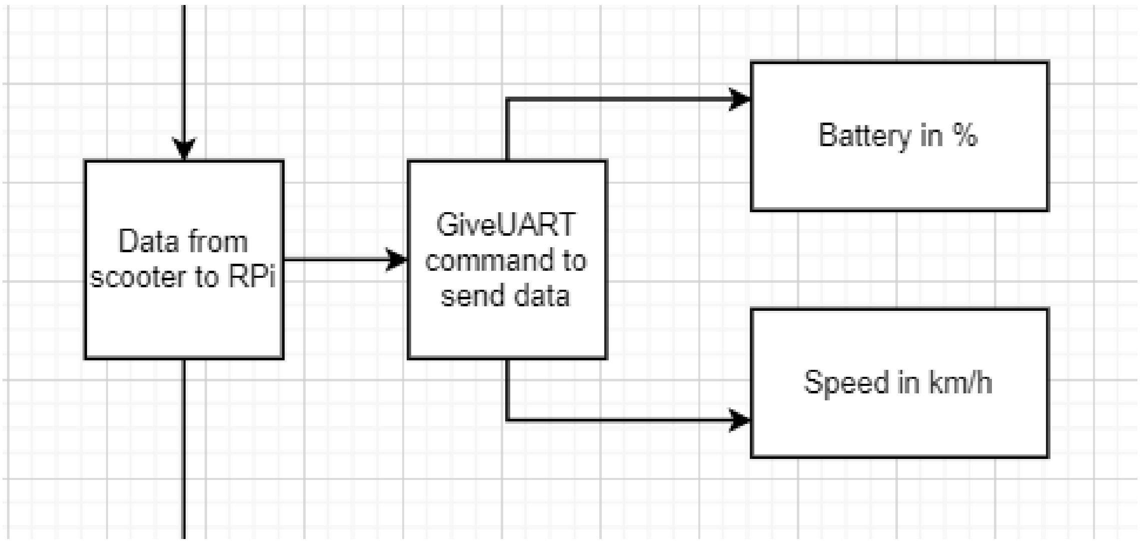


Figure 4.2: How the data is extracted and saved on the RPi

The data is sent as explained in Figure 4.3 below. The RPi creates different topics for the speed, battery and GPS coordinates as shown in Figure 2.1. The RPi connects to the 4G network and publishes velocity and battery percentage on separate topics to MQTT. Telegraf subscribes to the topics and pushes the data into InfluxDB. How the data is sent is explained further in chapter 4.4.

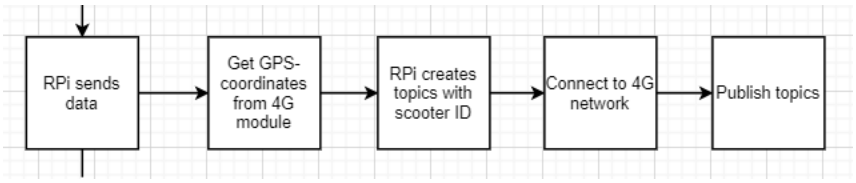

RPi sends data
Get GPS-coordinates from 4G module
RPi creates topics with scooter ID
Connect to 4G network
Publish topics

Figure 4.3: How the Rpi sends data

Figure 4.4 below shows how Grafana gets the data from InfluxDB depending on what type of data is supposed to be displayed, history or live feed. The time span is chosen by the user from a menu in Grafana.

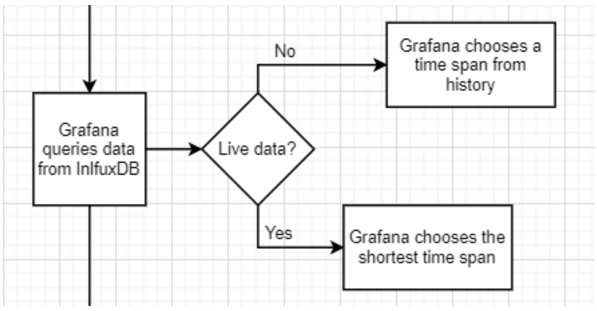

Grafana queries data from InlfuxDB
Live data?
No
Grafana chooses a time span from history
Yes
Grafana chooses the shortest time span

Figure 4.4: How Grafana chooses the data from InfluxDB

Every scooter gets it data displayed in Grafana. This is done as shown in Figure 4.5, a specific scooter is chosen and the data is presented with the help of a speedometer, a bar graph and a map.

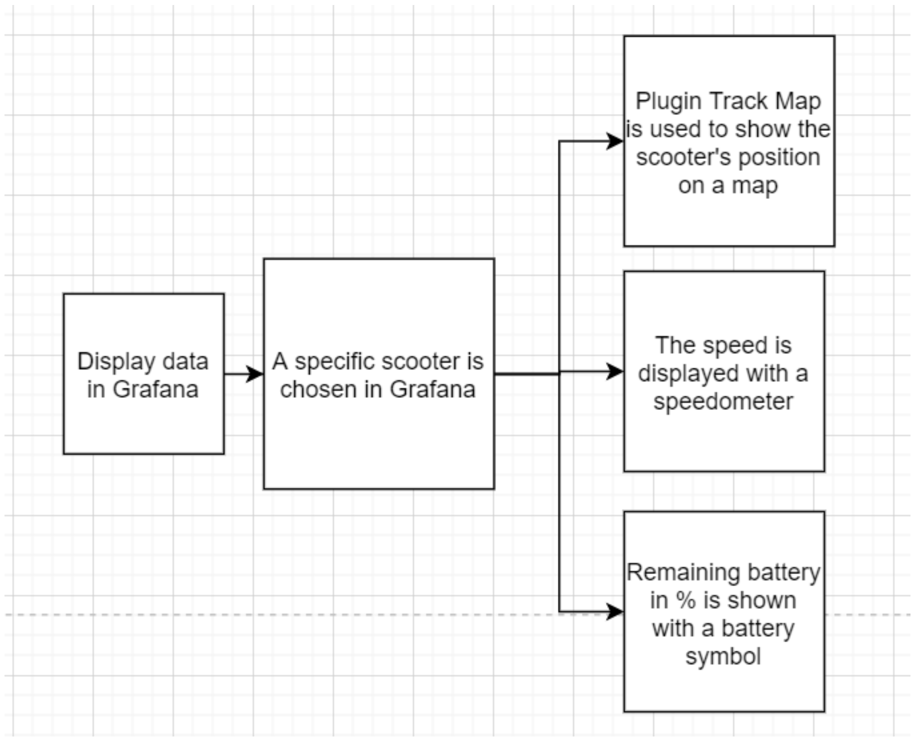

Display data in Grafana
A specific scooter is chosen in Grafana
Plugin Track Map is used to show the scooter's position on a map
The speed is displayed with a speedometer
Remaining battery in % is shown with a battery symbol

Figure 4.5: How Grafana displays the data

4.2 Phase One - Protocol confirmation and setup

How to receive the serial data coming from the scooter needed to be tested. This was accomplished by confirming the scooter protocol [10] found during research. As explained in chapter 2, the platform was supposed to consist of a RPi. Since the developers had previous experience with Arduino and serial communication on that device, an Arduino was connected to the scooter and a code for reading the serial port, see appendix 3, was tested. With this code the scooter protocol could be validated and relied on for further development. The scooter was continuously transmitting default messages in hexadecimal numbers. The messages consisted of a destination address, contents and a checksum to confirm the integrity of the message.

The contents of the message are different scooter variables. Two of these variables in the default message were the brake and throttle values. When actuating the brake or the throttle on the scooter it was clearly visible that the expected variables changed.

4.2.1 Front end

Grafana was intended to be running on the centralized server to simulate that the software was installed locally. Sample data was displayed through different preconfigured visualization tools to give a first draft picture of how the end result should look like.

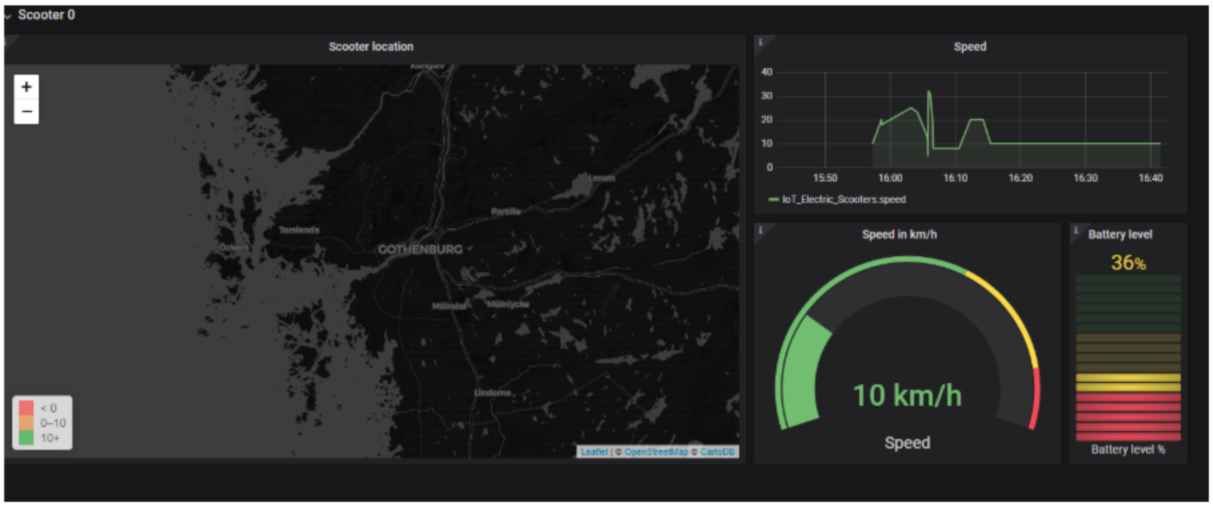


Figure 4.6: First draft of front end in Grafana

4.2.2 4G implementation

The 4G module and the triple antenna was attached to the RPi. A SIM card with a prepaid data plan was inserted into the hat. To confirm that a reliable 4G internet connection could be established the PPP installer guide [11] was used and after WiFi connection was disabled the 4G symbol appeared and an internet connection could be confirmed through the web browser.

4.2.3 GPS implementation

The GPS feature of the 4G module had to be tested. An example python script was used for this purpose and the RPi was brought outside to be exposed to open air for better connectivity. The test was successful and GPS position was obtained.

Figure 4.7: GPS tested for coordinate data

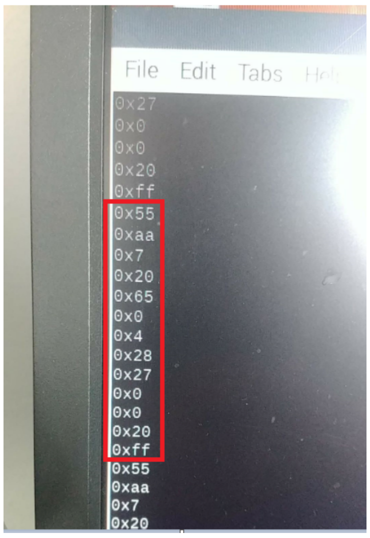
4.2.4 Store data locally

All the data that is sent from the scooter to the RPi was saved locally on the RPi's SD card. This is to make sure that no data will be lost due to internet connection loss and for data logging purposes. This was accomplished by implementing a small Python script, that collects data from the scooter and then writes it to files on the SD card. The data is initially stored in three different files, one for GPS coordinates, one for the battery status and another one for speed parameters.

4.3 Phase Two - Raspberry Pi Implementation

Since the Arduino had only been used to test and validate the scooter protocol, it had to be replaced with a RPi. The scooter was connected to the RPi according to the explanation in section 4.3.1. A simple Python script, see appendix 4, was implemented that listened to the serial data communication on the scooter's bus. No data was written to the bus at this point. The purpose was to read the data and confirm that the protocol could be used with a RPi as well. The test showed that the scooter was transmitting the default messages according to the protocol.

Figure 4.8 below shows the output of the default messages sent from the scooter where the hexadecimal numbers corresponds to values sent from the scooter according to the decoding protocol, see appendix 2. The message goes from 0x55 to 0x20. 0x55 marks the start of a new message, 0x55 and 0xaa are fixed headers that signals that a message is being transmitted and 0x7 equals the length of the message. Then follows other values sent from the scooter where 0x28 and 0x27 are the throttle and break values respectively. When actuating the break or throttle on the scooter is was clearly visible that both of the values behaved as expected, increasing and decreasing. It was confirmed that the UART was configured correctly.



```
0x27
0x0
0x0
0x20
0xff
0x55
0xaa
0x7
0x20
0x65
0x0
0x4
0x28
0x27
0x0
0x0
0x20
0xff
0x55
0xaa
0x7
0x20
```

Figure 4.8: Output showing the extracted default data from the scooter in hexadecimal values

4.3.1 Connecting the scooter to the RPi

The RPi is connected to the scooter as seen in Figures 4.9 and 4.10 below. Ground on the scooter is connected to ground on the RPi. To be able to recieve data from the scooter and read it on the RPi, a split wire is connected between the scooters UART bus data output and the RX and TX pins on the RPi. The RX and TX pins on the RPi are connected to each other with a resistor and a diode in between. This creates a conversion needed for half duplex one wire UART data communication.

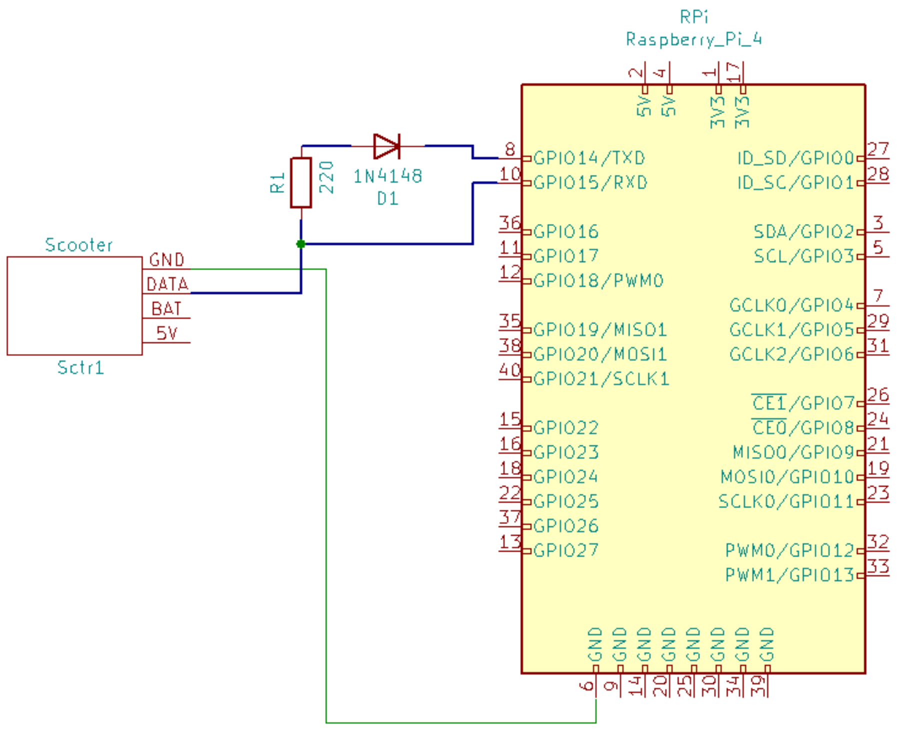


Figure 4.9: Schematics of how the scooter and RPi are connected

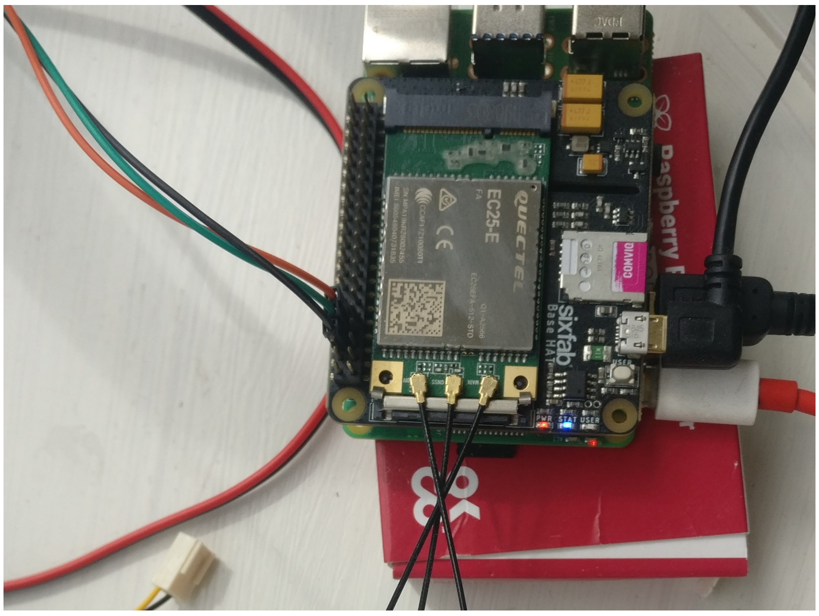


Figure 4.10: How the scooter and Rpi are connected

4.3.2 Decoding

The next step was to interpret the data stream coming from the scooter and "talk" to the scooter. That means that in order to retrieve the information, code was required that enabled writing data to the bus.

4.3.3 Retrieval of desired data

Since the speed and battery status were not sent by default, the scooter had to be "told" to send the desired data according to the protocol. This was done by adding a function to the python script. The function sends a specific message to the scooter containing information about what data is requested, the scooter then sends a reply message containing that data. The decoding script can be seen in appendix 5.

During implementation of this functionality all went well except at first the data was expected to be located in the first retrieved message following a request. However this was not the case, and after some trial and error the system was adapted accordingly.

The raw data was processed and calculated to presentable values. Battery level to a percentage and velocity to km/h.

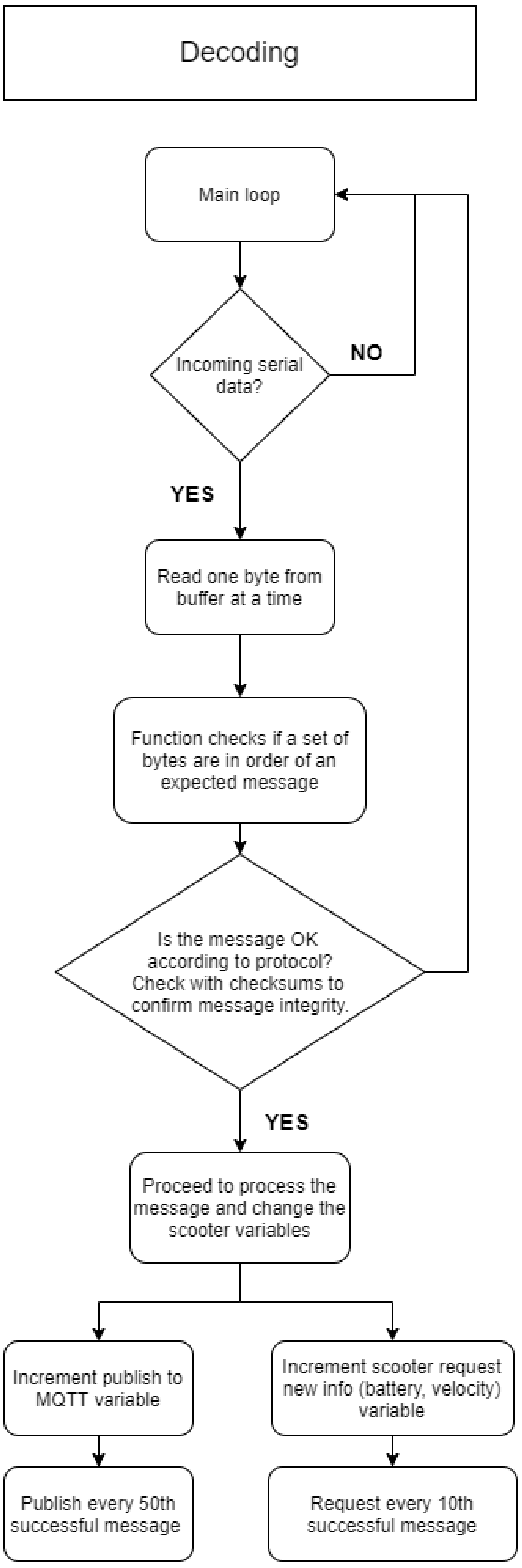
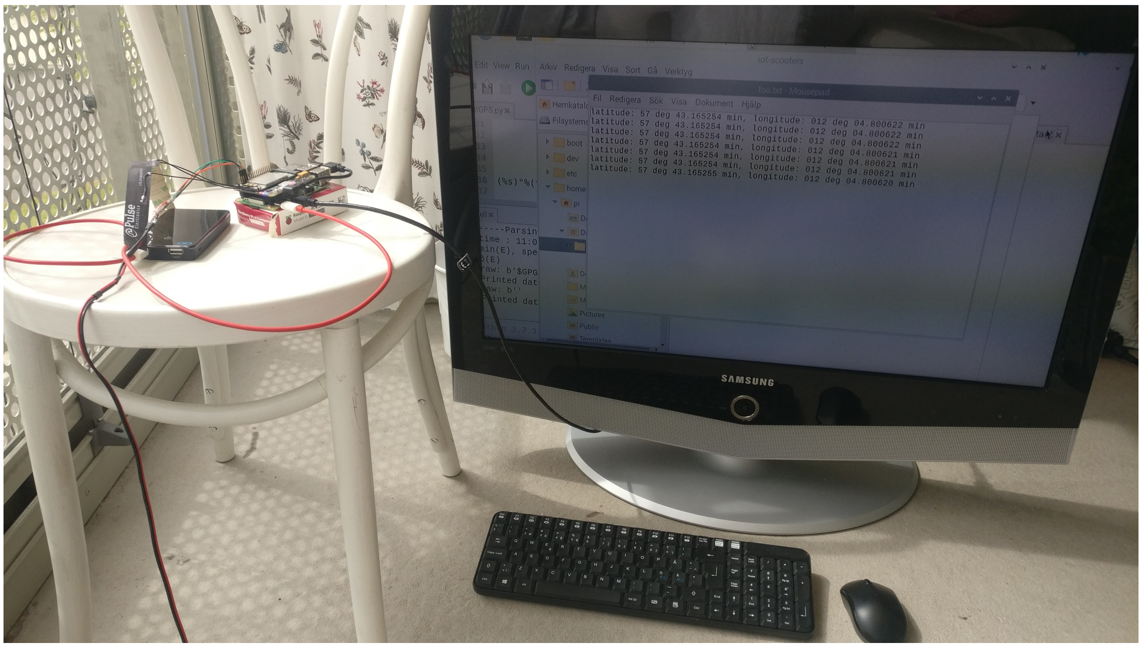

Decoding
Main loop
Incoming serial data?
NO
YES
Read one byte from buffer at a time
Function checks if a set of bytes are in order of an expected message
Is the message OK according to protocol? Check with checksums to confirm message integrity.
YES
Proceed to process the message and change the scooter variables
Increment publish to MQTT variable
Increment scooter request new info (battery, velocity) variable
Publish every 50th successful message
Request every 10th successful message

Figure 4.11: Decoding script flowchart

4.3.4 Start on boot

Since the RPi had to be connected to a screen and a power cord it was cumbersome to do the iterative programming outside. However the GPS module had to be outside to be able to pick up signals from the satellites, therefore a test of the system was executed on one of the project members' balcony. See Figure 4.12.



```
latitude: 57 deg 43.165254 min, longitude: 012 deg 04.800622 min
latitude: 57 deg 43.165254 min, longitude: 012 deg 04.800622 min
latitude: 57 deg 43.165254 min, longitude: 012 deg 04.800622 min
latitude: 57 deg 43.165254 min, longitude: 012 deg 04.800621 min
latitude: 57 deg 43.165254 min, longitude: 012 deg 04.800621 min
latitude: 57 deg 43.165254 min, longitude: 012 deg 04.800621 min
latitude: 57 deg 43.165255 min, longitude: 012 deg 04.800620 min
```

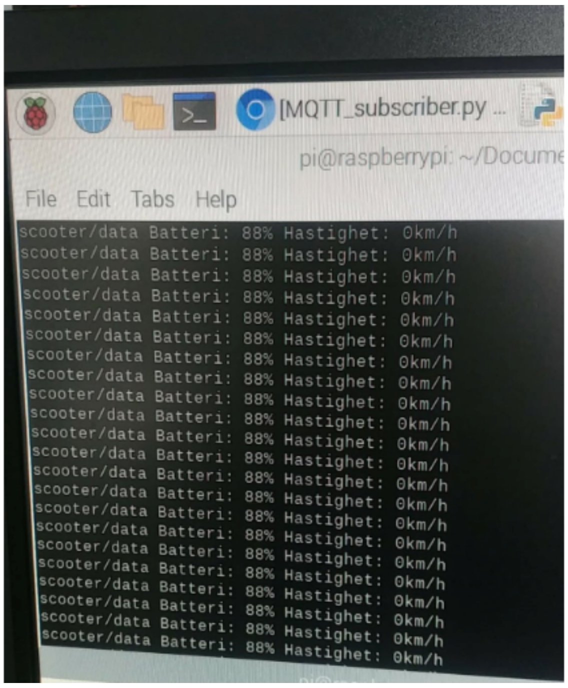
Figure 4.12: Balcony test rig

During development the system had been powered by a power adapter but was now in need of being mobilised. A way of starting the scripts without user interference was researched and implemented. Initially this was implemented using the RPi crontab[12] utility, which is a time based job scheduler. The scripts were to start on boot up. Crontab worked but was not stable enough which led to finally using the "rc.local"[13].

4.4 Phase Three - Telecommunication

A test to send the correct values from the scooter, i.e battery status, speed and GPS position, using the MQTT protocol via 4G to another client was set up.

This test was performed by one of the developers taking a test ride outside on the scooter, with the RPi and 4G/GPS module connected and publishing data to MQTT. The other developer was sitting inside on another RPi subscribing to the published topics. This resulted in successfully retrieving live feed data being sent from one client to another. The subscriber could in real time observe the speed, battery status and GPS position of the publisher. It was observed that the latency between the hardware and the web server was with good margins below required threshold of two seconds.



```
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
scooter/data Batteri: 88% Hastighet: 0km/h
```

Figure 4.13: Output showing battery and speed being sent from one client to another

However in this test it was also discovered that when cutting power to the system, the function that stores the data locally on the SD card didn't work as intended. The data was being corrupted due to sudden power loss from turning the RPi off by abruptly pulling the cord.

Another discovery was that sending data from the scooter to InfluxDB and then displaying it in Grafana would take longer time than was set in the requirements.

The result from this test was discussed and evaluated at the Gate 3 meeting. Actions to improve the system were taken and two new requirements were added, see point 7 and 8 in section 1.4.

4.4.1 Scalability

One of the goals for this project was to make the platform scalable for usage of more than one scooter at a time. To make this possible each scooter had to send a unique message or ID to the database, so when displaying the data in Grafana it would be clear which data belonged to which scooter. This was accomplished by making each scooter publish a unique topic when sending the data. For example, the first scooter would send topics that looked like scooter1/battery or scooter1/speed, the second scooter would then send topics like scooter2/battery, and so on. This way, when the Telegraf receives the data it will know which scooter sent the data by looking at the topics. The data can then be stored and ordered in the database according to which scooter sent it and Grafana can display data from multiple scooters simultaneously.

4.4.2 Telegraf

Telegraf is used as described in section 2.2.2 as a middleman plugin between MQTT and InfluxDB. Telegraf is set to subscribe to the topics published by each scooter's RPi. Telegraf inserts the data into the database, this enables Grafana to use that data for visual presentation in real time.

4.4.3 Attaching the RPi on the scooter

At first the scooter was temporarily equipped with the RPi with its 4G hat and antennas and a powerbank. See Figure 4.14.

Although for the platform to be attached safely and look better, the RPi, 4G module and the coupling deck were installed in a coupling box. The box is strapped to the scooter below the handlebars with cable ties. The cables were drawn from the scooters bluetooth board out trough the same hole as the brake wire goes through and in to the coupling box, see Figure 4.16 below. The reason for this solution is that Infotiv did not want a permanent solution, or for any holes to be drilled in their scooters to attach the coupling box.

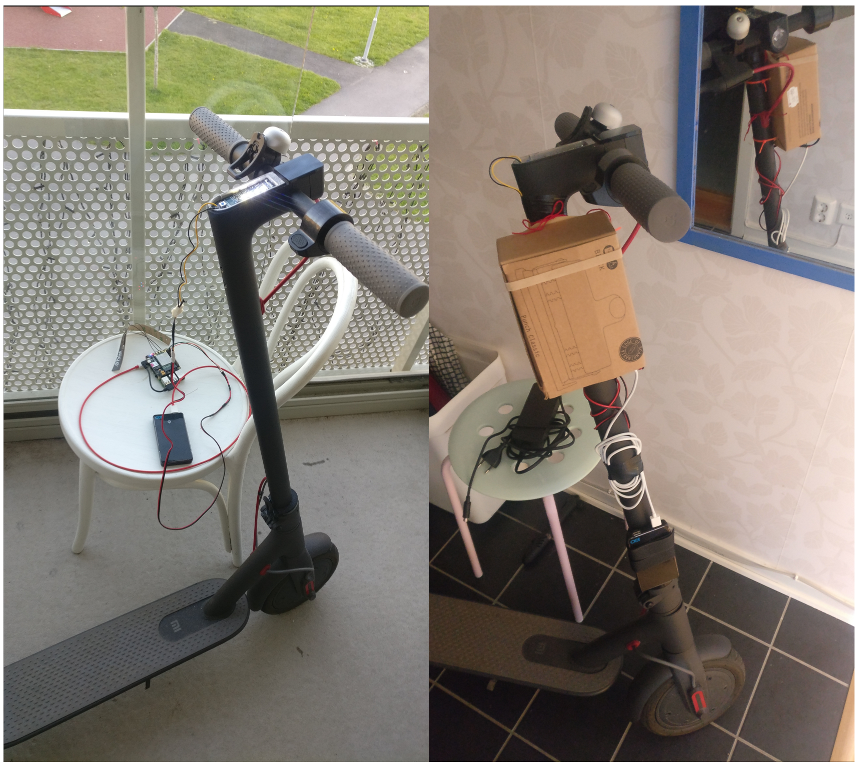


Figure 4.14: How the temporary attachment looked like. Very much prototype-looking.

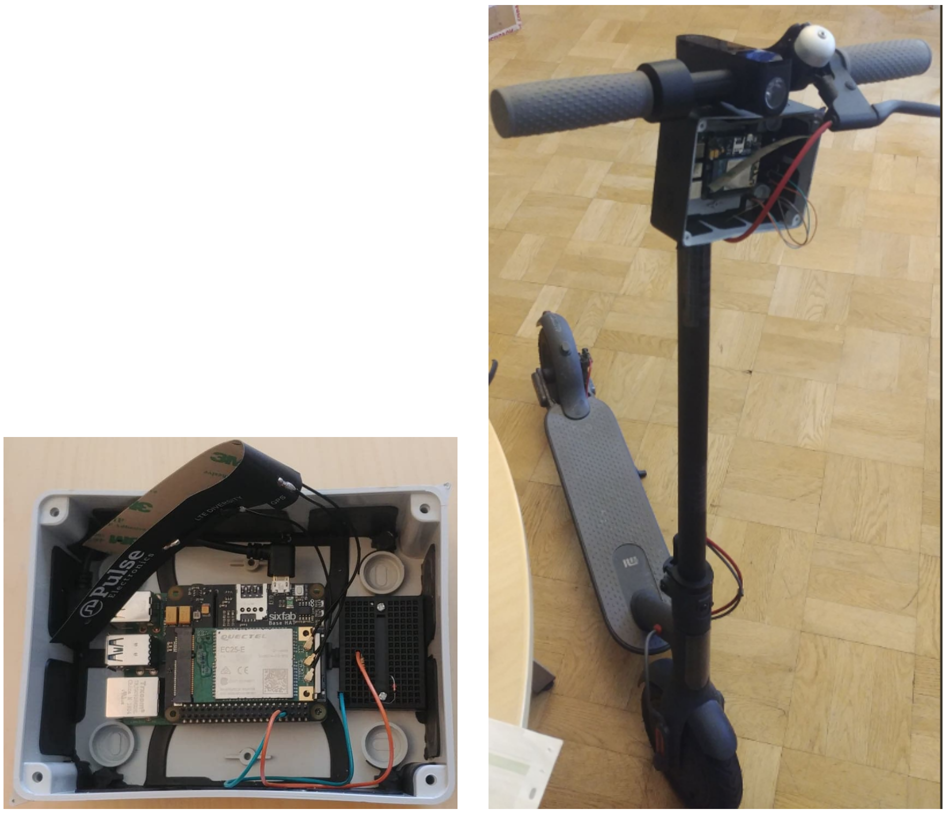


Figure 4.15: How the RPi is mounted on the scooter

4.5 Phase Four - The Final Version

The final test was made with two scooters to test both the whole chain and scalability. This test was successful and speed, battery status and GPS positions from two scooters could be sent to InfluxDB and then displayed in Grafana, see Figure 4.16 below. This test showed that the platform works as intended and described in Figure 2.1. This final test also included the systems ability to re-initate itself on powerloss or reboot, which worked exactly as planned.

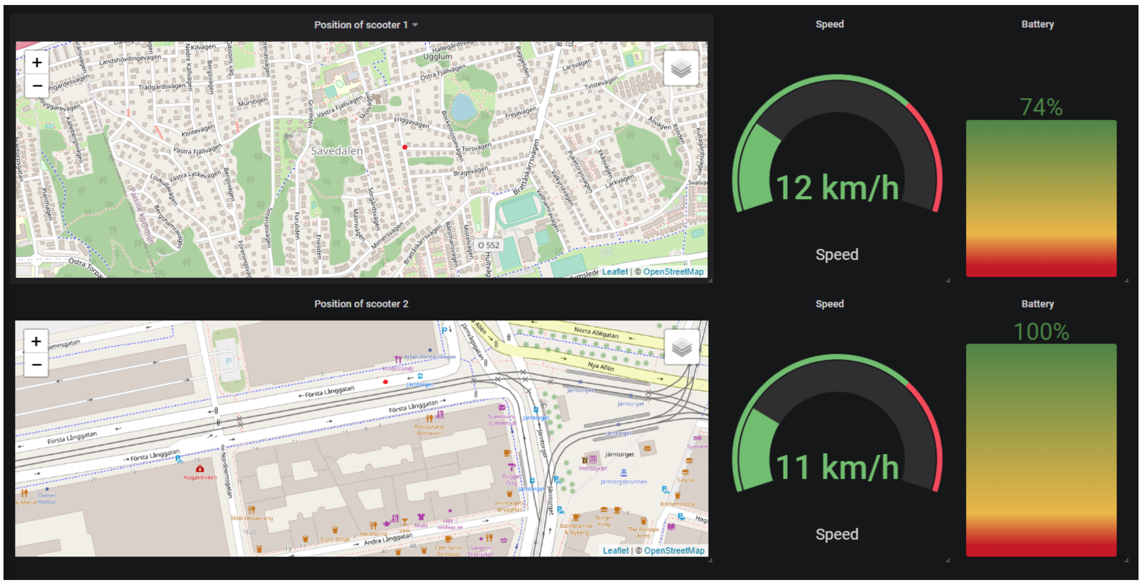


Figure 4.16: Data from two scooters displayed in Grafana.

5 Results

The project resulted in a fully functional IoT-platform that can extract serial data from an electric scooter and send it wirelessly to a database and then display the data in a graphical user interface. The majority of the goals and requirements set up in the beginning of the project were met. Meaning it is possible to combine existing technology into a system for electric vehicle fleet monitoring.

The back end can extract data from the scooter and decode it, it can also send the data to the database over MQTT via 4G. The data can then be displayed in a basic front end made in Grafana.

In the final test it was confirmed that it takes a maximum of 9 seconds to send the data and display it in Grafana.

In phase 3, the rate at which the hardware communicates with the web server was tested and confirmed to be below two seconds.

Scalability is confirmed by two scooters equipped with a Rpi and a 4G and GPS module. Both are fully functional and can extract data from the scooter and find a GPS position.

The system can reinitiate itself on power loss or reboot.

During testing it was discovered that the GPS module had trouble finding the required satellite fix to get an exact position. This resulted in the module believing it was in Kungsbacka, around 26 Km south of Göteborg which was its actual position.

It is not implemented any functionality for storing the data on the SD card since the discovery of data corruption due to sudden power loss.

6 Conclusion

This chapter presents a discussion of the result and the questions asked in the beginning of the project. It also presents future work that could be done to improve the platform and how this project relates to ethics and sustainable environment.

6.1 Discussion of questions

- Is it possible to build a platform, with available tools, that extracts information from an existing scooter and sends it to a web client?

If the right tools are used in an efficient way it is possible to build such a platform. However it takes a lot of research and decision making on what hardware and software would be best to use.

- Can the platform be scalable and used on several scooters at the same time?

As said in chapter 5, the system has only been tested on two scooters. However it should not be a problem to add more scooters to the system as the python scripts should work on any scooter of the same model. Furthermore, as MQTT is designed to use more than one publisher, sending data from multiple scooters at the same time should work as long as each scooter publishes a unique topic. The subscriber can then listen to the different topics, store the data in InfluxDB and display it in Grafana.

- Does the platform open up for any further possibilities?

Yes, this platform opens up for a lot of different possibilities. The data extracted from the scooters in this projects is limited to speed and battery status, but the scooter can send a variety of different values. For example the scooter can measure cell voltage and the temperature of the battery, it also has an odometer. All these different values can be used in different projects in the future.

6.2 Future work

This platform can be improved in many ways, below follows some suggestions for future work.

The python scripts are not optimised. An optimisation of the code could speed up the process of extracting and sending data. Storing the data on the SD card can most probably be solved by applying routines for safely writing to the SD card and then test robustness against power loss. A function that stores the data in a buffer and then writes it to the SD when the buffer is full could also be implemented. This would reduce the number of writes to the non-volatile SD card which would give it a longer life.

More data could easily be extracted by sending more request messages to the scooter bus in a similar way of how it is already done. The publishing to the database can be tested for speed and robustness.

One of the requirements for this project was to install the RPi and 4G module on the scooter. This was done as explained in section 4.5.3. However this solution could be improved by designing and 3D-printing a case that fits all the equipment better than the coupling box. If made correctly the 3D printed case would also be better suited for outdoor usage and be attached to the scooter in a good-looking way.

Another improvement could be connecting the RPi to the scooters internal battery. That way when the scooter is turned on, the RPi will automatically turn on as well and start collecting data without user interference. It would also mean that it would not be necessary to use a power bank for the RPi, which is currently needed.

Alternatively a platform could be implemented using the device most people carry around today. The smartphone already has GPS functionality which could be combined with scooter data sent through bluetooth and publishing the data from the phone instead of from the scooter. Then the scooter would be detached from the sending part of the system, this could be benefitial if the system is to be used by different users and could also be more energy efficient.

Another approach to the issue is to completely replace one of the scooters' microcontrollers, for example the bluetooth board, then the added system would perhaps fit in the frame and more options and customization could be added. A potentially viable microcontroller similar to the one used in this work is the Raspberry Pi Zero which is smaller.

6.3 Ethics

When someone is moving around, either on foot or in a vehicle, where they have been and when they were there is their personal information. This means that when the RPi is sending data to the database it has to be encrypted and with a secure connection. In this case, the MQTT broker helps with encryption and makes sure that only the subscriber can read the data sent from the publisher.

Furthermore, the scooter should be password protected to prevent unauthorized people from accessing the RPis' internal SD card where all extracted data could be stored. This would also prevent unauthorized people from sending data back to the database as the the scripts start running as soon as the RPi is turned on.

6.4 Environmental impact

The development of this platform is from an environmental aspect a good and sustainable project. The platform itself has no effect on the environment, but using it makes it easier and more fun to ride electric scooters. This may lead to more people choosing to ride their scooters to and from places instead of taking the bus. This could reduce the amount of carbon dioxide in the air due to less exhaust from buses.

However, if this platform is to be used on a large amount of scooters it can have a bad impact on the environment. Since each scooter will need to be equipped with a RPi and a 4G and GPS module these components will need to be manufactured in a bigger scale. This is not good for the environment since they contain a lot of rare metals. Also, these electric components are seldom recycled.

Bibliography

[1] MQTT.org, "Frequently asked questions" [Online]. Available: http://mqtt.org/faq [Used: February 2020]

[2] Influx data, "Telegraf" [Online]. Available: https://www.influxdata.com/time-series-platform/telegraf/ [Used: May 2020]

[3] physics.org, "How does GPS work?" [Online]. Available: http://www.physics.org/article-questions.asp?id=55 [Used: May 2020]

[4] Circuit Basics , "Basics of UART communication" [Online]. Available: https://www.circuitbasics.com/basics-uart-communication/ [Used: May 2020]

[5] Half Duplex UART, "Universal asynschronous receiver transmitter" [Online]. Available: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter [Used: June 2020]

[6] Influx data, "InfluxDB 1.8 documentation" [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/ [Used: February 2020]

[7] Time Series Database, "Time series database" [Online]. Available: https://en.wikipedia.org/wiki/Time_series_database [Used: June 2020]

[8] Grafana Labs "What is Grafana" [Online]. Available: https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/ [Used: February 2020]

[9] Actinius "Icarus IoT Board" [online]. Available: https://www.actinius.com/icarus [Used: Mars 2020]

[10] GitHub "M365-BLE-PROTOCOL" [Online]. Available: https://github.com/CamiAlfa/M365-BLE-PROTOCOL?fbclid=IwAR21xJmMdB8FCzFbmjA0DF9jkIGYNuG4XDlfjfoiaRZJAVwOtzV2gH2fa-M [Used: February 2020]

[11] Internet installer "PPP installer for Sixfab Shield/Hat" [Online]. Available: https://sixfab.com/ppp-installer-for-sixfab-shield-hat [Used: June 2020]

[12] Crontab "Cron" [Online]. Available: https://en.wikipedia.org/wiki/Cron [Used: June 2020]

[13] RC Local "Rc.local" [Online]. Available: https://www.raspberrypi.org/documentation/linux/usage/rc-local.md [Used: June 2020]

[14] Scrum Methodology "What is Scrum?" [Online]. Avaiable: https://www.scrum.org/resources/what-is-scrum [Used: June 2020]

[15] Engineering Method "Engineering Method" [Online]. Available: https://sites.tufts.edu/eeseniordesignhandbook/2013/engineering-method/ [Used: June 2020]

[16] Pomodoro Disciplinary Technique, "The Pomodoro Technique" [Online]. Available: https://francescocirillo.com/pages/pomodoro-technique, [Used: June 2020]

[17] Raspberry Pi 4, "Raspberry Pi 4" [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/ [Used: June 2020]

[18] 4G hat, "Raspberry Pi 3G/4G LTE Base Hat" [Online]. Available: https://sixfab.com/product/raspberry-pi-base-hat-3g-4g-lte-minipcie-cards/ [Used: June 2020]

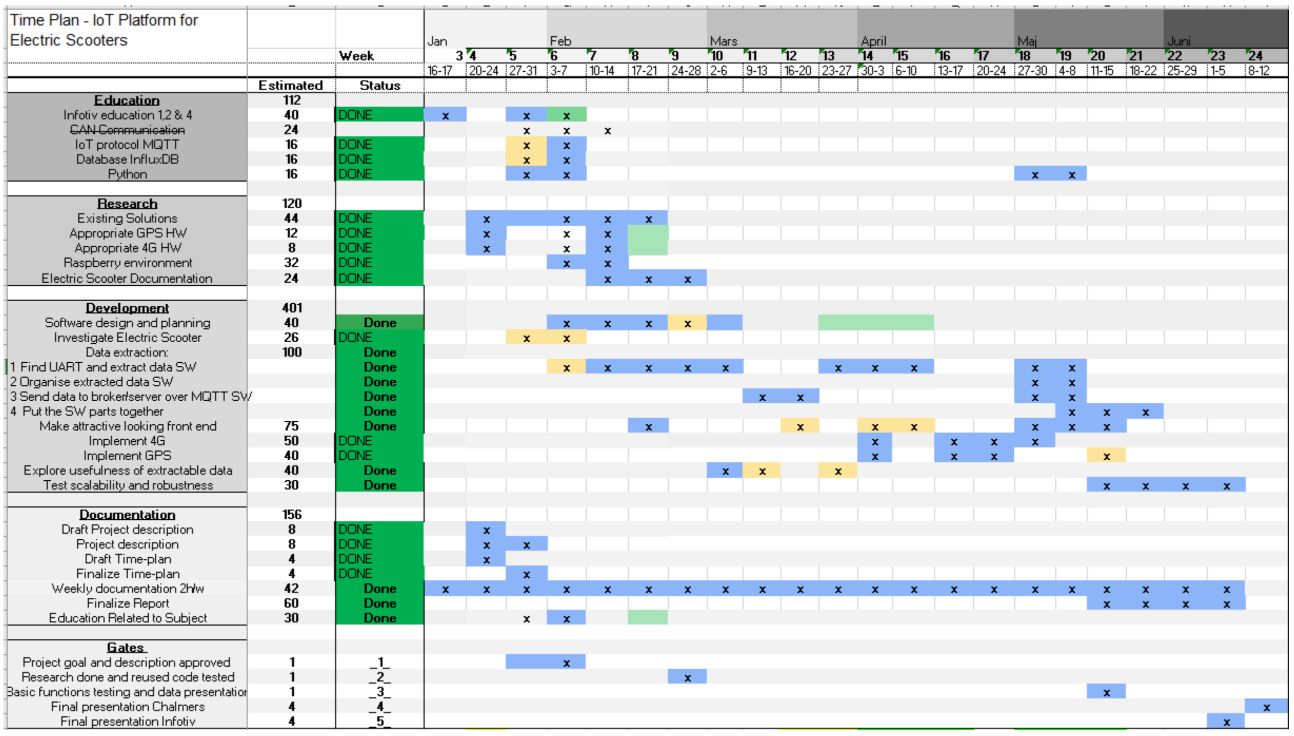
[19] 4G Module "Quectel EC25 Mini PCle 4G/LTE Module" [Online]. Available: https://sixfab.com/product/quectel-ec25-mini-pcle-4glte-module/ [Used: June 2020]

[20] 4G Antenna "LTE Main Diversity GNSS Triple Port u.FL Antenna – 100mm" [Online]. Available: https://sixfab.com/product/lte-main-diversity-gnss-triple-port-u-fl-antenna-100mm/ [Used: June 2020]

[21] TLS "Transport Layer Security" [Online]. Available: https://en.wikipedia.org/wiki/Transport_Layer_Security [Used: June 2020]

A

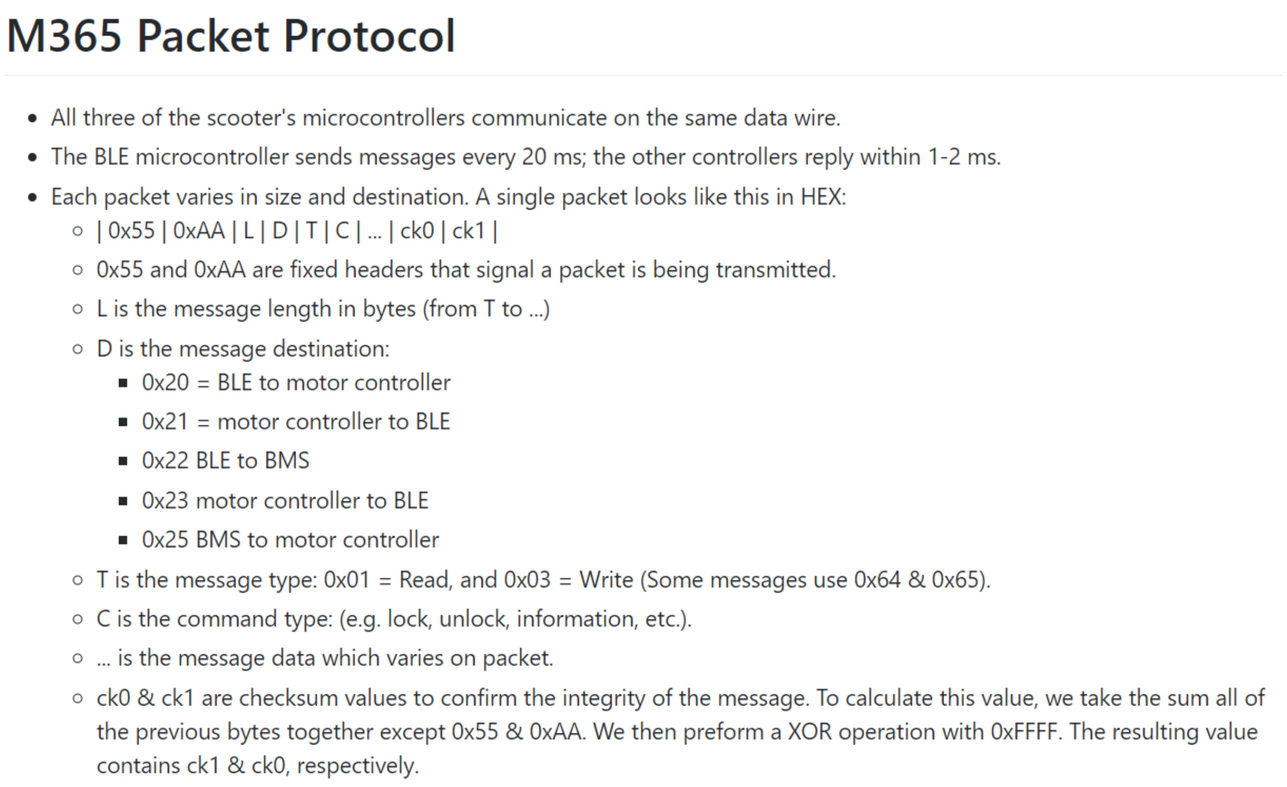
Appendix 1 - Time plan



Time Plan - IoT Platform for Electric Scooters			Jan			Feb				Mars				April				Maj				Juni		
		Week	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
			16-17	20-24	27-31	3-7	10-14	17-21	24-28	2-6	9-13	16-20	23-27	30-3	6-10	13-17	20-24	27-30	4-8	11-15	18-22	25-29	1-5	8-12
	Estimated	Status																						
Education	112																							
Infotiv education 1,2 & 4	40	DONE	x		x	x																		
~~CAN Communication~~	24				x	x	x																	
IoT protocol MQTT	16	DONE			x	x																		
Database InfluxDB	16	DONE			x	x																		
Python	16	DONE			x	x												x	x					
Research	120																							
Existing Solutions	44	DONE		x		x	x	x																
Appropriate GPS HW	12	DONE		x		x	x																	
Appropriate 4G HW	8	DONE		x		x	x																	
Raspberry environment	32	DONE				x	x																	
Electric Scooter Documentation	24	DONE					x	x	x															
Development	401																							
Software design and planning	40	Done				x	x	x	x															
Investigate Electric Scooter	26	DONE			x	x																		
Data extraction:	100	Done																						
1 Find UART and extract data SW		Done				x	x	x	x	x			x	x	x			x	x					
2 Organise extracted data SW		Done																x	x					
3 Send data to broker/server over MQTT SW		Done									x	x						x	x					
4 Put the SW parts together		Done																	x	x	x			
Make attractive looking front end	75	Done						x				x		x	x			x	x	x				
Implement 4G	50	DONE												x		x	x	x						
Implement GPS	40	DONE												x		x	x			x				
Explore usefulness of extractable data	40	Done								x	x		x											
Test scalability and robustness	30	Done																		x	x	x	x	
Documentation	156																							
Draft Project description	8	DONE		x																				
Project description	8	DONE		x	x																			
Draft Time-plan	4	DONE		x																				
Finalize Time-plan	4	DONE			x																			
Weekly documentation 2h/w	42	Done	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Finalize Report	60	Done																		x	x	x	x	
Education Related to Subject	30	Done			x	x																		
Gates																								
Project goal and description approved	1	_1_				x																		
Research done and reused code tested	1	_2_							x															
Basic functions testing and data presentatio	1	_3_																		x				
Final presentation Chalmers	4	_4_																						x
Final presentation Infotiv	4	_5_																					x	

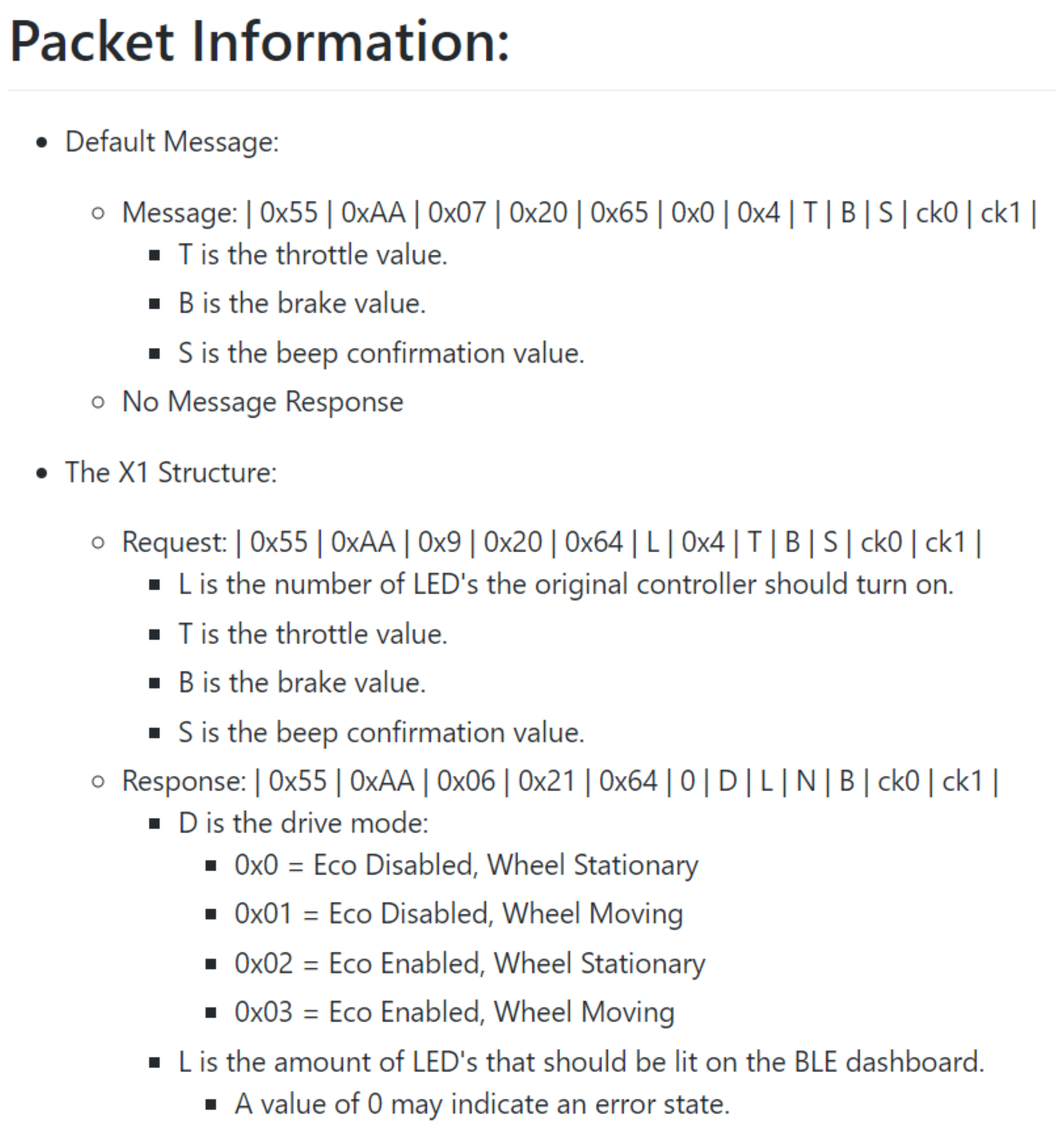
B

Appendix 2 - Decoding protocol



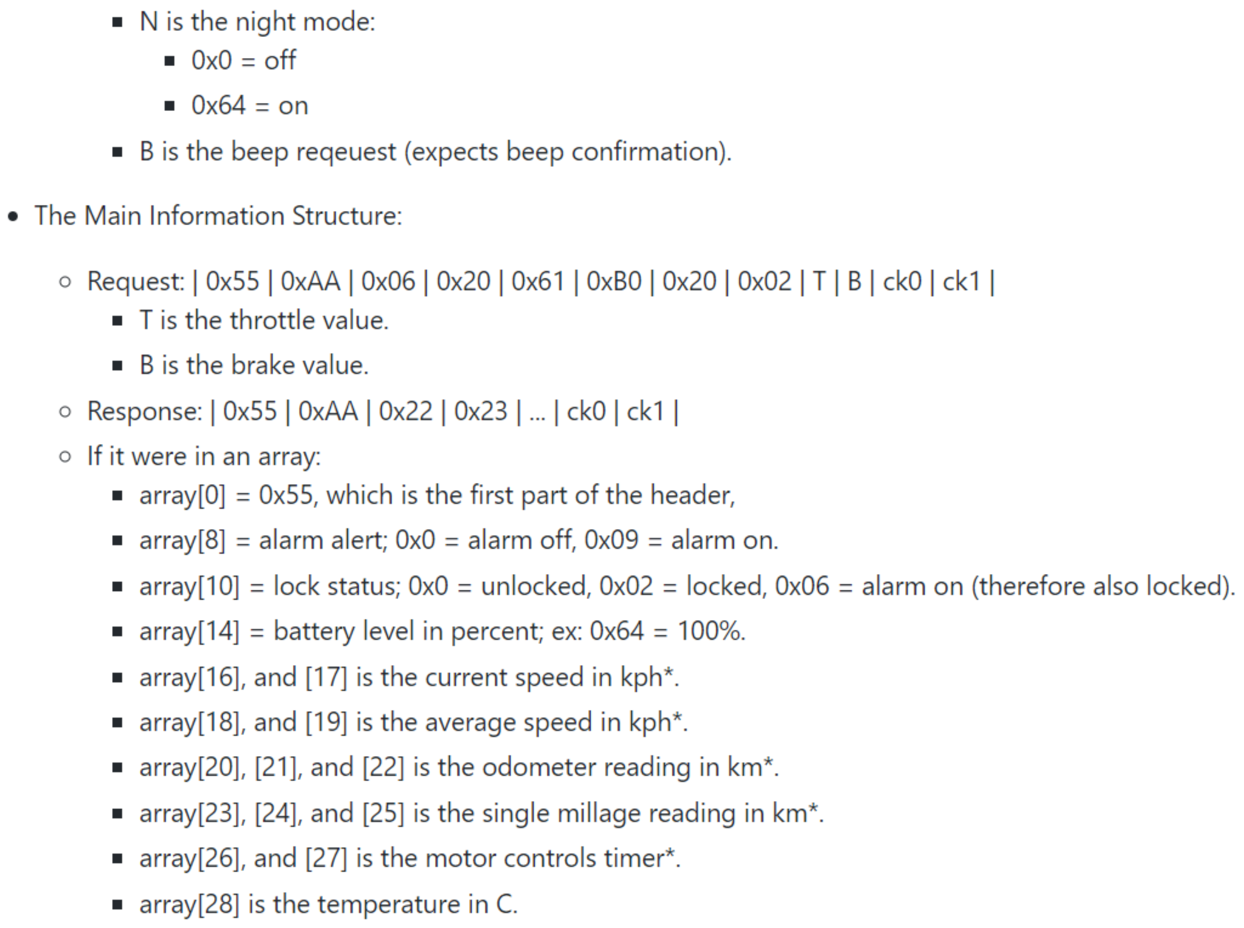
M365 Packet Protocol

- All three of the scooter's microcontrollers communicate on the same data wire.
- The BLE microcontroller sends messages every 20 ms; the other controllers reply within 1-2 ms.
- Each packet varies in size and destination. A single packet looks like this in HEX:
  - | 0x55 | 0xAA | L | D | T | C | ... | ck0 | ck1 |
  - 0x55 and 0xAA are fixed headers that signal a packet is being transmitted.
  - L is the message length in bytes (from T to ...)
  - D is the message destination:
    - 0x20 = BLE to motor controller
    - 0x21 = motor controller to BLE
    - 0x22 BLE to BMS
    - 0x23 motor controller to BLE
    - 0x25 BMS to motor controller
  - T is the message type: 0x01 = Read, and 0x03 = Write (Some messages use 0x64 & 0x65).
  - C is the command type: (e.g. lock, unlock, information, etc.).
  - ... is the message data which varies on packet.
  - ck0 & ck1 are checksum values to confirm the integrity of the message. To calculate this value, we take the sum all of the previous bytes together except 0x55 & 0xAA. We then preform a XOR operation with 0xFFFF. The resulting value contains ck1 & ck0, respectively.



Packet Information:

- Default Message:
  - Message: | 0x55 | 0xAA | 0x07 | 0x20 | 0x65 | 0x0 | 0x4 | T | B | S | ck0 | ck1 |
    - T is the throttle value.
    - B is the brake value.
    - S is the beep confirmation value.
  - No Message Response
- The X1 Structure:
  - Request: | 0x55 | 0xAA | 0x9 | 0x20 | 0x64 | L | 0x4 | T | B | S | ck0 | ck1 |
    - L is the number of LED's the original controller should turn on.
    - T is the throttle value.
    - B is the brake value.
    - S is the beep confirmation value.
  - Response: | 0x55 | 0xAA | 0x06 | 0x21 | 0x64 | 0 | D | L | N | B | ck0 | ck1 |
    - D is the drive mode:
      - 0x0 = Eco Disabled, Wheel Stationary
      - 0x01 = Eco Disabled, Wheel Moving
      - 0x02 = Eco Enabled, Wheel Stationary
      - 0x03 = Eco Enabled, Wheel Moving
    - L is the amount of LED's that should be lit on the BLE dashboard.
      - A value of 0 may indicate an error state.



    - N is the night mode:
        - 0x0 = off
        - 0x64 = on
    - B is the beep reqeuest (expects beep confirmation).

- The Main Information Structure:
  - Request: | 0x55 | 0xAA | 0x06 | 0x20 | 0x61 | 0xB0 | 0x20 | 0x02 | T | B | ck0 | ck1 |
    - T is the throttle value.
    - B is the brake value.
  - Response: | 0x55 | 0xAA | 0x22 | 0x23 | ... | ck0 | ck1 |
  - If it were in an array:
    - array[0] = 0x55, which is the first part of the header,
    - array[8] = alarm alert; 0x0 = alarm off, 0x09 = alarm on.
    - array[10] = lock status; 0x0 = unlocked, 0x02 = locked, 0x06 = alarm on (therefore also locked).
    - array[14] = battery level in percent; ex: 0x64 = 100%.
    - array[16], and [17] is the current speed in kph*.
    - array[18], and [19] is the average speed in kph*.
    - array[20], [21], and [22] is the odometer reading in km*.
    - array[23], [24], and [25] is the single millage reading in km*.
    - array[26], and [27] is the motor controls timer*.
    - array[28] is the temperature in C.

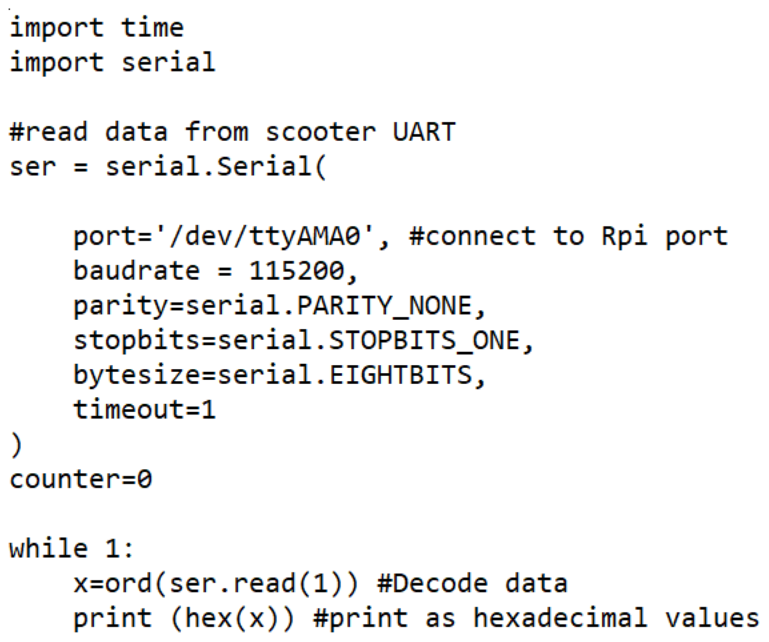
C
Appendix 3 - Arduino code

```
static byte databuf[30];

void setup() {
  Serial.begin(115200);
  Serial1.begin(115200);
    int antalbytes = Serial1.available();
    Serial.print(antalbytes);
    Serial.print("     ");


}
```

```
void loop() {
  //static long currentMillis = millis();
  static byte tempthrottle = 0;
  static byte tempbrake = 0;
  static int counter = 0;
  static int throttlesum = 0;
  static int throttlemed = 0;
  static int mappedThrottleMed = 0;
  static int brakesum = 0;
  static int brakemed = 0;
  static int mappedBrakeMed = 0;
    //Serial.println("Serial available");
    while(!Serial1.available()){}
    //int bytesnum = Serial1.available();
      if (Serial1.read() == 0x55 && Serial1.peek() == 0xAA){
        byte kasta = Serial1.read();
        int laengd = Serial1.peek();
        for(int i = 0; i <= laengd+3; i++){
          byte tempdata = Serial1.read();
          //if (tempdata == 0x55) break;
          if(laengd == 7){
            if(i == 5) tempthrottle = tempdata;
            if(i == 6) tempbrake = tempdata;
          }
  throttlesum += tempthrottle;
  brakesum += tempbrake;
  counter++;
  if(counter == 20){
    throttlemed = throttlesum/20;
    brakemed = brakesum/20;
    counter = 0;
    throttlesum = 0;
    brakesum = 0;
    mappedThrottleMed = map(throttlemed, 40, 197, 0, 100);
    mappedBrakeMed = map(brakemed, 39, 175, 0, 100);
    Serial.print("0 ");  // To freeze the lower limit
    Serial.print(mappedBrakeMed);
    Serial.print(",");
    Serial.print(mappedThrottleMed);
    Serial.println(" 100");  // To freeze the upper limit

  }

  delay(2);
}
```

D

Appendix 4 - Python script reading default data from the scooter



```
import time
import serial

#read data from scooter UART
ser = serial.Serial(

    port='/dev/ttyAMA0', #connect to Rpi port
    baudrate = 115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1
)
counter=0

while 1:
    x=ord(ser.read(1)) #Decode data
    print (hex(x)) #print as hexadecimal values
```

E

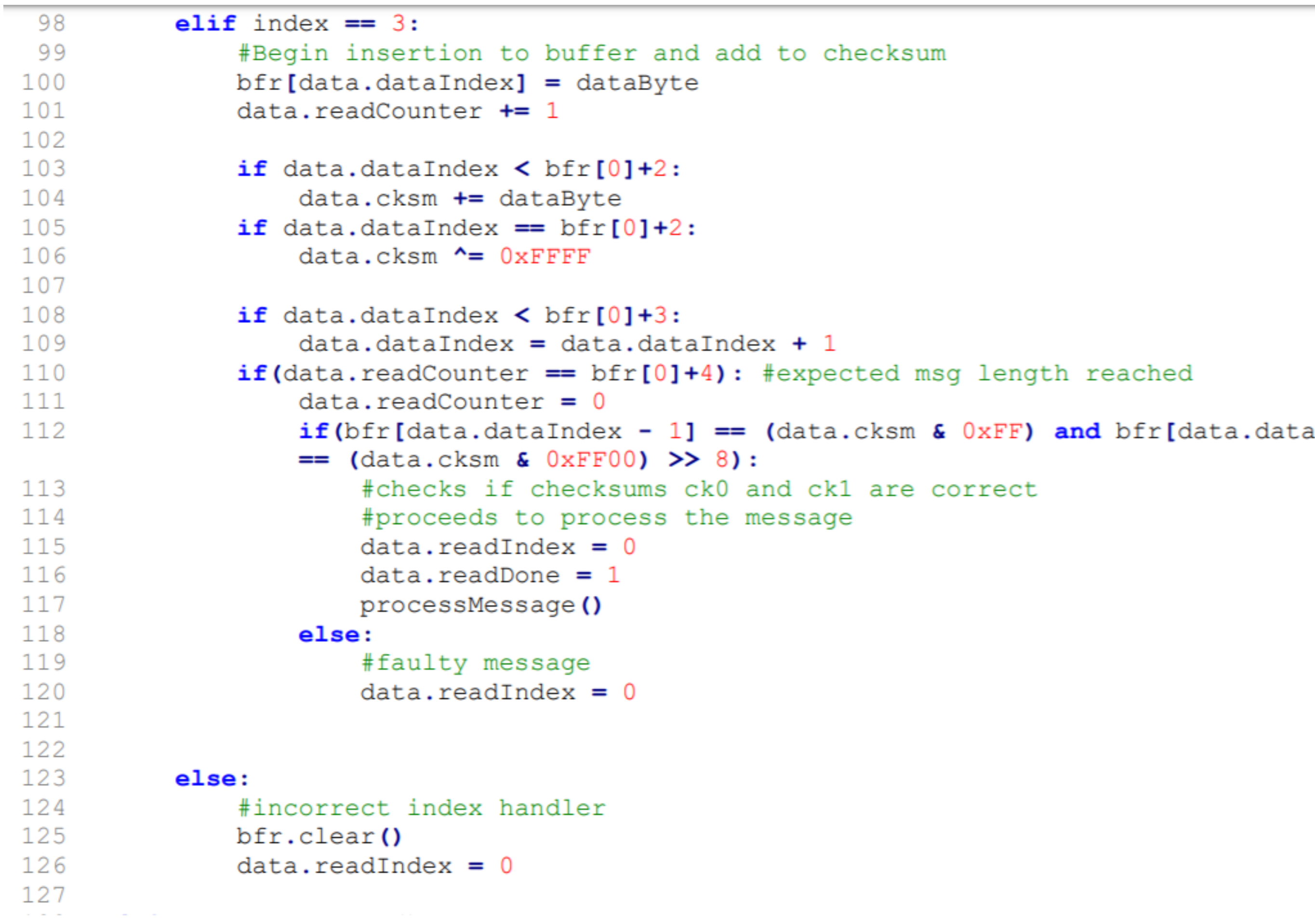
Appendix 5 - Python script that extracts, decodes and sends scooterdata

```python
#!/usr/bin/env python3


#This is the script that decodes messages on the scooter bus and sends them through
MQTT
#The credits for scooter protocol interpretation belongs to Camilo Ruiz
github.com/CamiAlfa/M365-BLE-PROTOCOL

#Expected message recipe for all messages:
#("First" refers to left-most byte)
#Header = First two bytes
#Message length = Third byte
#Message destiny = Fourth byte
# 0x20 = Bluetooth board to motor controller
# 0x21 = Motor controller to bluetooth board
# 0x22 = Bluetooth board to BMS
# 0x23 = Motor controller to Bluetooth
# 0x25 = BMS to motor controller
#Checksum = Last two bytes

#Important default message specific = Throttle at byte 8, Brake at byte 9

# Example default message:         T    B            ck0  ck1
#[0x55,0xAA,0x07,0x20,0x65,0x0,0x4,0x28,0x27,0x0,0x0,0x20,0xff]




import time
import serial
import paho.mqtt.publish as publish

#MQTT settings
host ="MQTT.infotiv.se"
```

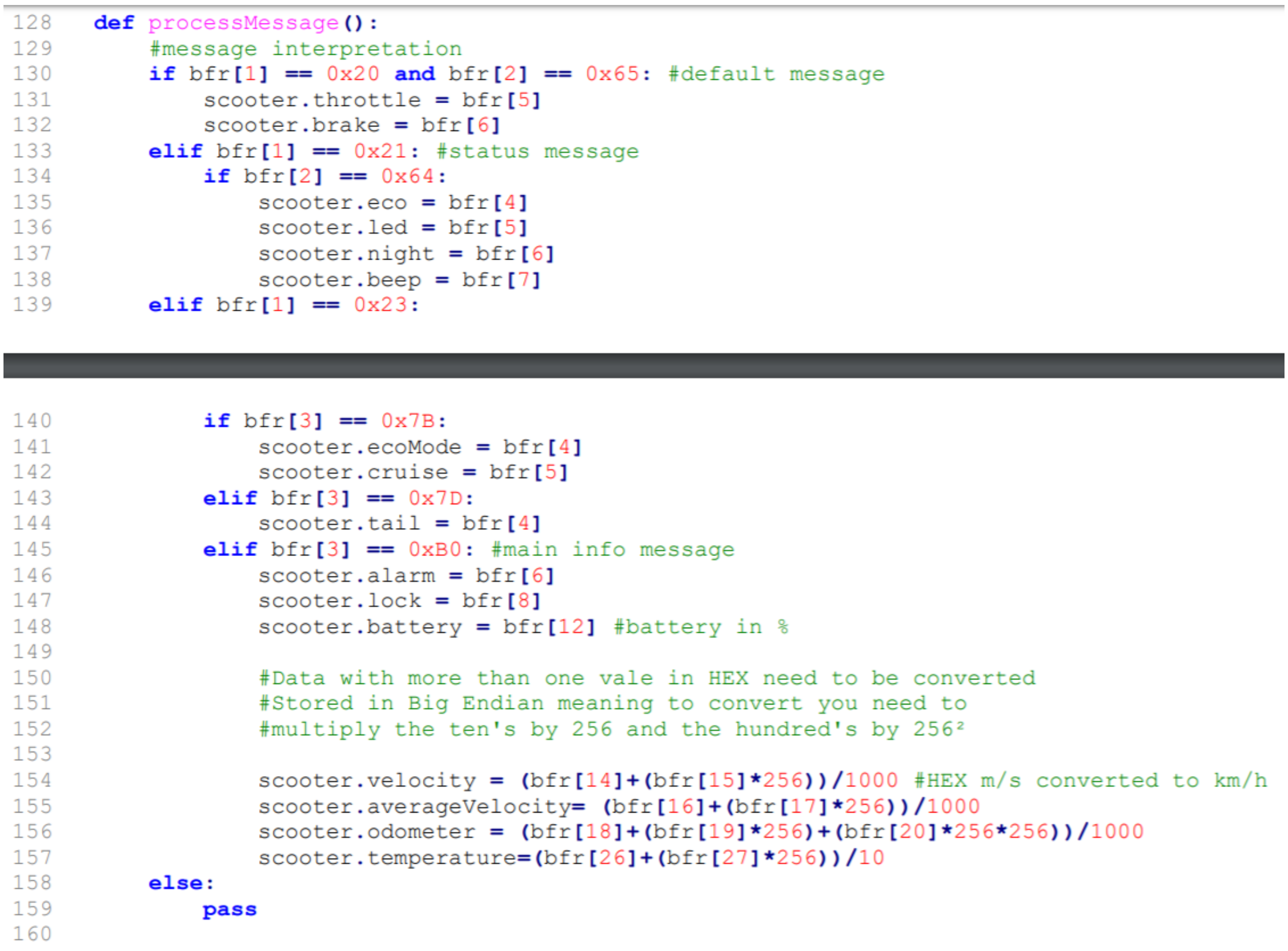
```python

class M365: #class to store scooter extracted data
    def __init__(self,throttle,brake,eco,led,night,beep,ecoMode,cruise,tail,alarm,lock,battery,velocity,lat,lng,averageVelocity,odometer,temperature):
        self.throttle = throttle
        self.brake = brake
        self.eco = eco
        self.led = led
        self.night = night
        self.beep = beep
        self.ecoMode = ecoMode
        self.cruise = cruise
        self.tail = tail
        self.alarm = alarm
        self.lock = lock
        self.battery = battery
        self.velocity = velocity
        self.lat = lat
        self.lng = lng
        self.averageVelocity = averageVelocity
        self.odometer = odometer
        self.temperature = temperature

#scooter object
scooter = M365(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

#counters and flag variables
class data:
    readIndex = 0
    dataIndex = 0
    readDone = 0
    writeDone = 0
    cksm = 0
    readCounter = 0
```

```python

#initialize serial connection to match the scooter
ser = serial.Serial(
```

```python
    port='/dev/ttyAMA0',
    baudrate = 115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1
)

#message buffer
bfr = [0] * 64
msg = 0
pubMQTT = 0
def readMessage(index,dataByte): #Reads one byte at a time
    if index == 0: #first byte of header
        if dataByte == 0x55:
            data.readIndex = 1
    elif index == 1:#second byte of header
        if dataByte == 0xAA:
            data.readIndex = 2
        else:
            #not a message, continue searching for header
            data.readIndex = 0
    elif index == 2:
        #Message header confirmed
        bfr.insert(0, dataByte)
        data.cksm = dataByte
        data.readCounter = 1
        data.dataIndex = 1
        data.readIndex = 3
```



```
        elif index == 3:
            #Begin insertion to buffer and add to checksum
            bfr[data.dataIndex] = dataByte
            data.readCounter += 1

            if data.dataIndex < bfr[0]+2:
                data.cksm += dataByte
            if data.dataIndex == bfr[0]+2:
                data.cksm ^= 0xFFFF

            if data.dataIndex < bfr[0]+3:
                data.dataIndex = data.dataIndex + 1
            if(data.readCounter == bfr[0]+4): #expected msg length reached
                data.readCounter = 0
                if(bfr[data.dataIndex - 1] == (data.cksm & 0xFF) and bfr[data.data
                == (data.cksm & 0xFF00) >> 8):
                    #checks if checksums ck0 and ck1 are correct
                    #proceeds to process the message
                    data.readIndex = 0
                    data.readDone = 1
                    processMessage()
                else:
                    #faulty message
                    data.readIndex = 0


        else:
            #incorrect index handler
            bfr.clear()
            data.readIndex = 0

```



```python
def processMessage():
    #message interpretation
    if bfr[1] == 0x20 and bfr[2] == 0x65: #default message
        scooter.throttle = bfr[5]
        scooter.brake = bfr[6]
    elif bfr[1] == 0x21: #status message
        if bfr[2] == 0x64:
            scooter.eco = bfr[4]
            scooter.led = bfr[5]
            scooter.night = bfr[6]
            scooter.beep = bfr[7]
    elif bfr[1] == 0x23:
```

```python
        if bfr[3] == 0x7B:
            scooter.ecoMode = bfr[4]
            scooter.cruise = bfr[5]
        elif bfr[3] == 0x7D:
            scooter.tail = bfr[4]
        elif bfr[3] == 0xB0: #main info message
            scooter.alarm = bfr[6]
            scooter.lock = bfr[8]
            scooter.battery = bfr[12] #battery in %

            #Data with more than one vale in HEX need to be converted
            #Stored in Big Endian meaning to convert you need to
            #multiply the ten's by 256 and the hundred's by 256²

            scooter.velocity = (bfr[14]+(bfr[15]*256))/1000 #HEX m/s converted to km/h
            scooter.averageVelocity= (bfr[16]+(bfr[17]*256))/1000
            scooter.odometer = (bfr[18]+(bfr[19]*256)+(bfr[20]*256*256))/1000
            scooter.temperature=(bfr[26]+(bfr[27]*256))/10
    else:
        pass

```

```python
def getInfo():
    #request main info message
    #sends specific message on the bus, including current throttle and brake values
    infocmd =
    [0x55,0xAA,0x06,0x20,0x61,0xB0,0x20,0x02,scooter.throttle,scooter.brake,0x0,0x0]

    #calculate checksum and insert into message
    infocksm = 0
    for j in range(2,10):
        infocksm += infocmd[j]
    infocksm ^= 0xFFFF #
    infocmd[11] = (infocksm & 0xFF00) >> 8
    infocmd[10] = (infocksm & 0xFF)

    #construct message
    scooterMSG = bytearray()
    scooterMSG.append(infocmd[0])
    scooterMSG.append(infocmd[1])
    scooterMSG.append(infocmd[2])
    scooterMSG.append(infocmd[3])
    scooterMSG.append(infocmd[4])
    scooterMSG.append(infocmd[5])
    scooterMSG.append(infocmd[6])
    scooterMSG.append(infocmd[7])
    scooterMSG.append(infocmd[8])
    scooterMSG.append(infocmd[9])
    scooterMSG.append(infocmd[10])
    scooterMSG.append(infocmd[11])


    #transmit message to scooter bus
    ser.reset_input_buffer()
    ser.write(scooterMSG)
    ser.flush

```

```python
#ser.flush() waits until all data in ser.write(data) have been written
#ser.in_waiting returns number of bytes in incoming buffer (int)
#ser.name names which port is being used
#ser.reset_input_buffer() discards input buffer


if __name__ == "__main__":
    #main loop
    while 1:
        if ser.in_waiting > 0: #checks if serial has incoming data
            nextByte = ord(ser.read(1)) #read first byte in serial buffer
            readMessage(data.readIndex, nextByte)
            if(data.readDone):
                #increment message counters on successful message read
                msg += 1
                pubMQTT += 1
```

```python

                if(msg > 10):
                    #request main info message every 10 message
                    getInfo()
                    msg = 0
                if(pubMQTT > 50):
                    #publish to MQTT every 50 message
                    #publish.single(topic="scooter/data",payload="Battery: %d%% Speed: %dkm/h" % (scooter.battery,scooter.velocity),hostname=host)

                    messages = [{'topic':"/scooteriot/scooter1/speed",'payload':scooter.velocity},
                    ("/scooteriot/scooter1/battery",scooter.battery, 0, False)]

                    publish.multiple(messages, hostname=host)
                    pubMQTT=0
                data.readDone = 0
```

F

Appendix 6 - Python script that collects and sends GPS positions

```python
#!/usr/bin/env python3

#This code is developed with the Sixfab 4G/GPS hat paired with Quectel EC25-E communication chip
#NMEA sentence writes are done through ttyUSB2 and responses are received at ttyUSB1
#Those response messages contain information, in this case the position coordinates

from time import sleep
import serial
import paho.mqtt.publish as publish
import math

#MQTT and serial settings
host ="MQTT.infotiv.se"
portwrite = "/dev/ttyUSB2"
port = "/dev/ttyUSB1"

#scooter variables
class scooter:
    latitude = 0
    longitude = 0
    satellites = 0

def parseGPS(data):

    if data[0:6].decode() == "$GPRMC":
        #NMEA sentence including pos. data
        sdata = data.decode().split(",")
        if sdata[2] == 'V':#invalid data, do not proceed
            #print ("no satellite data available")
            return
        #print ("-----Parsing GPRMC-----")

        lat= decode(sdata[3]) #latitude
        scooter.latitude = lat
        lon = decode(sdata[5]) #longitude
        scooter.longitude = lon
        print("Publishing coordinates lat: %s and long: %s to MQTT" % (lat,lon))
        #Publish to MQTT
        messages = [{'topic':"/scooteriot/scooter1/lat",'payload':scooter.latitude},
                    ("/scooteriot/scooter1/lng",scooter.longitude, 0, False)]

        publish.multiple(messages, hostname=host)

        ## Additional data
        #time = sdata[1][0:2] + ":" + sdata[1][2:4] + ":" + sdata[1][4:6]
        #dirLat = sdata[4]      #latitude direction N/S
        #dirLon = sdata[6]      #longitude direction E/W
        #speed = sdata[7]       #Speed in knots
        #trCourse = sdata[8]    #True course
        #date = sdata[9][0:2] + "/" + sdata[9][2:4] + "/" + sdata[9][4:6]
        #variation = sdata[10]  #variation
        #degreeChecksum = sdata[11]
        #print ("time : %s, latitude : %s(%s), longitude : %s(%s), speed : %s, True Course : %s, Date : %s, Magnetic Variation : %s(%s)"%(time,lat,dirLat,lon,dirLon,speed,trCourse,date,variation,degreeChecksum))
    else:
        pass
        #Prints the NMEA message
        #print(data[0:6])

#truncation function needed for GPS coordinate conversion from Deg,Min,Sec to Decimal Degree
def truncate(n, decimals=0):
    multiplier = 10 ** decimals
    return (int(n * multiplier) / multiplier)


def decode(coord):
    x = coord.split(".")
    head = x[0]
    tail = x[1]
    deg = head[0:-2]
    min = head[-2:]
    tail = truncate(float(tail),-1)/1000000
    DD = float(deg)+(float(min)/60)+(tail/3600)
    #print(deg + " " + min + " " + str(tail))
    summa = truncate(DD, 8)
    #print(DD)
    return DD

#Connecting port and starting GPS
serw = serial.Serial(portwrite, baudrate = 115200, timeout = 1)
serw.write("AT+QGPS=1\r".encode())
serw.close()
sleep(0.5)

#Open serial port for GPS data
ser = serial.Serial(port, baudrate = 115200, timeout = 0.5)

while 1:
    #main loop
    data = ser.readline()
    parseGPS(data)


```