Institutt for Informasjonsteknologi
Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo
Besøksadresse: Holbergs plass, Oslo
Telefon: 22 45 32 00

PROSJEKT NR.
23-2022

TILGJENGELIGHET
Offentlig

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL	DATO
IoT - Oppgave	25.05.2022
	ANTALL SIDER / BILAG Over 9000
PROSJEKTDELTAKERE Andreas Torres Hansen Uy Quoc Nguyen Anders Hagen Ottersland	INTERN VEILEDER Jianhua Zhang

OPPDRAGSGIVER	KONTAKTPERSON
Accenture	Daniel Meinecke

SAMMENDRAG

Dette er en kjempe kul prosjekt ☺

3 STIKKORD
Internet of Things
Raspberry Pi
Sentralisert System

DATA3900 - Bachelor project
Final report

Gruppe 23:
Hansen, Andreas Torres (s338851)
Nguyen, Uy Quoc (s341864)
Ottersland, Anders Hagen (s341883)

Total pages: 49

Last updated:
May 22, 2022

Abstract

A summary of the project, including the result that the group reached.

Preface

This is the report of our bachelor thesis at Oslo Metropolitan University, Faculty of Technology, Art and Design. We tried to develop a solution for traffic management with self-driving cars and server communication. Our project was created for Accenture and lasted from January 2022 to May 2022. In this project, we documented the functionality and process of making it. We physically demonstrated our proposed solution with a Raspberry Pi computer working as a car.

The report is split into seven chapters—introduction, research areas, process documentation, implementation, results, discussion, and conclusion. We have added a Moscow analysis, sprint overview, and our project journal as appendices. Technical terms are in the appendix for those with less technical knowledge. (Kan kanskje droppe dette)

We would like to thank everyone that has contributed to our project. We would especially like to thank:

- Ivar Fauske Aasen, Solfrid Johansen and Benjamin Vallestad, representatives from Accenture.
- Dr. Jianhua Zhang, internal supervisor at OsloMet.

Contents

PART I

PRELIMINARY

This text is supposed to convey what this part of the report is about.

Chapter 1

Introduction

1.1 Stakeholders

1.1.1 Students

Andreas Torres Hansen, Software Engineering

Anders Hagen Ottersland, Software Engineering

Uy Quoc Nguyen, Software Engineering

1.1.2 Product owner

Benjamin Vallestad

1.1.3 Supervisors

Professor Jianhua Zhang, Internal Supervisor

Ivar Austin Fauske, External Supervisor

Solfrid Hagen Johansen, External Supervisor

1.1.4 Client

Accenture AS

Accenture is an international IT firm that operates in 200 different countries worldwide. They offer a wide range of services in many fields, like artificial intelligence, data analytics, and cloud computing. The main office is in Dublin, and the Norwegian main office is in Fornebu. Accenture has 674 thousand employees internationally, of which 1000 work in Norway (**accenture_earning_report_2021**). In 2021, Accenture generated a revenue of approximately $50.3 billion (**accenture_about**).

1.2 Project Description

1.2.1 Project background

Accenture has, over the years, offered final year students at OsloMet and Høyskolen Kristiania innovative projects for their bachelor thesis. In 2020, a group of students from Høyskolen Kristiania was developing model-sized self-driving vehicles using Raspberry Pi in conjunction with machine learning as their project. This year our group was offered to extend this project further; to explore plausible improvement with the addition of a centralized communication system.

Norway is one of the countries that are ready to utilize self-driving cars, according to Accenture. However, self-driving vehicles alone are likely not enough to solve all of today's traffic challenges. Hence, this project aims to solve the issue by introducing a management system for autonomous vehicles and evaluating the value such a system can provide.

1.2.2 Significance

Our project provides value for Accenture in the shape of building knowledge around new technology and theory. Accenture also wants to explore the potential positive societal and climate effects a centralized communication system for transportation could provide.

1.2.3 Goals and requirements

The project aims to produce a prototype that can be shown to stakeholders and, either physically or digitally, demonstrate how Accenture could combine self-driving cars with a centralized system. We will utilize the self-driving vehicle built for Accenture from the previous bachelor thesis. Preferably, the prototype can show at least one situation where the outcome differs depending on the use of self-driving cars plus a centralized system versus only using self-driving vehicles. The system should also be scalable so that more vehicles can be added or removed at a later point in time.

1.2.4 Problem statement

Based on the goals and requirements set by Accenture, we have formulated the research and development question:

> "How can we improve traffic flow, by using a combination of self-driving cars and a centralized communication system?"

PART II

PROCESS DOCUMENTATION

This text is supposed to convey what this part of the report is about.

Chapter 2

Work methodology and technology used

2.1 Development method

We had very few requirements and technical restrictions when we received the project, which left the project open to interpretation. Therefore, we wanted to choose a flexible work methodology. Agile work methods focus on continuous planning throughout the process and having frequent communication with the client, in our case Accenture. We had meetings with our external supervisors from Accenture once every second week, which meant the agile model was a good fit for our project. We took inspiration from two light frameworks, Scrum and Kanban.

We took inspiration from two light frameworks, Kanban and Scrum. Scrum is an agile, light framework that helps people and teams work together. Scrum describes a set of meetings, roles, and tools.

A sprint is an essential part of using the Scrum framework. Sprints are a fixed time length, often between one and four weeks. In this specified time length, the teams do tasks assigned from the sprint backlog. Each sprint starts with sprint planning and ends with a sprint retrospective. We found it most viable for our project to plan in increments of two weeks. We chose two-week increments because we felt it was an even balance between work and planning. As mentioned, we also had meetings with our client Accenture every two weeks, which fitted well with the time increment.

2.1.1 Scrum and sprints

Scrum is a framework that dictates how developers work in teams to solve complex problems. The development process is also divided into time intervals called a sprint. A sprint is an essential part of using the Scrum framework (**prosjektveilederen**). Sprints are a fixed time length, often between one and four weeks. In this specified time length, the teams do tasks assigned from the

sprint backlog. Each sprint starts with sprint planning and ends with a sprint retrospective.

We found it most viable for our project to plan in increments of two weeks. We chose two-week increments because we felt it was an even balance between work and planning. This time increment also fitted our bi-weekly meeting plan with our supervisors.

Our group also used the meetings in the Scrum framework, which consist of sprint planning, sprint retrospective, and daily standups. Sprint planning is a meeting or event which starts before a sprint. During sprint planning, teams agree on goals for the sprint and what tasks from the backlog should be prioritized. The backlog is a list of functionality the product should contain. In addition, we wrote down the tasks for the specific sprints in Google Docs. These tasks were to be finished by the next sprint. Figure 2.1 shows our sprint planning document that lists our goals for each sprint.

Sprints for bachelorproject

Sprint id	Oppgaver
1	Implementere web API Lagring av informasjon til database Organisering av prosjektet på git-hub
2	Integrere web-api løsningen vår med bilene fra forrige prosjekt Implementere enhetstesting Gjøre mer research for løsninger på oppgaven(webhooks, kafka, signalR)
3	Gjøre mer research på webhooks Få bilene til å sende riktig innformasjon til server Implementere signalR server
4	Starte å skrive på bacheloroppgaven Få server til å sende kommandoer til flere biler Få bilene til å reagere på kommandoene til serveren
5	Bygge bil nr 2 Lage et veisystem for demo med veikryss Skrive et førsteutkast for rapporten
6	Lage en demo som viser IoT-systemet Bli ferdig med å bygge bil 2 Skrive mer om implementasjon og starte å skrive på løsningen på rapporten
7	Gjøre ferdig alt det tekniske ved oppgaven Ferdigstille demoen Skrive ferdig implementasjon og løsnings-delen på oppgaven

Figure 2.1: An overview of our sprints. Each sprint lasted 2 weeks. This figure shows the main tasks of each sprint.

After a sprint, we would have a sprint retrospective to discuss what went well and what we could have done better. We would also examine if the task assigned in the sprint meetings was finished or needed more work. These meetings helped us reflect over the prior week and adjust accordingly, if necessary. The sessions also helped us determine if we were on track with our initial plan.

In addition to the weekly meetings, we also had daily standups. Daily standups are short meetings, usually lasting around five minutes, where each person answers three questions:

- What did the person do last time?
- What is the person going to do today?
- Are there any challenges?

We implemented daily standups because it helped our team get on the same page, and it made it easier to plan what each of us had to do that specific day.

Scrum often consists of a team with different roles. As a team of three, we did not feel the necessity to have specified roles because we usually worked together on our projects. However, we alternated on being the scrum master. The scrum master's responsibility is to keep track of the backlog and lead the sprint planning meetings.

2.1.2 Kanban as our backlog

Our implementation of Kanban was to use a Kanban board as the backlog. We used a Kanban board to visualize where a task is in the work process. Figure 2.2 shows an example from our project, with description labels and priority labels:

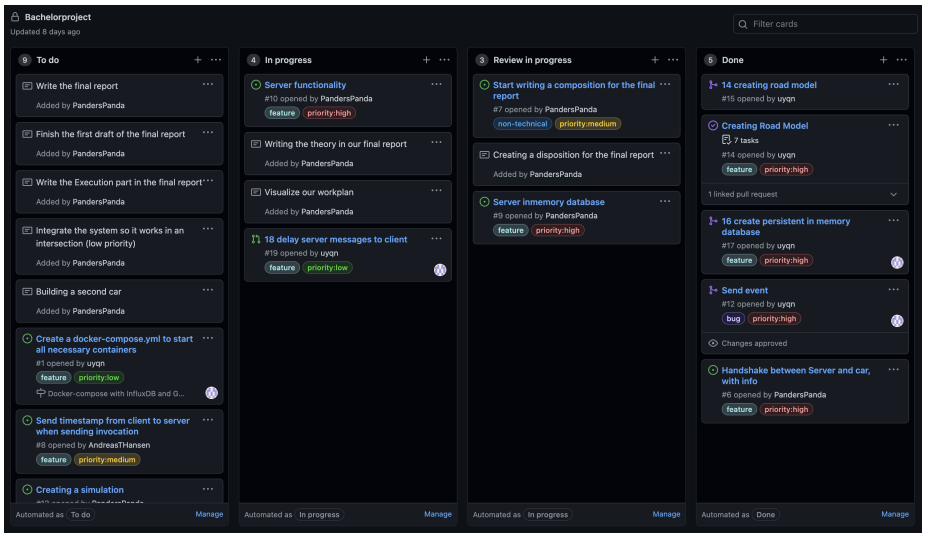


Figure 2.2: Extract of our Kanban board from Github. Each task has a color-coded label representing the priority, and a label to describe the task.

We have four columns that represent which phase a task is in. The backlog is the tasks in the to-do column. We dragged it over to the "In progress"-column when we were working on a task. After finishing a task, it went to the "Review in progress"-column, where we reviewed it. If we concluded that the task was finished in the reviewing, it was dragged over to the "Done"-column. Whenever a task is created we tagged it with a high, medium or low to indicate its priority. In addition, to the priority tag, we also gave each task a label; feature, bug, and non-technical to communicate what the task entails. You can also connect the

tasks to a specific branch, so the task automatically gets finished when merging the branch into the main branch. The Kanban board was a great tool to see which tasks to choose for our sprints and also keep track of where the tasks were in their process.

However, we did not use the Kanban board throughout the whole process. This is because the backlog was changing a lot, and the Kanban board needed many modifications to be up to date. We figured out that it was more beneficial to focus on one framework, which in our case was Scrum. In addition, we were able to keep track of the tasks by having frequent meetings.

2.2 Tools and technologies

The circumstances surrounding Covid 19 meant that we could not meet our supervisors in person at the start. Luckily, the group could still meet physically a few times a week. We had to use a wide range of tools for communication. Most restrictions were removed later in the project, but we kept our meetings with our supervisors digital throughout the project.

- Email - Formal communication with supervisors and product owner
- Teams - Meetings, and the platform of choice for communicating with the external supervisors on an informal level.
- Zoom - Meetings with our internal supervisor at OsloMet
- Messenger - Communication internally in the group. We used it to send messages to each other when we were not physically together, and to send pictures of code.

The project required us to collaborate while working on different personal computers, which can lead to overlapping. Therefore, tools that helped us work on the project together were essential. Here are the tools that helped us:

- Git and Github - Version control of choice
- Google docs - Used to write our journal and other documents that need to get updated regularly, and to share documents.
- Latex - Used to write reports.

We also needed text editors that supported the programming languages that we used. The client was built in python, while we built the server in C#.

- Pycharm - IDE for coding in python
- Visual studio and Rider - IDE for coding in C#
- Thonny - Text editor for coding in python on Raspberry pi

Project planning and documentation were also an important part of the project. The tools we used for the project planning were:

- Github project - Kanban board and creating backlog tasks
- Excel - Used for visualizing our workplan by using tables and a Gantt diagram

2.3 Prioritization method

The MoSCoW method is a prioritization technique used in project management. The word MoSCoW is an acronym where:

- "Mo" stands for must-have and represents our project's most prioritized requirements. These are necessary for the success of our project.
- "S" stands for should have. Our project should include these requirements, but they are not mandatory.
- "Co" stands for could have. We want to include these requirements in our project but they are not prioritized.
- "W" stands for will not have this time. The requirements in the "W" section might be for a later group if someone wants to build on our project further.

Since we were unsure of how many features we could finish within the time frame of the project period, we thought a prioritization method was a good fit. Figure 2.3 shows our visualization of the Moscow method.

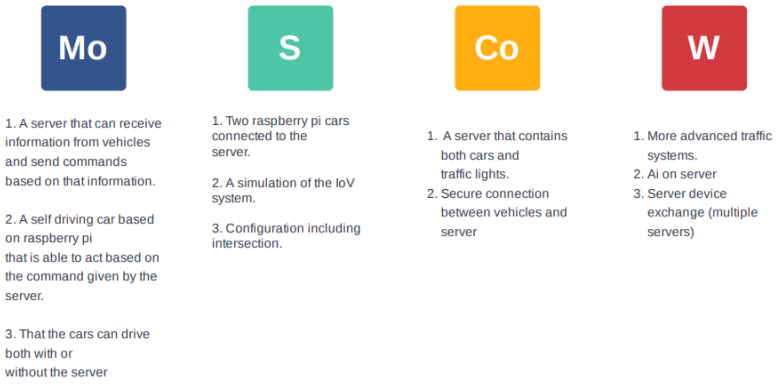


Figure 2.3: Visualuzation MoSCoW method. Everything under "Mo" are requirements we must have in our project, under the "S" sections are features we should have. Under "Co" are features we should have, but are not necessary. Under "W" are features we will not have this time

Chapter 3

Phases

Our sprints each lasted two weeks, as per described in section 2.1.1 Scrum and sprints, but in retrospect, it is apparent that we can divide our process into four phases:

Phase 1	-	Planning and research phase	week (1-4)
Phase 2	-	Rest API and why it did not work with our project	week (5-8)
Phase 3	-	Websocket with SignalR	week (9-12)
Phase 4	-	Demonstration of our system	week (13-18)

An overview of our sprints are in figure Figure 2.1

3.1 Phase 1 - Planning and research phase

After we had gotten in touch with Accenture and spoken with the supervisors and the product owner, the group had to make a few decisions regarding the project's direction.

An important choice for the project was to either build and train an AI model for the vehicles from scratch or use the existing model. Building a new AI model would provide a deeper understanding of the model we could utilize. Due to the time constraint, we determined that it was more favorable to continue with the existing model. In addition, we believed that the project would have the potential to be too similar to the previous project. Our group also had more prior experience with networking than with Raspberry Pi and AI. We, therefore, chose to use the AI model from the previous project.

In this phase, we did not have access to the vehicle, nor the code, made by the prior bachelor group. However, there was a need for project planning and research before we could start developing our IoT system anyway. The topics that needed to be researched were:

- What causes traffic jams and solutions to fix it.
- IoT-systems and how they function with vehicles.

- Planning and development methods that will fit our project

We also used the pre-project phase to get to know Accenture, their guidelines, and their workspace. Moreover, we participated in a "Design thinking"-workshop.

3.1.1 Design thinking workshop

3.1.2 Choice of programming languages

We chose to implement the client in python and the server in C#. The group before us had used python for their Raspberry Pi vehicle, making python a natural choice to extend the code from their project. Our group also had experience with networking in python. Furthermore, the .NET ecosystem has well-developed solutions for creating IoT applications, microservices, and web applications (Legg til kilde her). To take advantage of these solutions we had to write our server in C#. The server needed to be as efficient as possible, and C# is also considered a fast programming language (kilde?). We also had some prior knowledge of coding in C#.

3.1.3 Internet of Things

The Internet Of Things refers to physical objects that communicate using sensors, cameras, software, or other technologies connecting and exchanging data. This communication takes place over the internet or other communication forms. The number of connected IoT devices in the world is increasing, and it is becoming a big part of society (**iot_analytics**). IoT has also been evolving in recent years due to other technologies becoming more accessible, such as machine learning and the 5G network.

IoT projects can, for instance, be applied in climate surveillance systems, energy, or transportation. In this thesis, we will explore the possibilities of using IoT in transportation, more specifically in personal automobiles. The convergence of these fields is more commonly known as IoV, Internet of Vehicles. An IoV system is a distributed system for wireless communication and information exchange between vehicles through agreed-upon communication protocols (**chinese_iov**). The system could potentially integrate functionality for dynamic information exchange, vehicle control, and smart traffic management. In our thesis, we will explore these possibilities on a small scale.

3.1.4 Preventing traffic congestions for a one-lane road

«««< HEAD Traffic congestion, also known as traffic jams, is when a long line of vehicles moves slowly or has stopped moving altogether. Traffic jams can create frustration, cause more fuel consumption and cause accidents. Many factors can cause traffic congestion, such as: ======= Traffic congestion, also known as traffic jams, is when a long line of vehicles moves slowly or has stopped moving altogether. Traffic jams can create frustration and disrupt nearby local environments with sound and gas emissions (**traffic_congestion_pollution**). Many factors can cause traffic congestion, such as: »»»> 96b3a6e138da6cbb5d54db95e15739fae12e6ed2

poorly designed roads, not wide enough roads, traffic light patterns, and accidents (**traffic_congestion**).

With this in mind, we started by focusing on a simple scenario: when a car drastically reduces its speed or completely stops on a single-lane road.

This scenario will lead to the vehicles behind needing to slow down drastically as well. This phenomenon is called traffic jam shockwave (**traffic_shockwave**). To prevent this, we propose a solution where cars reduce their velocity before they reach the destination of where the shockwave started. For this to happen, a server could keep track of the cars' positions and send information to the vehicles behind, when required.

«««< HEAD After a few discussions, we came up with an idea of how the interactions between the server and the cars would be. First, the car had to connect to the server and give information about its current speed, weight, width, and length. The server would keep track of all the cars positions on the road. There was only one dimension to worry about since there was only a single road. The cars would send information to the server if their velocity changed. This message would trigger an event on the server where it would command all the cars behind the car that triggered the event to slow down accordingly. Figure 3.1 shows a flow chart of a potential demonstration with the cars from the previous group: ======= We came up with an idea on how the server and cars should interact. First, the car would connect to the server and provide information about its current speed, weight, width, and length. The server would use this information to keep track of all the cars' positions on the road. The cars would send information to the server if their velocity changed. This message would trigger an event on the server where it would command all the cars behind to slow down accordingly. Figure 3.1 shows a flow chart of a potential simulation of this solution: »»»> 96b3a6e138da6cbb5d54db95e15739fae12e6ed2

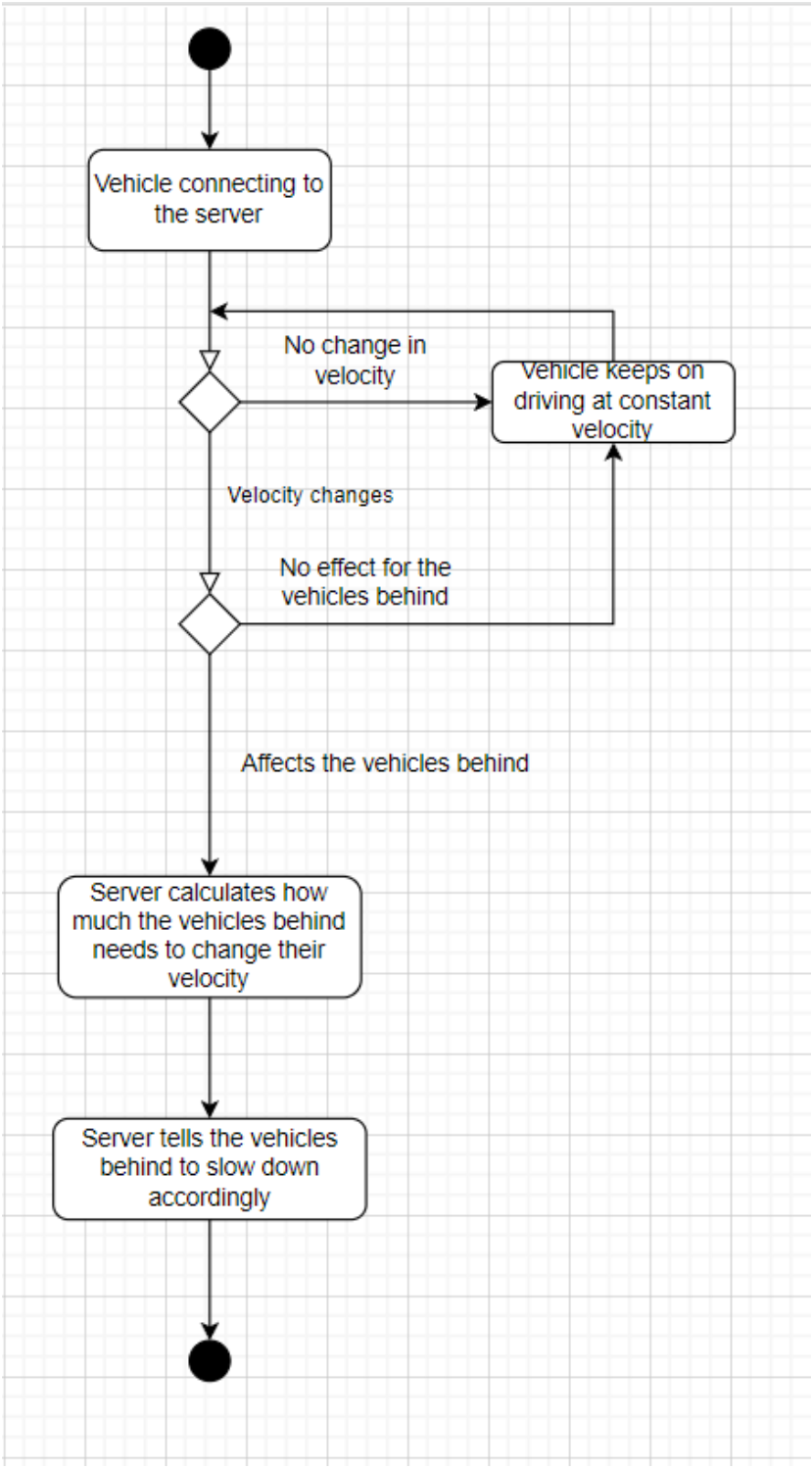


Figure 3.1: This figure shows the flow diagram of our first proposed solution.

3.2 Phase 2 - REST API and why it did not work with our project

The project aimed to connect Raspberry Pi devices to a server. Our initial approach was to implement a RESTful API on the server, as the connection layer between them. We also considered data security a critical aspect of an upscaled version of such a system. The group discussed how to store the data about the vehicle's positions during this phase, keeping in mind that we wanted our proof-of-concept to be scalable to real life.

Representational state transfer (REST) *application programming interface* (API) provides a way for clients and servers to establish communication through *hypertext transfer protocol* (HTTP), which is a protocol for transferring data between network devices. Using a REST API, clients can send requests to a server to perform standard CRUD (create, read, update and delete) operations on a database (**rest_api**). We wanted to keep track of all connected cars, on a database on the server. Therefore, REST API seemed like a good fit for our current solution.

Due to the time constraint of the program development, quickly choosing an appropriate type of database for our system was desirable, as migrating databases later could be a big timesink. In this case, we chose to incorporate a time-series database, which automatically includes a timestamp for each database entry. InfluxDB is a time-series database created by InfluxData and provides SQL-like syntax that is quick to query resources. Moreover, it provides both ease of installation and supports a multitude of languages (**influxdb**).

During this phase, the group implemented a standard REST API server with C# with the idea that the Raspberry Pi vehicles should exchange information on their velocity, acceleration, and position to the server. When initially connecting to the server, the client should first provide the information of its properties, through a POST call. The server will then add the vehicle to the InfluxDB to keep track of the vehicle's information. The vehicle should be able to perform a GET request to the server to retrieve its information.

At this stage, the client would be able to send a PATCH request to the server to update its information. In addition, the server will add a new entry to the database whenever it receives this request. The idea was that the client and server would be able to continuously communicate with each other, creating a live feed of the clients' behavior.

However, a RESTful API server could not perform all the required tasks the group wanted the server to do. Firstly, the server was only able to communicate with one client at a time, i.e., the client that sends a request. What the group wanted was that the server could respond to other clients without a request. In our case, whenever a vehicle sends a PATCH request, the server should be able to inform other vehicles of this event. This is not possible with a RESTful API.

A new solution had to be in place to achieve our goal, so we consulted our supervisors for help. They proposed some solutions for us to research, including long-polling, Webhooks, and Webscokets.

3.2.1 Alternative solution 1 - Short-polling and long-polling

Polling refers to the server pushing resources to the client. There are mainly two types of polling; short- and long-polling.

When short-polling, a client requests a resource from the server, and the server responds with an empty response if the resource is not available. The client will then send a new request after a short amount of time, and the cycle repeats until the client receives the resource it has requested.

Long-polling is similar to short-polling, but the server does not send anything back until the resource is available. In other words, the client sends a request to the server, and the server holds this request until it has a response available to the client. In our case, we wanted every client to perform a GET request to the server. Then the server holds onto this request until it has further instructions for the requesting client.

3.2.2 Alternative solution 2 - Webhooks

Webhooks, according to **webhooks**, is a user-defined callback over HTTP. In our case, implementing webhooks to post notifications on clients based on events sent to the server. This was a good contender to solve our issue. However, we found little information on how to utilize webhooks in our project.

3.2.3 Alternative solution 3 - Websockets

Websockets is a protocol that provides a bidirectional communication between clients and server by establishing a single TCP connection in both direction (**rfc_websockets**).

3.3 Phase 3 - Websocket with SignalR

After exhaustive discussions on how to solve the two-way communication discussed in 3.2 Phase 2 - REST API and why it did not work with our project, the group agreed that websockets would be a good solution to our problem. Using websockets both client and server can transfer data whenever they see fit. However, **microsoft_websockets** discourage developers from implementing raw websockets for most applications, and recommends using SignalR instead.

ASP.NET SignalR is a library that at the top layer provides real-time communication using websockets while also provides other transport methods such as long polling as fallback (**microsoft_signalr**). Furthermore, SignalR API supports *remote procedure calls* (RPC) using hubs, meaning we can invoke subroutines on the client from the server and vice versa (**microsoft_signalr**).

During this phase we disregarded our old REST API server and InfluxDB completely. First, setting up an echo server using SignalR, while simultaneously implementing the client code. The client code is required to be implemented independent from the Raspberry Pi code because our goal was to create a communication module that could be reused through inheritence for other devices,

e.g. traffic lights, should it be required to set up a new hub with other devices.

After successfully implementing all the necessary methods on the client. The vehicle class that represented the Raspberry Pi device was created. Vehicle class inherits the client class which gives it the ability to connect, listen and send data to the server. Furthermore, the client can also subscribe to events that the server can trigger using RPC.

After witnessing a successful connection between the Raspberry Pi vehicle and our SignalR server we started to implement necessary functionality on the server. A simple scenario was first taken into consideration when we first developed new functionalities. The client will inform the server whenever its velocity has been changed. In this case, the server should informs every vehicles behind that vehicle on the same road to adjust their own velocity accordingly. As a result of this functionality, we are required to continuously keep track of the vehicle´s position. Thus, raising a new issue on how the vehicle information should be stored.

InfluxDB could in theory be used to store the vehicle's position however, since the position is constantly changing it would require the server to continuously read and write on Influx. Hence, the group concluded that in theory this will impact latency on server responsiveness. Thus, unanimously we determined that a live-in-memory database using lists would be better. Using simple mathematics the server could recalculate the vehicle's position based on its previous velocity and a stopwatch whenever it retrieves the information of a vehicle instead.

==========

3.3.1 Implementation of intersections

We now felt ready to show the product owner from Accenture what we had been working on, so we invited him to the next meeting with our external supervisors. This short demonstration consisted of us running the client program two times with the server running, effectively making a wholly digital simulation of two cars on the road. This simulation showed that, by implementing two way communication with SignalR, one car slowing down could trigger the server to ask all cars behind it to slow down too. We got some positive feedback on this demonstration, but were challenged by our supervisors to make our solution more complex. They believed implementing an intersection where multiple cars meet could show more advanced management of traffic.

Expanding further on the concept of calculating a vehicles position, new functionalities on the SignalR server was developed to handle vehicles approaching an intersection. With a more complex topology we were also required to expand the database to account for the new road network. Hence, road models and intersection models were created to represent these concepts. Furthermore, the vehicle model on the serverside now also composed of a route planner to represent what it means to be approaching an intersection.

==========

Expanding further on this concept, new functionalities on the SignalR server was developed to handle vehicles approaching an intersection. With a more complex topology it is also required to expand the database to account for the new road network. Hence, road models and intersections models were created to represent these concepts. Furthermore, the vehicle model on the serverside now also composed of a route planner to represent what it means to be approaching an intersection.

With these improvements, SignalR server proves successful in establishing communications with clients. In addition, with the implemented functionalities the server is able to command the clients to adjust their velocity to avoid collision between vehicles approaching an intersection simultaneously. By reducing velocity of some vehicles it also became apparent that traffic flow is improved, in contrast to stopping a vehicle.

3.4 Phase 4 - Semi physical demonstration

At this point in the development we were sure about what kind of situation we wanted to simulate to make a satisfying product for Accenture, and to answer the problem statement we had decided on. Following the work requirements we now needed to make a demonstration that showed how the system worked. At this point we also decided that we wanted to make a physical demonstration instead of making a digital simulation, although this was a solution that would take less effort and still be a valid solution. At the start of this work phase we had started to consider making only a digital simulation, but after a meeting with our external supervisors at Accenture, in which we were advised that a physical demonstration was more in line with Accenture's goals for the project, we finally decided to make a physical demonstration.

3.4.1 Building the car

The previous group had only built one car for their project. To show a situation where two cars meet at an intersection we needed to build a new car. Luckily, Accenture kept a box of unused components from the previous group. However, we only had one Tpu, the Coral Usb Accelerator. This was an important component for giving extra processing power to the computer, and it was necessary to run the artificial intelligence the previous group had used (source from previous project).

Without this accelerator we could not run the artificial intelligence that the previous group had made. Due to the global chip shortage caused by the Covid pandemic, the accelerator was not available to purchase anywhere. This also made the camera and distance measuring sensor redundant, as these used artificial intelligence to process data. Not having two cars that utilized artificial intelligence could be a challenge, because one of the required features of our solution was that the cars should be able to override the server. We decided that as long as we had one car that could override the server commands the other car could drive solely on commands from the server. With the components, and the product documentation of the previous project (source from previous project), we were able to build a copy of the car as seen in Figure 3.2.

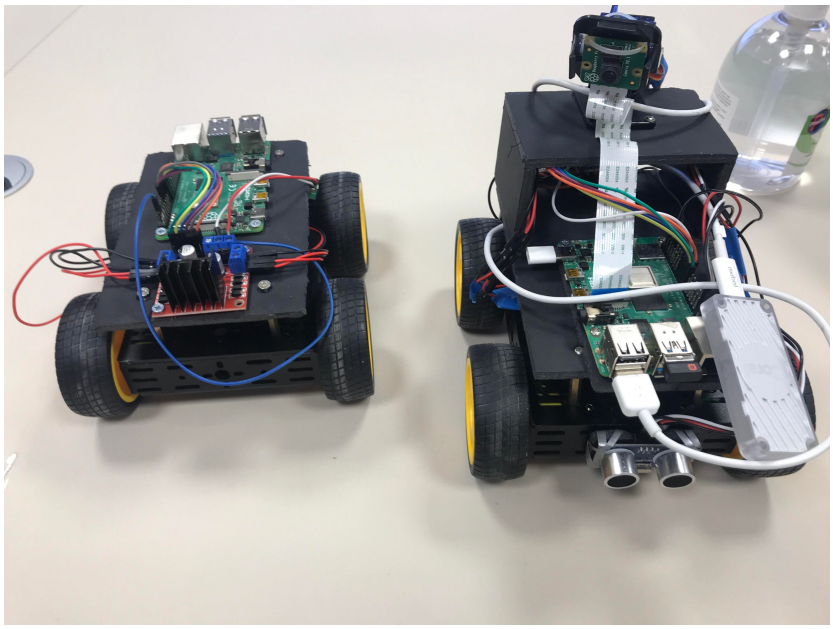


Figure 3.2: Figure of the two cars we built. To the left is the car without a camera. To the right is the car with a camera on top, and a Coral USB Accelerator Edge Tpu. This car can take advantage of the on-board AI.

Power (?)	Length (cm)	Time (s)	Velocity (cm/s)
40	467	8.98	52.00
50	425	7.28	58.38
60	400	6.06	66.01
70	357	5.18	68.92
80	325	4.49	72.32
90	314	4.03	77.92
100	286	3.62	79.01

Table 3.1: Test text

3.4.2 Calibration of the cars

When the server, vehicles and client were implemented, and the second vehicle built, we did some testing to figure out how the car's behaved when given directions by the server. In this test the vehicles were given a specific velocity and driving distance by the server. When the vehicles arrived at their destination the server would tell them to stop. The car's drove in a straight line.

The vehicles were able to send information, and respond correctly to the servers commands. We also observed that the vehicles drove a different length for each velocity given even though the length was the same. This is because the velocity given to the vehicles is the amount of power going into the car's motors, not the actual velocity of the car's. We wanted the demo to be accurate so our group did some further testing where we wrote down the results.

The data Power was the velocity given by the server. Velocity was the actual velocity in our testing, which is length divided by time. As you can see the velocity was not the same as the power. We then made a graph to visualize the two values. The y-axis was the velocity while the x-axis is the power.

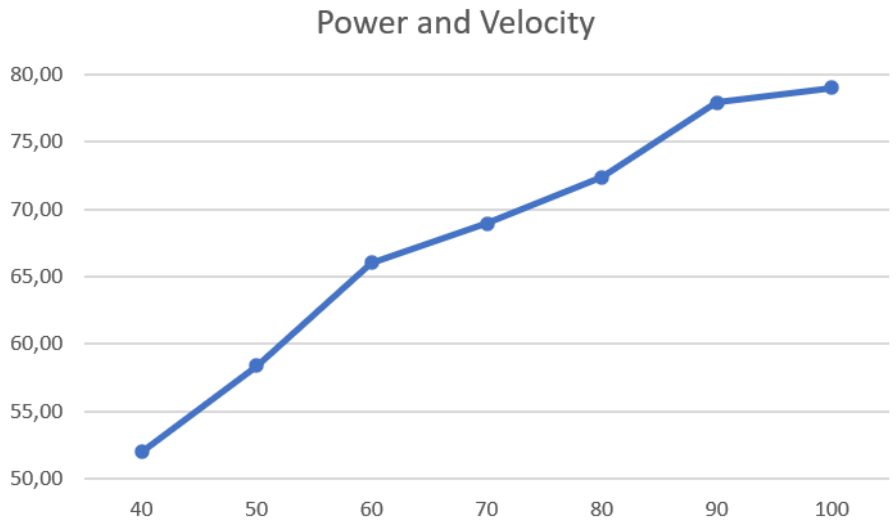


Figure 3.3: Graph of velocity as a function of power

We observed that the correlation between power and velocity seemed linear. This means we could make a specific formula that describes the correlation between the two values. We used linear regression to figure out this formula:

The formula we ended up with was as follows: $v(P) = 0.4516 \cdot P + 36.189$, where P is power and v is velocity, with a mean square error of $R^2 = 0.9653$. When we coded the formula into the vehicles we did another set of testing. We observed that the vehicles drove more or less the same distance for each power given. If we wanted an even more accurate formula we could have tuned the formula with the test results from our new test. Although the results were not hundred percent accurate, we concluded it was accurate enough for our demonstration.

To test the solution we have worked on, we made a physical demonstration with two cars that meet at an intersection, as part of the product documentation. We want to test that a combination of a centralized communication system and artificial intelligence can improve traffic flow. What we wanted to observe was if the velocity of the vehicles were not drasticly changed and therefore not distrupting the traffic flow.

3.4.3 Construction of semi physical demonstration

We found a space at accenture that was big enough to build the track. Because of the limitations of the raspberry pi were the power given to the vehicles could not be under 40 and over 100 (We are not sure what the metrics is for power), we needed a road that could be over three meters long. If the road was under three meters the vehicle that had be given a power by the server which were under

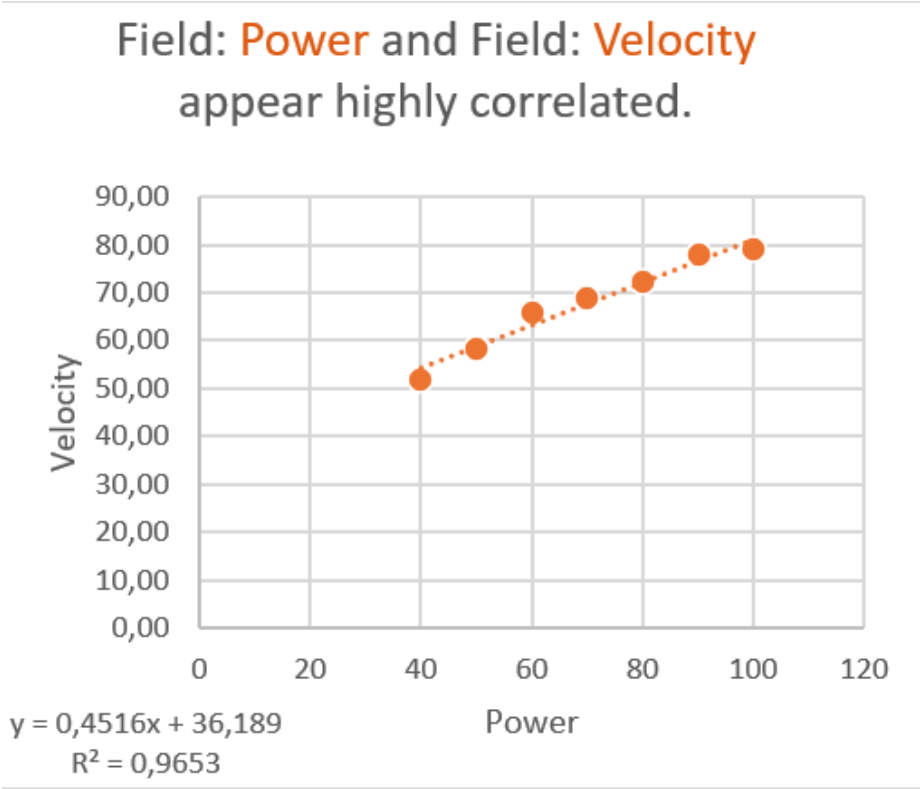


Figure 3.4: Graph of velocity as a function of power with linear regression

the limit. The server works in a way that the demo would not start until both vehicles had connected to it. Theoretically this means that the vehicles should start at the same time. We also placed the vehicles at the same lenght apart from the intersection so that they would crash if the server did not intervene. This way we could know for sure that the server were giving directions to the vehicles. We also made the server log the velocity it sent to the cars so we could keep track of how much the cars velocities changed.

Makeing the demo one hundred percent accurate were not possible ij our circumstanced. This is beacause of the limitations of the raspberry pi. As mentioned it was not a vast gap between the lowest and the highest velocity of the vehicles. The vehicles were not able to recieve the messages at the excact same time as well. This ment that they could start with a difference of half a secound. Another factor was that the vehicles were not always moving completely straight forward which the server assumed. However we were able to get a consistent demo with enough margins. We made vast margins by making a buffer zone around the vehicles. The buffer zone was about twenty centimeters. An example of a non-successful demo can be seen in figure 3.6.

After we implemented the buffersone we were able to get a consistent semi physical demonstration. Here is how a succesfull demo would go:

When both cars connected to the server, they startet at the same velocity which

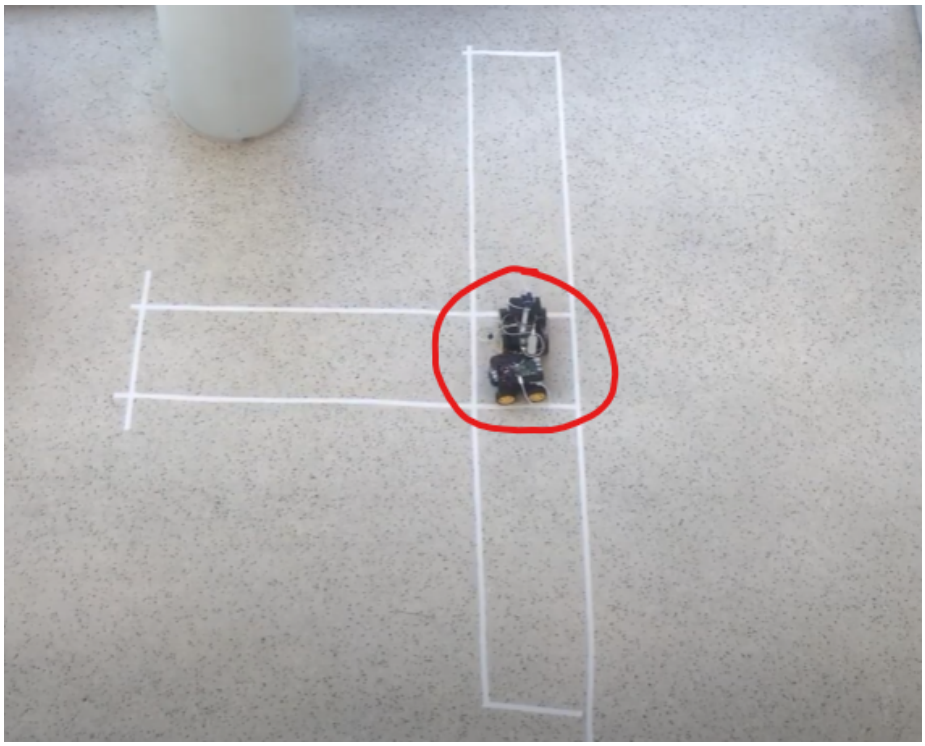


Figure 3.5: Here we can observe the two cars colliding in the intersection. The colliding happened because one of the cars started about 5 centimeters further behind than the other car as well as starting a few millisecounds after. Meanwhile the server assumed they startet at the excact same time at equal distance to the intersection. Those small errors led to a crash because we did not have enough margin for error in our demo yet.

were specificly set to 80cm/s in our demo. We decided on this velocity because it was the highest velocity that the Raspberry Pi cars could have, according to our previous measurments. Not long after they started to drive, the server recognizes that the cars are near the intersection. Then the server calculates which car has to slow down and how much the car needs to slow down to avoid collision. The server calculates using the cars velocity, position and length. In our case the server calculated the velocity 55 cm/s which was the lowest possible velocity according to our measurements. Further the server sends its calculated velocity to the car that needs to slow down. The car that needs to slow down is the car that would enter last into the intersection. After the other car has supposedly passed the intersection the car that slowed down gets told by the server to speed up to its original velocity. An example of a successful demo can be seen in figure 3.7.

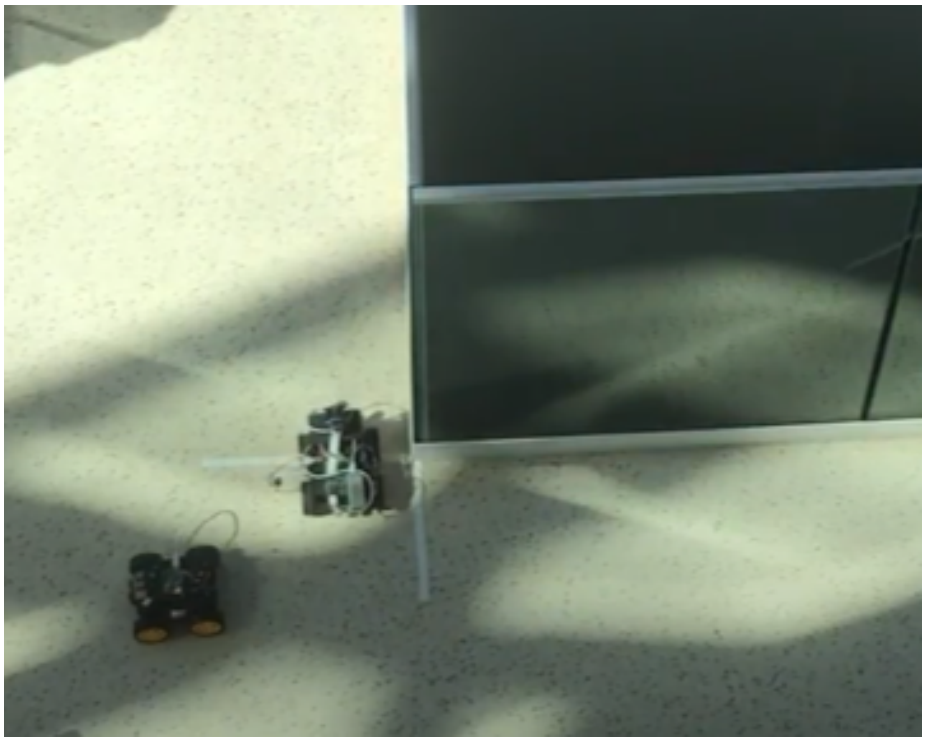


Figure 3.6: Here is a snipped from a successful demo. The car to the left has just passed the intersection which is marked by the white tape. The car furthest up is therfore about to accelerate up to its original velocity. Here we can observe that the margins where big enough to prevent the cars colliding

3.5 Results

After finishing the fourth development phase we showed the video snippets we had filmed to our supervisors and the product owner, in addition to giving an in-depth explanation of the outcomes the demonstration showed. They seemed pleased with the product, and the feedback we got was positive. The product owner confirmed that our results adhered to the requriements set.

The summary of the results from the process was:

The summary of the results from the process was:

The server where multiple vehicles can communicate with each other. The server functions to increase traffic flow and has responsibility for calculations and decision-making.

A client program that enables the two cars to connect to the server.

A semi-physical demonstration that shows how the system works in an intersection.

The technical description of how the applications work is in the product documentation.

3.5.1 Data

The data we can extract from this project comes from our semi-physical demonstration. We observed throughout multiple semi-physical demonstrations that even if the cars started at the same distance from an intersection and had the same initial velocity, they would be able to pass the intersection without having to stop. Stopping is unnecessary because of the server's intervention. An example of a successful demo is in figure 3.6. The server intervened by telling the car furthest from the intersection to slow down so that the other car could pass. The server's log showed that the car that slowed down before the intersection had changed its velocity from 80cm/s to 55 cm/s. The car had then returned to 80cm/s after the other car had passed the intersection.

A change in velocity can lead to a disruption of traffic flow. Therefore, we can conclude that in the specific scenario shown in the semi-physical demonstration, the server improved traffic flow.

3.5.2 Real world scenario

We compared the data from the semi-physical demonstration to a real-world scenario where instead of 80 cm/h, the cars would drive at 80 km/h before an intersection. In the real world, one of the cars would have to stop before a traffic light while the other could drive past the intersection. That would lead to a velocity change from 80 km/h to 0. In our scenario, the car would only have to slow down to 55 km/h.

There are many factors to consider in a real-world scenario, such as curved roads, human mistakes, and animals jumping onto the road. Therefore the data extracted from the demonstrations can not perfectly correlate with a real-world scenario.

3.5.3 Possible improvements

There are many factors to consider in a real-world scenario, such as curved roads, human mistakes, and animals jumping onto the road. Therefore the data extracted from the demonstrations can not perfectly correlate with a real-world scenario.

All the requirements in the MoSCoW method from "could have" to "won't have this time" are requirements for a future project, either for Accenture to improve or for a future bachelor project. Although fulfilled the product requirements given by Accenture, there is room for improvement. In addition, some improvements can be made with the accuracy of the demo, specifically by doing more calibration tests and making a more accurate formula than we were able to produce. A more accurate demonstration can also be made by changing some of the wheels so that the vehicles can drive more straight.

Our demo only contains two cars, but our server is built in a way where multiple vehicles can connect to it. There are also possibilities to connect other devices to the server, for example, traffic lights. However, there are no specific functionalities regarding traffic lights on the server. Scaling the IoT system for functionalities with traffic lights is a task for future development. Adding roads and making a more complex road system would also be a task for future development.

As mentioned, we made a semi-physical demonstration. To extract more data, making a virtual simulation would be sufficient. In a virtual simulation, multiple scenarios could be tested with more vehicles.

Making a viable product in the real world is a long way ahead. That will require a lot more testing and implementation on a bigger scale. However, we hope that our testing and research can be of value towards that step.

Chapter 4

Discussion

In addition to implementing the program, we also had discussions regarding the viability of the program. Here we will discuss our own process and how such and IoT-system would apply to the real world. We will use our own data as well as some research to reflect over the usability of an IoT system where self driving cars can communicate with eachother.

4.1 Self evaluation

As mentioned earlier, we used inspiration from two agile frameworks: scrum and kanban, but chose to lean more towards scrum in the end. We felt the use of scrum helped us reach our goals. However, we could have included our external supervisors more in the scrum process in hindsight. We could have done this by including them in sprint retrospective meetings and discussing what our following sprint goals should have been together.

We satisfied most of the product goals and requirements, although some extra hours were needed towards the end. Because of the time constraint, we found it difficult to focus on scalability. The IoT system can handle more vehicles, but the road model does not fully support functionalities for having more roads than we currently have in our demonstration. We focused on getting a working demonstration rather than making it scaleable.

One challenge in the process was that our group could not meet as much as we had wished because of work. If we had worked more throughout the process, the need for extra work, in the end, could have been prevented. In the MosCoW method (figure 3.1), we were able to finish all the requirements in "must have" and "should have" sections, but none of the "can have" sections. All in all, the group was satisfied with the process.

4.1.1 Educational Value

Developing said program has been a challenging process, which our group has learned a lot from. We all feel that we have evolved into better developers,

and that we now would be better suited for solving such a project in the future.

The project description we were given was very open to interpretation, which gave us a lot of room for exploration. We chose to go with an IoT-solution using the car Accenture already owned, and also building a new one. This choice has given us significant insight into the making of IoT-systems and ways to set up communication between them. We believe this resulted in a much more interesting demonstration for Accenture to showcase. Further, how to combine such a system this with artificial intelligence has been a very interesting learning possibility. Due to the time constraints of such a project we, for instance, did not get to develop our own AI model, that could have helped the server make decisions based on optimal rulesets. This is an improvement that can be considered for further work.

While we still were in the first development phase, we decided to adopt the code of the previous group, instead of developing our own AI model. We also believe that this was the right choice, as we believe it would have been a big time sink to start from scratch. Although it was a challenge to understand the program the previous group had written our assumption was that this way we would get to focus more on our demonstration. The importance of good documentation has also been made clear to us, which has benefited our own documentation.

Our group had little to no experience working with an agile work methodology. In retrospective we believe we benefitted from this choice. Our daily stand ups allowed us to have a clear vision of the group's collective challenges. Although we did not implement all aspects of scrum-development or kanban-development we have gained insight into how a small development team can structure and plan out its work flow in an agile manner, that enables frequent changes in g

4.2 Real world application

How to implement self-driving in society in the best way is a question that will take a long time to answer. Through our work with this proof-of-concept we believe we have made an addition to this discussion. Due to time constraints, we have chosen not to implement some features that would make the product work in a more complex environment. These features could be explored if this project were to be further developed and applied to the real world.

4.2.1 Edge computing and AI

In the future when self-driving vehicles become more prominent, and the 5G network becomes more available there could be a possibility for IoT-systems handling traffic management. Our group therefore did some research regarding how the system will extend to the real world's applications.

The IoT systems usually follow the fog or edge computing architecture with distributed or even decentralized concepts (**iot_platforms**). This is to prevent overloading on servers handling a lot of data.

One solution to distribution of data is that each road has one server responsible for their respective road. If a road is long it will be split into geographical areas where one server has responsibility for their geographical area. Intersections will have a server handling information from both road's respective servers, since information from both servers are needed to make decisions in intersections.

In the traffic there are a lot of unforeseen situations that can happen. Traditional coding will not be able to cover every outcome in a traffic situation, therefore there will be a need for AI on the servers in addition to the cars. The AI will probably need to train in a safe test enviroment just like the automomous vehicles had to, before it can be released into the real road system.

In our IoT-program, the server gives commands to the cars, which overrides the cars AI. However, in the real world there will be a need for an interaction between the AI for the server and the AI on the cars.

4.2.2 System security

Security and privacy are important topics for any IoT-system. Because these systems gather and work with huge amounts of data, they are naturally prone to being attacked. And as the systems grow and become more interconnected, with many devices around the world, the imposed risk of such an attack increases drastically [kilde på dette].

Therefore a lot of security measures needs to be implemented in the real world application such as anonymization of personal data, securing connections and a intruder detection system. As mentioned earlier, the cars can drive both with or without the system. This is an importaint feature in system security. That is because if there is a need for the server to shut down, the cars would need to be able to drive on their own as well.

4.2.3 Distribution to the real world application

PART III

PRODUCT DOCUMENT

This text is supposed to convey what this part of the report is about.

Chapter 5

Product documentation

5.1 Preface

The product documentation is a technical description of the product.

The product documentation assumes the reader has some prior knowledge in basic programming.

5.2 Program description

The program is an IoT-system were vehicles can communicate with eachother through a server. It constists of a server which has the responsibility for the calculations and decision making, and a client which are the vehicles. The function of the program is to increase traffic flow and prevent traffic congestion. As of now the program can be used in an intersection with two vehicles, but can be scaled to include more vehicles. Here is a diagram that describes the flow of our server in the demonstration:

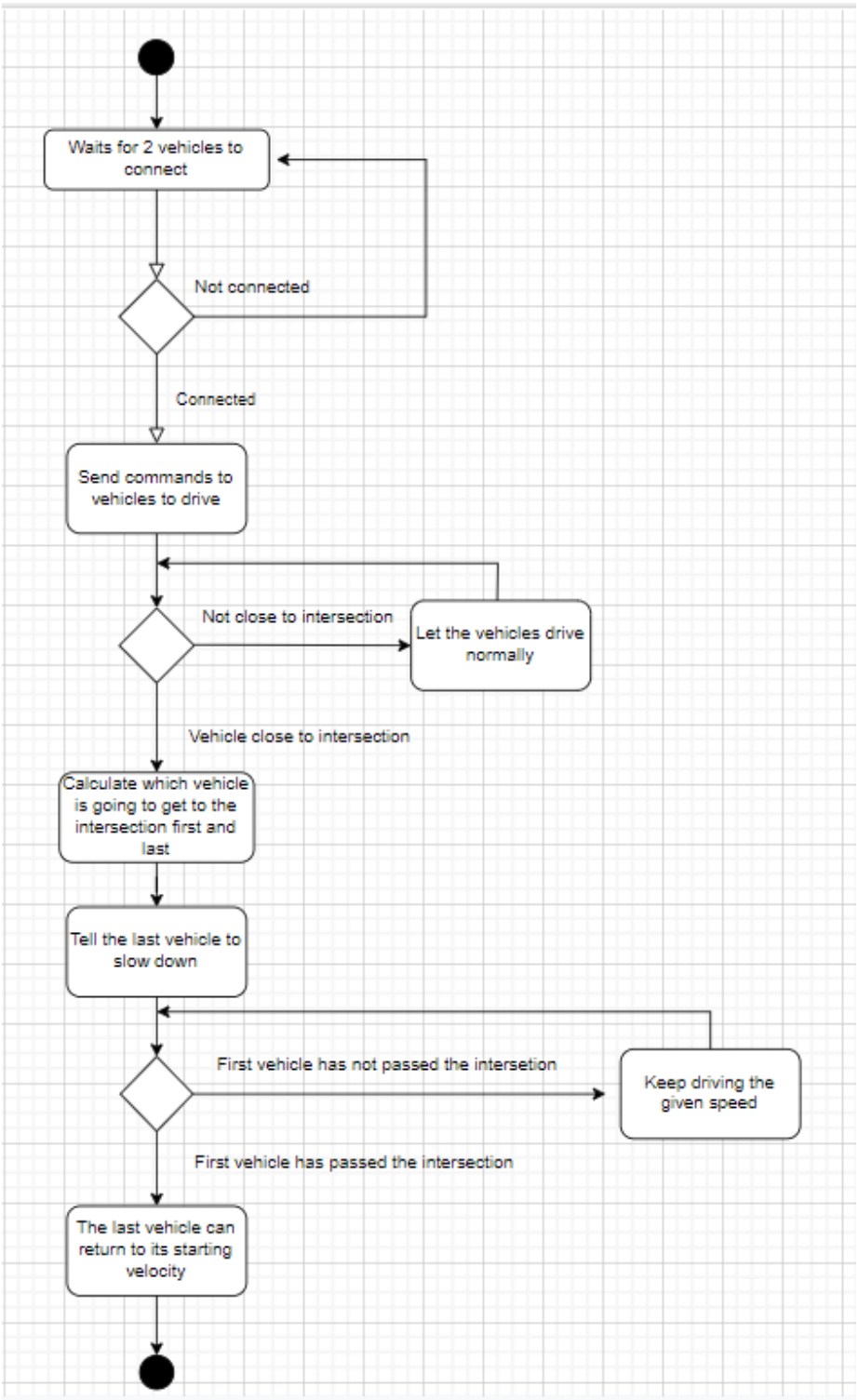


Figure 5.1: Flow diagram for the server. This diagram specificly shows the flow of the semi physical demonstration.

5.3 Adherence to project requirements

Our solution adheres to the requirements Accenture set for us. Our prototype cars can work both with, and without being connected to a server. The server can handle multiple connections and adjust trafic based on the situation on the road. Through er deomnstration we also showed a situation where the outcome differs depending on if the cars are connected to the server or just driving on the on-board AI.

5.4 User manual

We have written a manual for people who want to recreate our demonstration. The demonstration could for example be shown off at exhibitions. The manual could also be used for people who want to further test and develop our IoT-system.

First check the IP-address of the internet you are connected to, and the usable ports. Make sure that your computer hosting the server and the vehicles are connected to the same network.

```
$ipconfig getifaddr en0
192.168.56.208
```

Then open the Server solution in your code editor, we have used visual studio. Under the folder "properties" there is a file called launchSettings.json. In that file write in the ip address and the port in the applicationUrl-section:

```
"profiles": {
       "SignalRServer": {
              "commandName": "Project",
              "dotnetRunMessages": true,
              "launchBrowser": false,
              "applicationUrl": "https://192.168.56.208:7058;http
                  ↪ ://192.168.56.208:5048",
              "environmentVariables": {
                     "ASPNETCORE_ENVIRONMENT": "Development"
              }
       },
```

After that open the Client solution. Here we used Pycharm as the IDE. Open the config.json document and write in the same ip-address and port:

```
"client": {
       "host": "192.168.56.208",
       "port": 5048,
       "delay": 0.1
},
```

If the vehicles haven't connected to that network before, they need to log on that network. To log on to a new network you will need to connect the raspBerry pi's to a screen. That could be done via the micro usb-port at the raspberry pi.

When the raspberry pi has booted up, click on the internet-icon and connect to the same network as the server. If you have connected to the internet it will be saved and the vehicle should connect to that internet automatically when booting up.

If you want to change the velocities of the vehicles it can be done here in the server:

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
        private readonly Intersection _intersection;

        private readonly HashSet<Vehicle> _vehicles = new();
        private readonly HashSet<Lane> _lanes = new();
        public int Count => _vehicles.Count;

        private readonly Dictionary<string, Vehicle>
            ↪ _connectionIds = new();
        private readonly Dictionary<Vehicle, string>
            ↪ _vehiclesConnectionId = new();
        public Dictionary<Vehicle, Thread?> VehicleThreads { get;
            ↪ }= new();
        public Thread? GlobalThread { get; set; }
        public Stopwatch Clock { get; } = new();

        public double SpeedLimit => 80;
```

The file is located at VehicleHubDatabase under the Database folder. The variable you want to change is the SpeedLimit.

Right under you can change the length of the roads:

```
public VehiclesHubDatabase()
{
        _intersection = new Intersection().
                AddRoad(new Road {Length = 300}.
                        AddLane(null, true).
                        AddLane()).
                AddRoad(new Road {Length = 300}.
                        AddLane(null, true).
                        AddLane());

        _intersection.ConnectedLanes().
                ForEach(lane => _lanes.Add(lane));
}
```

The lenght is in centimeters and needs to corrolate with the lenght of the physical track. We used tape to show were the roads were, however this is not necessary. For the best results, we recommend a track between three to four meters long.

We have written more about the specifics of the system in our product documentation. Then place the vehicles down at the start of the track, turn on

the server and connect to the power banks. The vehicles should automatically connect to the server after 20-30 secounds. The demonstration starts when both vehicles has connected to the server.

5.5 Essential code snippets behind the system

The main part of this project is to develop a client-server communication system, with the purpose of producing a physical simulation on how a centralized system can contribute to an improved traffic flow. Due to the nature of this project, no graphical user interfaces has been developed. Hence, it is deemed necessary to present key parts of the code that are responsible for such a system to work. This section will therefore elaborate, in detail, how essential code snippets are interacting with each to produce the result.

Furthermore, the code that has been written during this project has been written in the languages Python and C# using Pycharm and Rider IDE respectively. Therefore, syntax highlighting has also been used to best simulate the same syntax highlighting used in both IDE respectively. In addition, some artistic freedom has been used to present the code snippets; the symbol ... has been used to indicate irrelevant code to the current discussion and the symbol ↪ simply means that the line of code following ↪ is on the same line above but is broken up due to lack of space. Also, each code snippets starts with the class and method it belongs to.

5.5.1 Initialization

Client.py

The client package is the main module in the Raspberry Pi vehicles. The class that is mainly responsible to connect, handle and sending data to the server is the client class in client.py. Client class is not meant to be used alone but rather as a super class for other IoT devices. Hence, it was developed with the intention to be inherited and handle everything that pertains to client-server communication in the background.

Client's init method does several things: It reads from the config.json file to store the defined host and port it is going to connect to.

```
class Client:
  def __init__(self, properties=None, **kwargs):
    ...
    with open("client/config.json") as f:
      config = json.load(f).get('client')
      if config is not None:
        self.__uri = f"://{config.get('host')}:{config.get('port')
            ↪ }/{self.__class__.__name__.lower()}sHub"
        self.__delay = config.get('delay')
    ...
```

Then, it starts a negotiation proccess with the server where it receives a connection id that the client will use during its connection lifetime to the hub.

```
class Client:
  def __init__(self, properties=None, **kwargs):
    ...
    urllib3.disable_warnings()
    response = requests.post(f"http{self.__uri}/negotiate?
        ↪ negotiateVersion=0", verify=False)
    self.connection_id = response.json().get("connectionId")
    self.websocket_uri = f"ws{self.__uri}?id={self.connection_id}"
    ...
```

The client also gives itself a random id that is stored as one of its properties. The client's id is also stored on the server and is mostly used to retrieve and update the client's information on the server.

Furthermore, `Client.__init__` also stores a dictionary of events.

```
class Client:
  def __init__(self, properties=None, **kwargs):
    ...
    self.subscribed_events = {
      "disconnect": self.disconnect,
      "force_patch": self.force_patch,
      "continuously_patch": self.continuously_force_patch
    }
    ...
```

The values of this dictionary is a reference to a function in this class and is used to invoke certain behaviours by the server. For instance,
`await Clients.Client(Context.ConnectionId).SendAsync("disconnect");`
from the server will call `def disconnect(self)` in the client.

Vehicle.py

The vehicle class is supposed to contain all the data and methods of the vehicle. Furthermore, `class Vehicle(Client)` inherits the client class which enables Vehicle to perform all the necessary operations to establish connection upon initiation. An important remark is that `Client` performs the negotiation to the server using endpoint
`{self.__class__.__name__.lower()}sHub/negotiate?negotiateVersion=0`
meaning that through inheritence and initialization of `Vehicle`, the subclass negotiates with the endpoint `vehiclesHub/negotiate?negotiateVersion=0`, which is mapped in `Program.cs` with

```
...
app.MapHub<VehiclesHub>("/vehiclesHub");
...
```

Furthermore, `Vehicle` also reads from `config.json` to define it's initial properties with the snippet shown below:

```
class Vehicle(Client):
  def __init__(self, properties=None, **kwargs):
```

```
    ...
    if properties is None and len(kwargs) == 0:
      with open("client/config.json") as f:
        config = json.load(f).get('vehicle')
        if config is not None:
          self.properties.update(config)
    ...
```

`Vehicle` also utilizes the property builder of `Client`.

```
class Vehicle(Client):
  def __init__(self, properties=None, **kwargs):
    ...
    self.property_builder(
      required={'length', 'height', 'width', 'mass'},
      optional={'velocity': 0, 'position': 0, 'travel_plan': None
          ↪ },
    )
    ...
```

In short, the property builder is used to define the required properties of the vehicle class. That is, if one should directly initialize Vehicle without using config.json one must assign values to length, height, width and mass. The meaning is to somewhat restrict what data the vehicle class should contain.

Lastly, `Vehicle` adds an additional subscribed events that the server can invoke:

```
class Vehicle(Client):
  def __init__(self, properties=None, **kwargs):
    ...
    self.subscribed_events.update({
      "adjust_velocity": self.adjust_velocity
    })
```

Likewise, as in `Client` should other events be required for vehicle, one can add it to the dictionary as proposed above.

5.5.2 Handshake and listener

After initializing the vehicle class as a client with

```
async def main():
  ...
  client = Vehicle()
  ...
```

then the client's listen method can be called:

```
async def main():
  ...
  listener = asyncio.create_task(client.listen())
  ...
```

The listener method is responsible handling responses and requests from the server. Hence, it is required to run concurrently as the vehicle continuously sends data to the server.

When the listener is called, the client performs the following code:

```
class Client:
  ...
  async def listen(self):
    async with websockets.connect(self.websocket_uri) as websocket
        ↪ :
      self.__websocket = websocket
      await self.__handshake()
      await self.__listen()
  ...
```

As shown above, the method first opens a websocket connection using the stored uri and stores this as a private variable for later use. Then, a handshake with the server is performed:

```
class Client:
  ...
  async def __handshake(self, protocol: str = "json", version:
      ↪ int = 1):
    data = self.signalr_encode_message({"protocol": protocol, "
        ↪ version": version})
    await self.__websocket.send(data)
    response = self.signalr_decode_message(await self.__websocket.
        ↪ recv())
    if "error" in response:
      print(response)
    else:
      await self.send_non_blocking("AddClient", self.properties)
  ...
```

The code above describes the handshake process between the client and the server. First, the client informs the server of the protocols that it will use throughout its lifetime. Then, it also informs the server to store the client, in this case the vehicle, to the server using the defined properties.

Further elaboration, the client invokes the method

```
public partial class VehiclesHub : Hub
{
  ...
  public async Task AddClient(JsonDocument jsonDocument) {...}
  ...
}
```

on the server. This method first creates a vehicle with all the provided information sent by the client

```
public partial class VehiclesHub : Hub
```

```
{
  ...
  public async Task AddClient(JsonDocument jsonDocument)
  {
    ...
    var vehicle = Vehicle.Create(jsonDocument);
    ...
  }
  ...
}
```

using the static method defined by the Vehicle model. In addition, it assigns the travel plan to the vehicle by using values defined by config.json from the client:

```
{
  ...
  "vehicle": {
    ...
    "travel_plan": {
      "start": {
        "road": 0,
        "lane_reversed": false
      },
      "end": {
        "road": 0,
        "lane_reversed": false
      }
    }
  }
}
```

Using the information provided above the vehicle's current lane is also assigned to keep track on which lane the vehicle is driving on. The vehicle is then added to the database, i.e. `public class VehiclesHubDatabase`, together with its connection id `Context.ConnectionId` for easy retrieval.

Furthermore, for the sake of the demo the server is also instructed to wait for a second vehicle to connect before allowing the vehicles to drive

```
public partial class VehiclesHub : Hub
{
  ...
  public async Task AddClient(JsonDocument jsonDocument)
  {
    ...
    vehicle.Velocity = 0;
    _database.Update(vehicle);
    await Clients.Client(Context.ConnectionId).SendAsync("
        ↪ adjust_velocity", vehicle);
    await WaitForVehicles(vehicle, _database.SpeedLimit, 2);
```

```
  }
  ...
}
```

by first setting the velocity of the vehicle to zero, updating the new velocity in the database and also adjusting the velocity of the client to zero. Lastly, it calls the `WaitForVehicles` method which will adjust every client's velocity to the defined `SpeedLimit` in `VehiclesHubDatabase`.

5.5.3 Patch

After initialization and handshake elaborated in 5.5.1 Initialization and 5.5.2 Handshake and listener respectively, the Raspberry Pi vehicles starts to drive into an intersection simultaneously. Throughout the journey the cars are continuously patching to the server, by calling the client's `async def send\_patch` method.

```
class Client:
  ...
  async def send_patch(self, **kwargs) -> None:
    if self.properties_has_changed(**kwargs) or self.
        ↪ __continuously_patch:
      await self.send_invocation("patch", self.properties)
    else:
      await asyncio.sleep(self.__delay)
```

As seen above `send_patch` calls the `async def send_invocation` method, which communicates the vehicle's current information by invoking `public async Task Patch` on `VehiclesHub`.

The patch method on `VehiclesHub` is responsible for handling the behaviour, specifically adjusting the velocity of individual vehicles:

```
public partial class VehiclesHub : Hub
{
  ...
  public async Task Patch(JsonDocument jsonDocument)
  {
    var vehicle = Vehicle.Create(jsonDocument);
    _database.Update(vehicle);
    vehicle = _database.Fetch(vehicle);
    ...
  }
}
```

The snippet above shows that the method first creates a new vehicle using the information provided by the `Client`. However, since this new vehicle does not contain all the information, such as the travel plan, the method first update the existing vehicle in the database in order to refresh the vehicle with the available information. It then fetch the same vehicle that was stored in the handshake, mentioned in 5.5.2 Handshake and listener. Assuming that the vehicle has been successfully retrieved it will then handle this vehicle accordingly:

```
public partial class VehiclesHub : Hub
{
  ...
  public async Task Patch(JsonDocument jsonDocument)
  {
    ...
    await HandleIntersection(vehicle);
    await HandleInsideIntersection(vehicle);
    await HandleEndOfRoute(vehicle);
    ...
  }
}
```

Shortly summarized `HandleIntersection` is responsible to adjust the velocity of every vehicle approaching the intersection to avoid collisions. Furthermore, `HandleInsideIntersection` increases the speed to `VehiclesHubDatabase` defined `SpeedLimit`. Lastly, `HandleEndOfRoute` ensure that any vehicles that has completed their journey, defined during the handshake, terminates their connection with the server.

5.5.4 VehiclesHubDatabase

The `VehiclesHubDatabase` played a key role in this project. Coming to the realization on 3.3 Phase 3 - Signal R the project needed a database that could handle the continuous changes in each `Vehicle` position in real time. During research the group was unable to conclude on any databases that would fit our requirement. Hence, `VehiclesHubDatanbase` was created to serve as a live in-memory database that could continuously update the positions of each vehicle on each lane.

Before starting with `VehiclesHubDatabase`, it was required to define what a road, lane and intersection is, respectively. Thus, `Road.cs`, `Lane.cs` and `Intersection.cs` was developed. Consequently, `VehiclesHubDatabase` was created.

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  private readonly Intersection _intersection;

  private readonly HashSet<Vehicle> _vehicles = new();
  private readonly HashSet<Lane> _lanes = new();
  public int Count => _vehicles.Count;
  private readonly Dictionary<string, Vehicle> _connectionIds =
      ↪ new();
  private readonly Dictionary<Vehicle, string>
      ↪ _vehiclesConnectionId = new();
  ...
  public double SpeedLimit => 80;
  ...
}
```

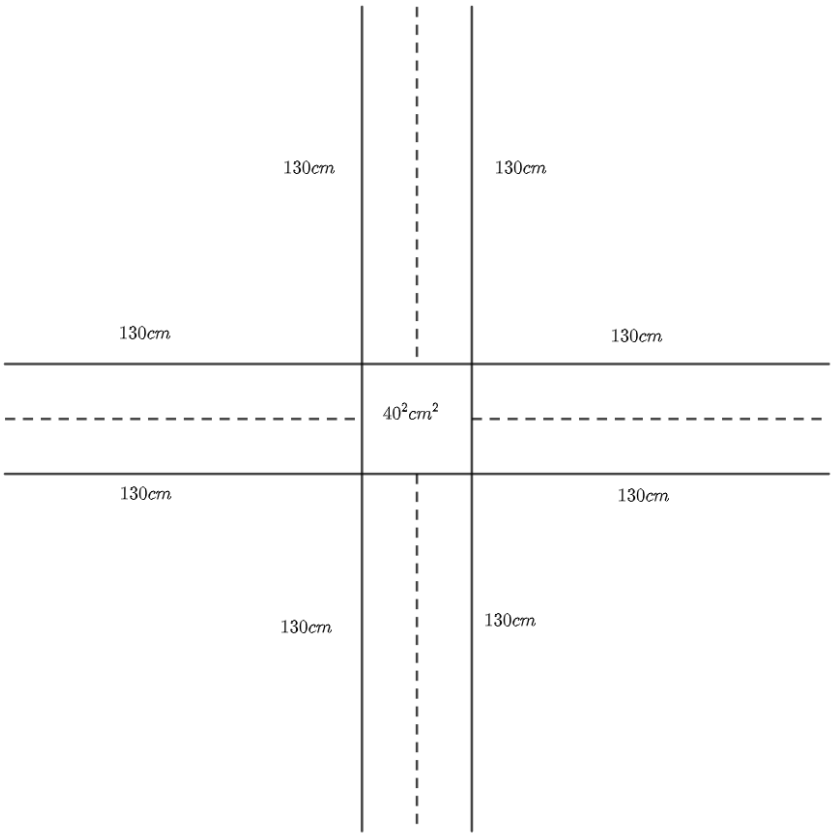


Figure 5.2: This figure shows two roads each with two lanes and a total length of 300cm. The overlapping part forms a square which is the intersection. This configuration was heavily considered when creating representational models on the server, and also used for the demo as a result.

Currently, `VehiclesHubDatabase` only holds one intersection, due to time constraint this was not extended for a configuration with multiple intersections. Moreover, both vehicles and lanes are stored inside a hashset for fast retrieval. In addition, `Count` is used in `WaitForVehicles`, elaborated in sub-section 5.5.2 Handshake and listener, and `_connectionId` and `_vehiclesConnectionId` is used during `Patch` to invoke `adjust_velocity` on individual vehicles. While, `SpeedLimit` defines the upper speed vehicles are limited to on the two roads shown in Figure 5.2.

The road configuration found in Figure 5.2 is defined in the constructor using a builder pattern.

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public VehiclesHubDatabase()
  {
    _intersection = new Intersection().
```

```
      AddRoad(new Road {Length = 300}.
        AddLane(null, true).
        AddLane()).
    AddRoad(new Road {Length = 300}.
        AddLane(null, true).
        AddLane());

    _intersection.ConnectedLanes().
      ForEach(lane => _lanes.Add(lane));
  }
  ...
}
```

Lastly, `VehiclesHubDatabase` is added as a singleton service in `Program.cs` to ensure that we have a static database throughout the lifetime of the program:

```
...
builder.Services.AddSingleton<IVehiclesHubDatabase>(new
    ↪ VehiclesHubDatabase());
...
```

It is also worth to mention some of the core functionalities of `VehiclesHubDatabase`:

Adding vehicles

Calling the `Add` method makes it possible to add vehicles:

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public void Add(Vehicle vehicle, string? connectionId = null)
      ↪ {...}

  public void Add(Vehicle vehicle, Lane? lane = null) {...}
  ...
}
```

Removing vehicles

Removing vehicles can be achieved by calling the `Remove` method:

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public void Remove(Vehicle vehicle) {...}
  ...
}
```

Updating vehicles

Updating either a specific information of a vehicle or all vehicles can be done by calling the `Update` method:

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public void Update(Vehicle? vehicle = null) {...}
  ...
}
```

Fetching vehicles

One can fetch an existing vehicle from the database by passing a vehicle with the same GUID with:

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public Vehicle? Fetch(Vehicle vehicle) {...}
  ...
}
```

Get the connection Id of a particular vehicle

By passing a vehicle into

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public string? ConnectionId(Vehicle vehicle) {...}
  ...
}
```

one can retrieve the connection Id that the given vehicle is using.

Find vehicles approaching the intersection

Maybe the most important feature of `VehiclesHubDatabase` is the two method shown below:

```
public class VehiclesHubDatabase : IVehiclesHubDatabase
{
  ...
  public IEnumerable<Vehicle> NextVehiclesIn() {...}
  public IEnumerable<Vehicle> OnlyFirstIntoNextVehiclesIn() {...}
}
```

The first method `NextVehiclesIn` returns a list of vehicles currently approaching the intersection defined in the constructor, ordered with respect to the vehicle closest to the intersection. The latter method `OnlyFirstIntoNextVehiclesIn` returns an ordered list of the closest vehicle per lane.

5.5.5 RoutePlanner and SetTravelPlan

`RoutePlanner` is a class that determines and keep track of the `Vehicle`'s journey. `RoutePlanner` saves segments of lanes and intersections as nodes in a linked list:

```
public class RoutePlanner
{
  private LinkedList<IRoadComponent> _visitedNodes = new();
  private LinkedList<IRoadComponent> _travelPlan = new();
  ...
}
```

The variable `_travelPlan` stores all the nodes that `Vehicle` is going to traverse through in succession. Whenever `Vehicle` has traversed the whole length of a node, the node is then moved to the tail of `_visitedNodes`. Thus, provides an easy way to keep track of the whereabout of `Vehicle` at all times.

During the handshake, described in 5.5.2 Handshake and listener, `Vehicle` builds the `_travelPlan` through its `SetTravelPlan` method:

```
public class Vehicle : IDevice
{
  ...
  public void SetTravelPlan(Lane startLane, Lane? endLane = null)
  {
    var reversed = startLane.Reversed;
    var startNode = startLane.Node();
    var intersections =
      reversed ?
        startLane.CurrentRoad()?.Intersections.OrderByDescending(
            ↪ kvp => kvp.Key) :
        startLane.CurrentRoad()?.Intersections.OrderBy(kvp => kvp.
            ↪ Key);
    ...
  }
  ...
}
```

By taking in the `startLane` and `endLane`, representing where `Vehicle` should start and end its journey, it first segments the `startLane` into a `LaneNode` and then find all the intersections on this current `Lane`, and order them chronologically depending of the lane is `Reversed` or not. `SetTravelPlan` will further on iterate through all the `Intersections` and append a `LaneNode` and an `IntersectionNode` to `RoutePlanner` for each `Intersection`.

```
public class Vehicle : IDevice
{
  ...
  public void SetTravelPlan(Lane startLane, Lane? endLane = null)
  {
    ...
```

```
    var prevPos = reversed ? startLane.Length : 0.0;
    Intersection? intersectionConnectedToEndLane = null;

    if (intersections != null)
      foreach (var (position, intersection) in intersections)
      {
        startNode.Length = Math.Abs((int) (prevPos - position - (
            ↪ reversed ? intersection.Length : 0)));
        _route.AddComponent(startNode).AddComponent(intersection.
            ↪ Node());
        prevPos = position + (reversed ? 0 : intersection.Length);
        if (endLane == null) continue;
        if (!intersection.ConnectedRoads.ContainsKey(endLane.
            ↪ CurrentRoad() ?? new Road())) continue;
        intersectionConnectedToEndLane = intersection;
        break;
      }
    ...
  }
  ...
}
```

The above loop will iterate until one of the `Intersection` is connected to the `endLane` or until all the `Intersections` are exhausted. In the end, `SetTravelPlan` will append the last segment of `startLane` or the remaining segment of `endLane` should `endLane` either be defined, and connected to one of `startLanes` intersections, or is equal to `startLane`.

The main reason for creating `RoutePlanner` in this project was mainly to answer the following questions:

- Is `Vehicle` currently inside an intersection?
- Is `Vehicle` currently on a lane?
- Which intersection is the next intersection for this `Vehicle`?
- How far is `Vehicle` away from the next intersection?

By using `RoutePlanner` we were able to answer the questions above with these following implementations respectively:

```
public class Vehicle : IDevice
{
  ...
  public bool InIntersection() =>
    _route.CurrentNode()?.Value is IntersectionNode;
  public bool OnRoad() =>
    _route.CurrentNode()?.Value is RoadNode or LaneNode;
  public IntersectionNode? NextIntersection() =>
    OnRoad() ? _route.NextNode()?.Value as IntersectionNode : null
        ↪ ;
  public double ToNextIntersection()
```

```
  {
    if (InIntersection())
      return 0;
    if (OnRoad() && NextIntersection() != null)
      return Math.Abs(Position - _route.DistanceWithCurrentNode);
    return -1;
  }
  ...
}
```

Glossary

Artificial intelligence - A field which combines computer science and robust datasets to enable problem solving (**artificial_intelligence**).

Internet of Things -

Internet of Vehicles - An IoV system is a distributed system for wireless communication and information exchange between vehicles through agreed-upon communication protocols (**chinese_iov**).

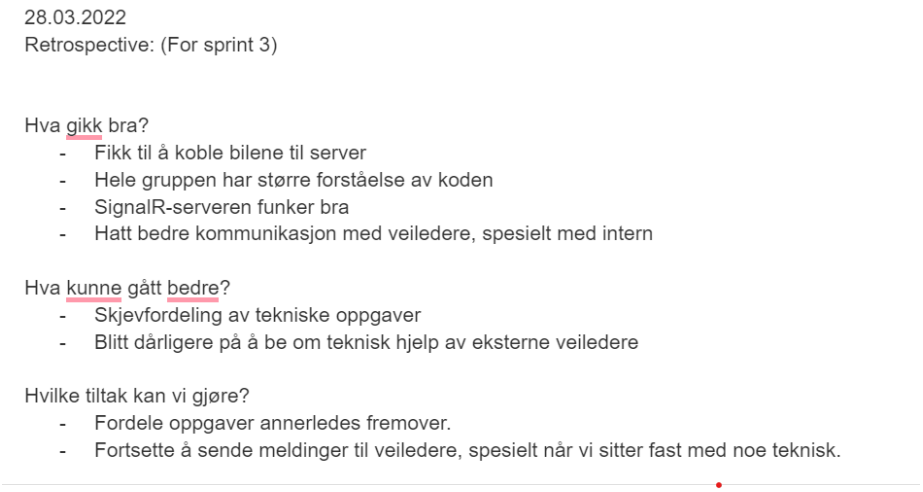
Appendix A

Acronyms

AI	-	Artificial intelligence
IoT	-	Internet of things
IoV	-	Internet of vehicles
REST	-	Representational state transfer
API	-	Application programming interface
HTTP	-	Hypertext transfer protocol

Appendix B

Sprint documents



28.03.2022
Retrospective: (For sprint 3)

Hva gikk bra?
- Fikk til å koble bilene til server
- Hele gruppen har større forståelse av koden
- SignalR-serveren funker bra
- Hatt bedre kommunikasjon med veiledere, spesielt med intern

Hva kunne gått bedre?
- Skjevfordeling av tekniske oppgaver
- Blitt dårligere på å be om teknisk hjelp av eksterne veiledere

Hvilke tiltak kan vi gjøre?
- Fordele oppgaver annerledes fremover.
- Fortsette å sende meldinger til veiledere, spesielt når vi sitter fast med noe teknisk.

Figure B.1: A snapshot of a sprint retroperspective meeting from our log. There are three sections: What went well? What could have gone better? And what can we do for next time?

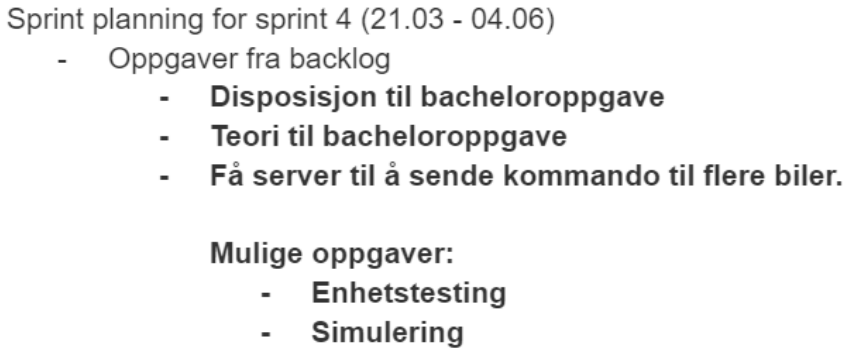


Figure B.2: A snapshot of a sprint planning meeting from our log. The upper section contains the tasks we are going to do from the backlog. The section under contains possible but not necessary tasks.