Institutt for Informasjonsteknologi
Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo
Besøksadresse: Holbergs plass, Oslo

PROSJEKT NR.
23-2022

TILGJENGELIGHET
Offentlig

Telefon: 22 45 32 00

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

Det er utviklet en proof-of-consept løsning for et sentralisert system der en server hvor selvkjørende biler kan kommunisere med hverandre øker trafikkflyt i et kryss. En semi fysisk demonstrasjon er laged for å vise fram løsningen.

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: 50

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, 2021). In 2021, Accenture generated a revenue of approximately $50.3 billion (Accenture, 2022).

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 self-driving cars plus a centralized system versus only using self-driving vehicles. Furthermore, the cars should also be able to drive with the incorporated AI model if the centralized system is out of reach. 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 (Vaagaasar & Skyttermoen, 2021). 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:

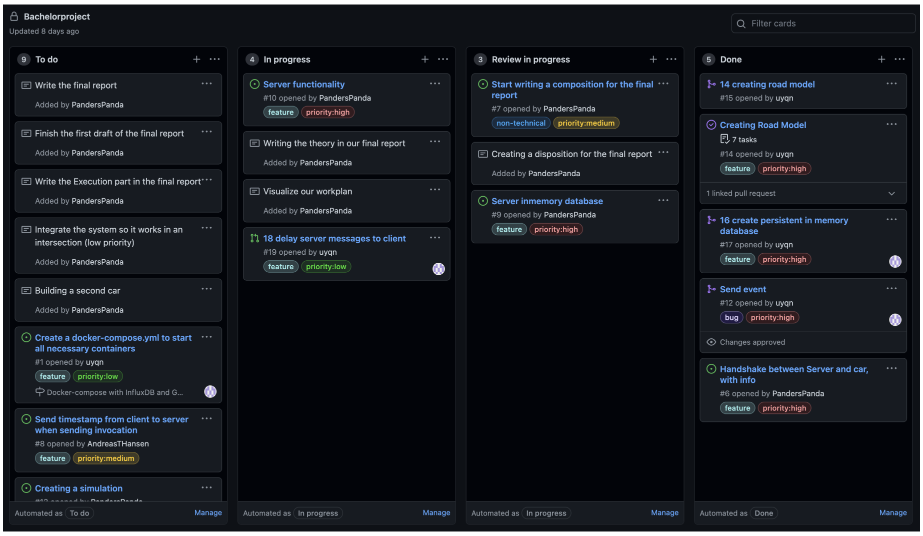

Figure 2.2: Extract of our Kanban board from Github. Each task has a color-coded label representing the priority, and a label to describe the task.

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.

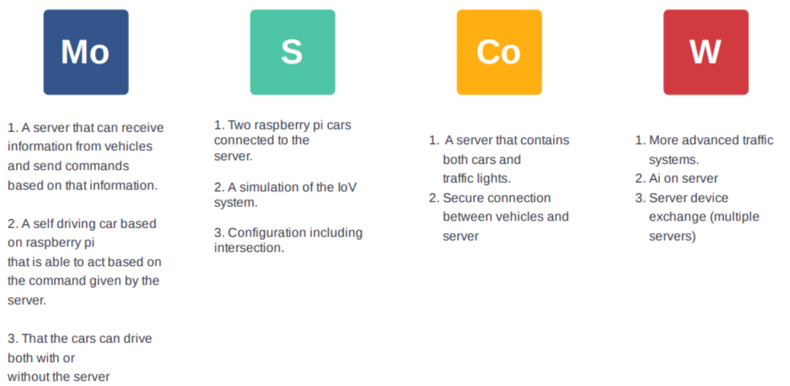

Figure 2.3: Visualuzation MoSCoW method. Everything under "Mo" are requirements we must have in our project, under the "S" sections are features we should have. Under "Co" are features we should have, but are not necessary. Under "W" are features we will not have this time

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 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#.

3.1.2 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 (Sinha, 2021). 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 APEC Delegation, 2014). 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.

3.1.3 Preventing traffic congestions for a one-lane road

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 (Naranjo et al., 2019). Many factors can cause traffic congestion, such as:

poorly designed roads, not wide enough roads, traffic light patterns, and accidents (Long et al., 2008).

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 (Levinson et al., 2021).

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.

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:

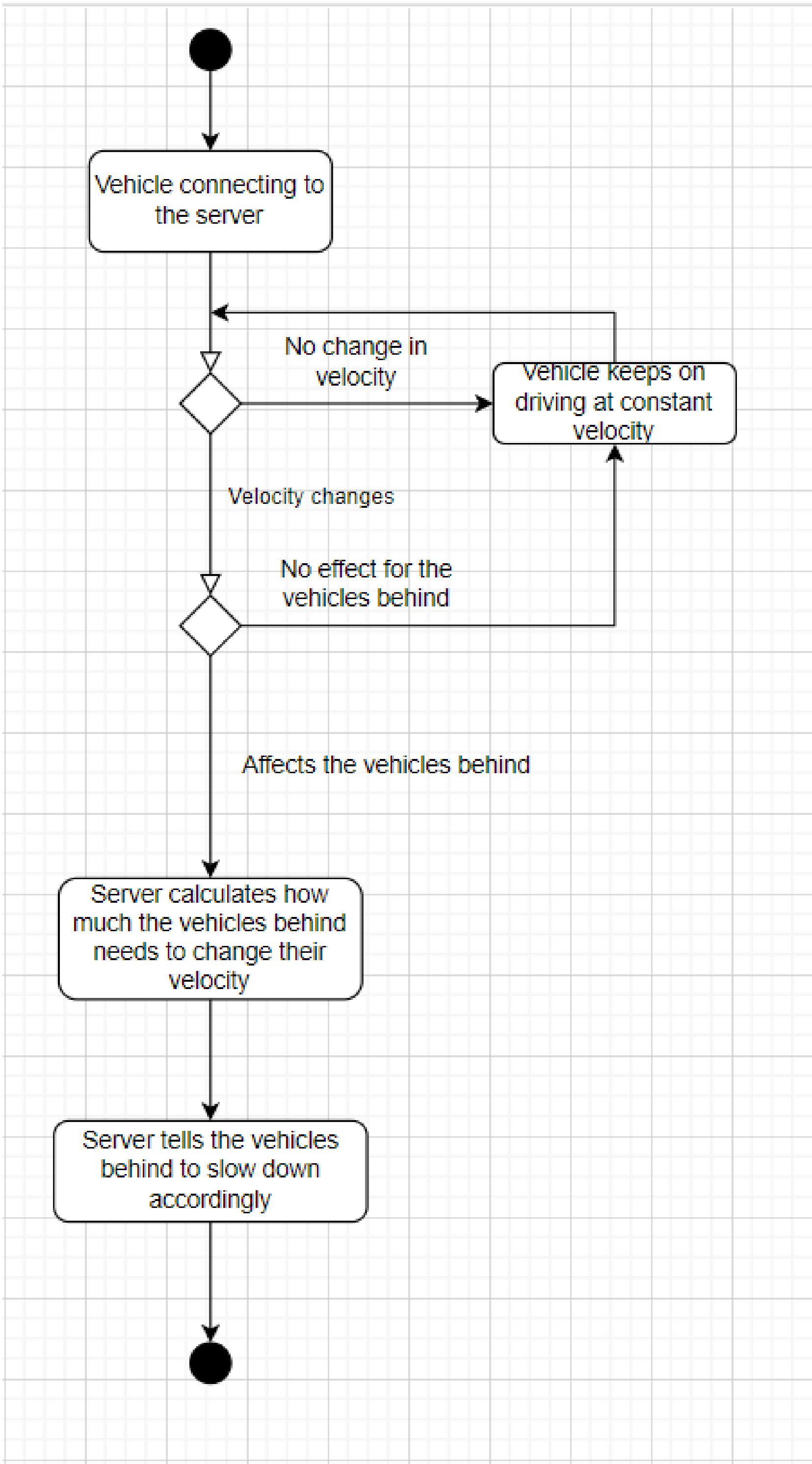

Figure 3.1: This figure shows the flow diagram of our first proposed solution.

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 and the standard HTTP methods; POST, GET, PATCH, DELETE, clients can send requests to a server to perform standard CRUD (create, read, update and delete) operations on a database respectively (IBM Cloud Education, 2021). 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 (Influxdata, n.d.).

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, REST API on the client, 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 Atlassian, 2019, 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, implementing webhooks includes extensive research into a system the group had never heard of, in addition to scarce information on how to create such a system. The group decided that the time constraint of this project did not justify the time it would take to implement such a system.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 (Fette & Melnikov, 2011).

3.3 Phase 3 - Websocket with SignalR

After extensive research 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 an excellent solution to our problem. Using WebSockets, both client and server can transfer data whenever they see fit. However, Microsoft, 2022b discourages 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 providing other transport methods such as long-polling as fallback (Microsoft, 2022a). 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, 2022a).

As a result of the failure to attain the result we wanted with REST API and InfluxDB, as discussed in 3.2 Phase 2 - REST API and why it did not work with our project, we decided to implement SignalR without InfluxDB on our server instead. First, set up an echo server using SignalR while simultaneously implementing the client code. The client code had to be implemented independently of the Raspberry Pi code. Because our goal was to create a communication module that is reusable through inheritance 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 representing the Raspberry Pi device became our next priority. By inheriting the client class, the vehicle class also inherits its 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.

3.3.1 Implementation of traffic on a single lane

After witnessing a successful connection between the Raspberry Pi vehicle and our SignalR server, we started to implement the necessary functionality on the server. We started with the simple scenario described in 3.1.3 Preventing traffic congestions for a one-lane road. The client will inform the server whenever its velocity has changed. Then, the server will relay this change to every other vehicle behind it on the same road and adjust their velocities accordingly. This server functionality also depends on the continuous monitoring of each 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 read and write on the database continuously. Hence, the group concluded that this would potentially impact latency on the server. Thus, we unanimously determined that a live-in-memory database storing vehicles in lists would be preferable. With simple mathematics, the server could recalculate the vehicle's position based on its previous velocity and a stopwatch whenever it retrieves the information about a vehicle instead.

We then determined to show the product owner from Accenture what we had been working on, thus inviting him and the external supervisor to the next meeting. This short demonstration consisted of running multiple clients with the server to simulate how multiple vehicles would behave with the intervention of our server. The simulation successfully demonstrated that vehicles were able to adjust their velocities in order to prevent the shockwave phenomena described in 3.1.3 Preventing traffic congestions for a one-lane road, and we received positive feedback. However, our supervisors challenged us to extend our concept and make our solution more complex. They believed implementing an intersection where multiple cars meet could show more advanced traffic management. They believed implementing an intersection where multiple cars meet could show more advanced traffic management.

3.3.2 Extending the concept to a more complex scenario - traffic on an intersection

Expanding further on the concept, new functionalities on the SignalR server were developed to handle vehicles approaching an intersection. With a more complex topology, expanding the database to account for the new road network is also required. Consequently, roads and intersections became required additions to our solution. Furthermore, the vehicle model on the serverside is now also composed of a route planner to represent what it means to be approaching an intersection.

With these improvements and new functionalities, the server can command the clients to adjust their velocity to avoid collisions between vehicles approaching an intersection simultaneously. By reducing the 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 answer the problem statement in 1.2.4 Problem statement. Following the work requirements, we still need to make a demonstration that showcases how the system works. Starting this work phase, the group and our internal supervisor strongly considered demonstrating our solution virtually. A virtual simulation could produce more data and consequently build a stronger case to answer our problem statement in 1.2.4 Problem statement. Contrary, supervisors at Accenture convinced us that a demonstration utilizing the existing Raspberry Pi vehicle was more in line with Accenture's goals for the project. Hence, the group started to build a new Raspberry Pi vehicle for this purpose.

3.4.1 Building a new car

The previous group had only built one car for their project. Consequently, we needed to build a new car to show a situation where two cars meet at an intersection. Luckily, Accenture kept a box of unused components from the previous group. However, we only had one TPU, the Coral Usb Accelerator. The TPU is an essential component for giving extra processing power to the computer, and it was necessary to run the artificial intelligence the previous group had used (Samset et al., 2020).

Without this accelerator, we could not run the artificial intelligence the previous group had incorporated into their solution due to the global chip shortage caused by the Covid pandemic; the accelerator was unavailable to purchase anywhere. As a result, the new car's camera and distance measuring sensor were absent. Not having two cars that utilized artificial intelligence could be a challenge. One of the required features of our solution was that the cars should be able to drive using the AI when a server connection was unavailable, as per described in 1.2.3 Goals and requirements. We decided that as long as we had one car that could navigate traffic independent of the server, the other car could drive solely on commands given by the server. With the components and the product documentation of the previous project 1.2.3 Goals and requirements, we were able to build a copy of the car as seen in Figure 3.2.

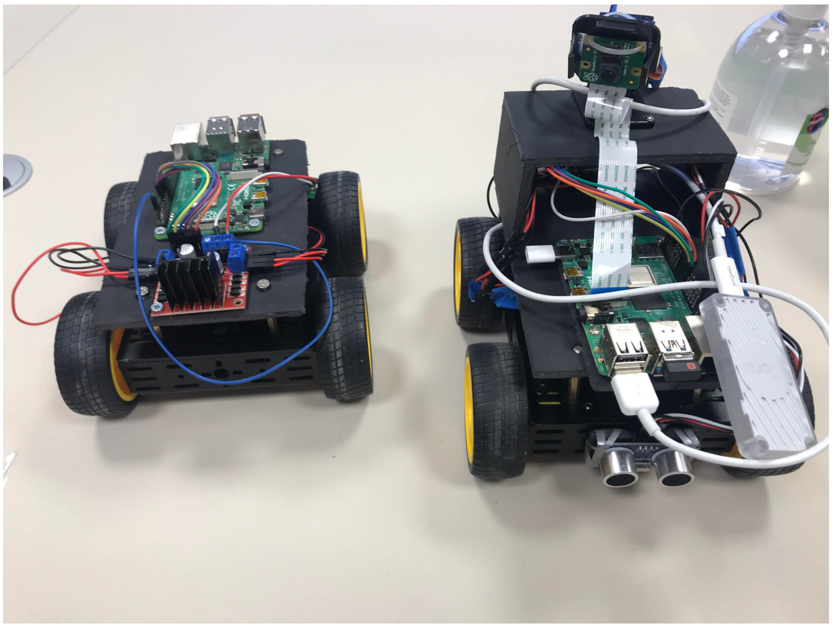

Figure 3.2: Photo of the two cars we built. To the left is the car without a camera. To the right is the car with a camera on top and a Coral USB Accelerator Edge TPU. This car can take advantage of the onboard AI.

3.4.2 Calibration of the cars

After building a second Raspberry Pi vehicle, we did some testing to figure out how the vehicle behaved together with the server when approaching an intersection. In this test, the vehicles were given a specific velocity and distance by the server. When the vehicles arrived at their destination, the server would tell them to stop and terminate their connection.

The vehicles were able to send information and respond correctly to the server's commands. We also observed that the vehicles drove a different length for each velocity given even though the length was the same. The unexpected behavior stems from the assumption that the outputting velocity is the same as the inputting power to the car's motors. The demonstration depends on the car's ability to input the correct amount of power to drive at specific speeds instructed by the server. Thus, the group started to measure the distance and time the cars would drive, given intervals of power to determine the relationship between power and velocity.

The measurement setup is as follows: The server instructs the cars to drive with a given power, which is what the server believed to be the velocity. After a time, the server will stop the car. The group then measured the distance the car had traveled. Given the measured distance and the time given by the server, the real velocity was obtained by dividing distance by time. Table 3.1 shows all the recorded measurements with this set up.

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: Measurements taken to determine the relationship between power and velocity. Units were provided to the different parameters, while the power unit was unknown.

We then made a graph in Excel by plotting our data with velocity on the y-axis and power on the x-axis as shown in Figure 3.3.

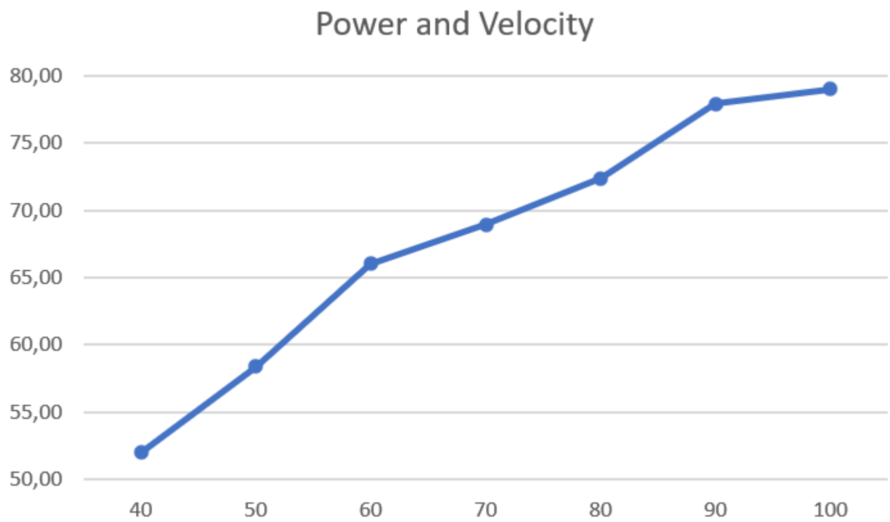

Figure 3.3: Graph of velocity as a function of power

From Figure 3.3, it appears that the power and velocity were linearly correlated. Hence, using linear regression, we obtained an approximate relationship between power and velocity. Figure 3.4 shows the result of the linear regression performed by Excel using the same dataset in Table 3.1.

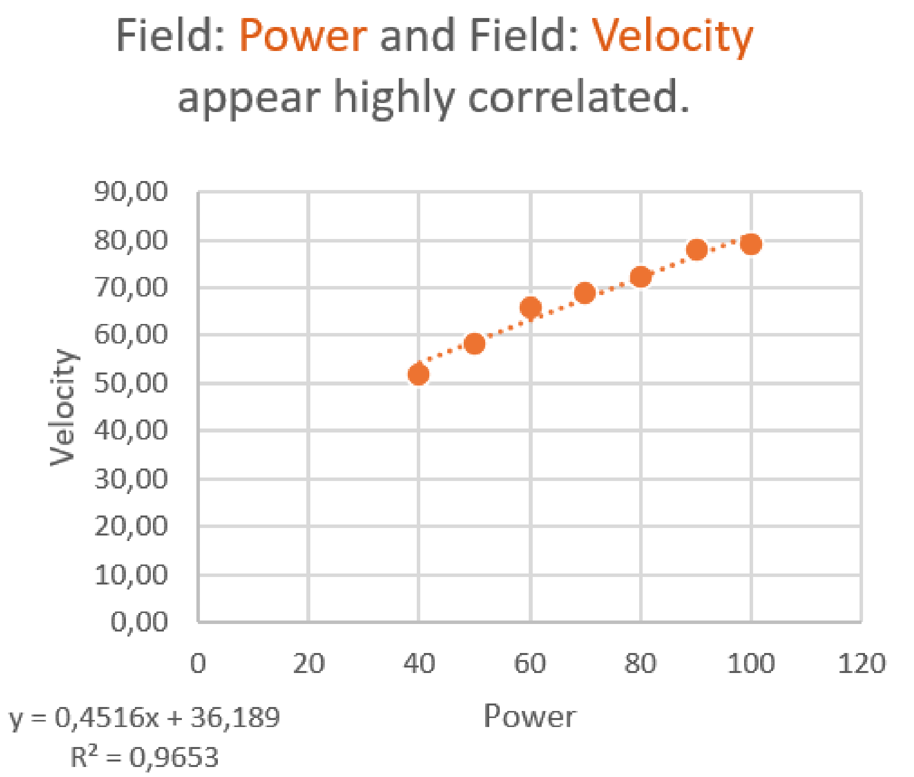

Figure 3.4: Graph of velocity as a function of power with linear regression in Excel

From Figure 3.4, the relationship between velocity and power is as follows: $v(P) = 0.4516 \cdot P + 36.189$, where P is power and v is velocity, with a goodness-of-fit measure of $R^2 = 0.9653$. We inserted the relationship into the vehicle and performed a new test to confirm the relationship. We observed that the vehicles drove more or less the correct distance for the desired velocity. Although the distance has some variance in the new tests, we concluded it was accurate enough for our demonstration. If, however, a more accurate formula is desired, then the test can be performed with smaller power intervals.

3.4.3 Construction of semi physical demonstration

We want to test that a centralized communication system and artificial intelligence can improve traffic flow. In order to test the solution we have worked on, we made a physical demonstration where two cars approach an intersection simultaneously. We wanted to observe if the velocity of the vehicles were not drastically changed and, therefore, decreased shockwave described in 3.1.3 Preventing traffic congestions for a one-lane road

We found a space at Accenture that was big enough to build the track. The power input on the Raspberry Pi vehicles is limited to between 40 and 100. Since the server will adjust the vehicle's speed to avoid a collision, the intersection was required to be at least 130cm away from the starting point to prevent the server from adjusting speed outside the equivalent power limit. Furthermore, the server prevents cars from driving before at least two vehicles have established a connection. Hence, both vehicles will start their journey simultaneously. We also placed the vehicles at the same distance from the intersection on their respective roads. Given these initial conditions, both vehicles are supposed to collide without the server's intervention. We also made the server log the velocity sent to the cars to track how much the cars' velocities changed.

Making the demo one hundred percent accurate was not possible in our circumstances due to the limitations of the Raspberry Pi. Furthermore, the vehicles were not able to receive the messages simultaneously. Consequently, this meant that they could start with a minor time difference. Another factor was that the trajectory of the vehicles was not always straight. However, we were able to get a consistent demo with enough margins. Figure 3.5 shows an example of a collision during a test demonstration.

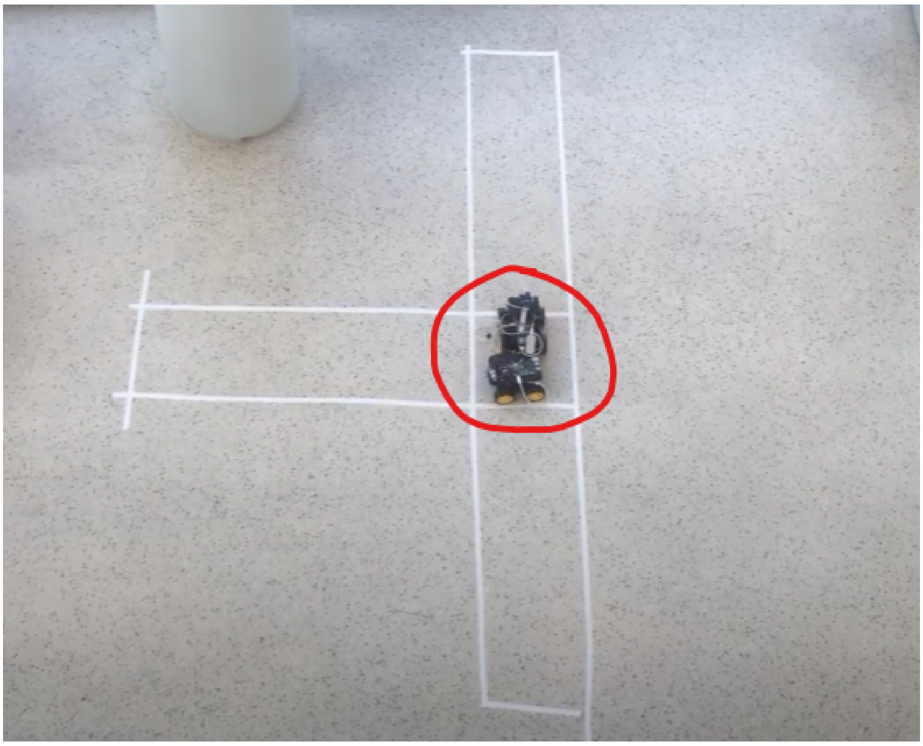

Figure 3.5: Here, we can observe the two cars colliding in the intersection. One of the cars started about 5 centimeters further behind the other car and started a few milliseconds later, resulting in a collision. Meanwhile, the server assumed they started simultaneously at the same distance from the intersection.

Furthermore, the server did not account for the lengths of the vehicles during its calculations. After we introduced the length of the vehicles and buffer zone, we were able to get a consistent semi-physical demonstration.

When both vehicles had connected to the server, the cars would drive with an initial speed of 80 cm/s, the upper limit of the Raspberry Pi. Not long after they started to drive, the server recognized the cars approaching the intersection. The server calculates using the car's velocity, position, and length. Then the server calculates which car has to slow down and how much the car needs to slow down to avoid a collision, in this case, to 55 cm/s. After the other car has supposedly passed the intersection, the car that slowed down gets told by the server to speed its velocity back to 80 cm/s. Figure 3.6 is a snapshot of a successful test demo.

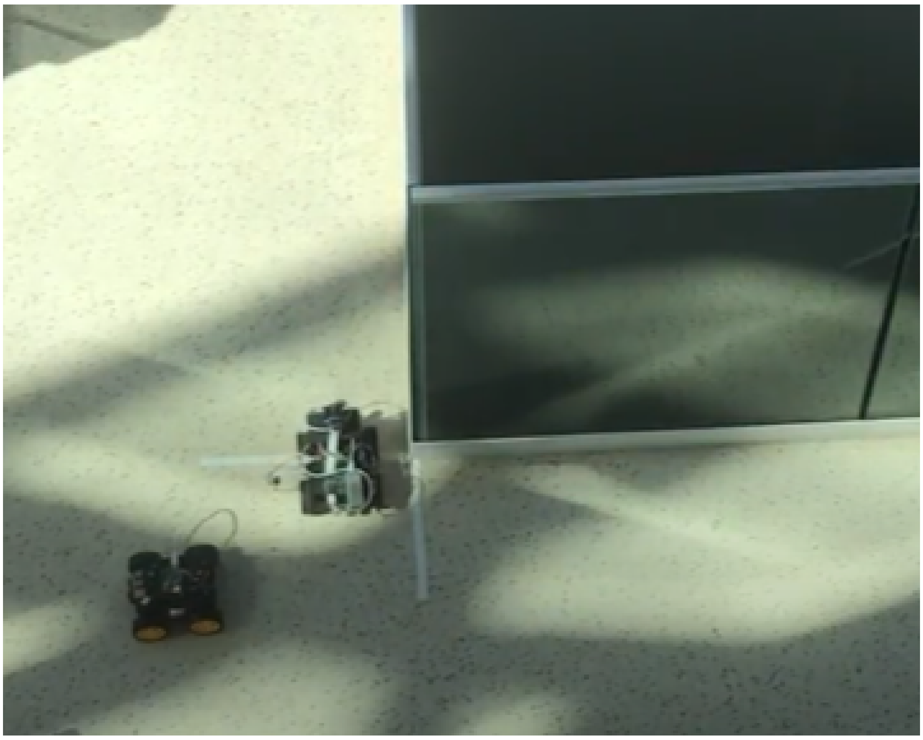

Figure 3.6: Here is a snipped from a successful demo. The car to the left has just passed the intersection, marked as a square with white tape. The car furthest up is, therefore, about to adjust back to its original velocity. Here we can observe that the server prevents a collision.

Chapter 4

Conclusion

4.1 Results

In regards to the problem statement in 1.2.4 Problem statement and the goals for this project discussed in 1.2.3 Goals and requirements, we have been able to successfully create a centralized system that communicates that can communicate with autonomous vehicles. A significant change in velocity can lead to a disruption of traffic flow, as discussed in 3.1.3 Preventing traffic congestions for a one-lane road. Through multiple semi-physical demonstrations discussed in 3.4.3 Construction of semi physical demonstration, it was apparent that the server was able to contribute to an improved traffic flow by preemptively reducing vehicles' to the necessary speed to prevent collisions. Even though this project did only account for scenarios involving intersections, it is also from the exploration done throughout this project that autonomous vehicles, together with a centralized system, can be the answer for improving traffic flow in the future.

Furthermore, we received positive feedback after showing our supervisors and product-owner video snippets of the conducted demonstrations. Additionally, the product owner confirmed that our results adhered to the requirements set by Accenture.

4.1.1 Real world scenario

By comparing the semi-physical demonstration to a real-world scenario where instead of 80 cm/s, the cars would drive 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. Hence, it will involve some cars moving while other stops. In our scenario, the car would only have to slow down to 55 km/h.

Furthermore, our project does not consider other factors, such as curved roads, human mistakes, and animals jumping onto the road. Hence, our demonstrations, conducted in a controlled environment, are not a good reflection of a real-world scenario.

4.1.2 Possible improvements

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 it fulfilled the product requirements given by Accenture, there is room for improvement. In addition, there is room for improvement with the accuracy of the demo, specifically by doing more calibration tests and making a more accurate formula than we were able to produce. Furthermore, changing the wheels and installing an edge TPU to exploit the existing AI model will result in a more predictable trajectory of the vehicles.

Our demo only contains two cars, but our server supports scenarios with multiple cars. 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. However, a virtual simulation will yield more data for research while also exploring other more complex road configurations.

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.

4.2 Further discussions

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

4.2.1 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.

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. Because of the time constraint, we also found it challenging to focus on scalability. We focused on getting a working demonstration rather than making it scaleable.

One challenge 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 2.3), 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.

4.2.2 Educational Value

Developing this solution has been challenging, and our group has learned a lot. We all feel that we have evolved into better developers and that we now would be better suited for solving similar projects in the future.

The project description we started with was very open to interpretation, which gave us much room for exploration. We chose to go with an IoT solution using the car Accenture already owned. Ultimately, we also decided to build a new one. This choice has given us significant insight into making IoT systems and ways to set up communication between them. We believe this resulted in a much more exciting demonstration for Accenture to showcase.

Furthermore, combining such a system with artificial intelligence has been an exciting learning possibility. Due to this project's time constraints, we, for instance, did not integrate machine learning with our server. Integrating machine learning on the server could expand its different capabilities and make the project more scalable and applicable to a broader range of scenarios. Such integration is deemed valuable for the further extension of this project.

While we still were in the first development phase 3.1 Phase 1 - Planning and research phase, we decided to adopt the code of the previous group instead of developing our own AI model. Consequently, we believed that this was the right choice, as we believe it would have consumed too much time to start from scratch. Although it was a challenge to understand the program the previous group had written, we assumed that this way, we would get to focus more on our demonstration. The importance of documentation has also been emphasized, which helped our group to write our report.

Our group had little to no experience working with an agile work methodology. In retrospect, 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 workflow in an agile manner.

4.2.3 Real world application

The best way to implement self-driving cars in society is a topic that will take a long time to explore fully. 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 create a system that will be viable for all scenarios. Such a system might still seem out of reach. However, individual contributions and new additions to this discussion might take us one step closer to a complete system.

4.2.4 Edge computing and AI

When self-driving vehicles become more prominent and the 5G network becomes more available, there could be a possibility for IoT systems to handle traffic management. Therefore, our group researched 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 to prevent overloading on servers handling vast loads of data (Zivkovic et al., 2020, pp 149).

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

In 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 and the cars. The AI will probably need to train in a safe test environment just like the autonomous vehicles before they are ready for existing road systems.

In our IoT program, the server gives commands to the cars, which overrides the cars' AI. In contrast, ideally, an interaction between the AI on the server and the cars is required for real-world scenarios.

4.2.5 System security

Security and privacy are important topics for any IoT system. Because these systems gather and work with vast amounts of data, they are naturally prone to be attacked. Moreover, as the systems grow and become more interconnected, with many devices worldwide, the imposed risk of such an attack increases drastically (Nurse et al., 2017). Anonymization of personal data, securing connections, and an intruder detection system are all security measures that must be considered. The cars must be able to drive both with or without the system, which is an essential feature of system security. If there is a need for the server to shut down, the cars would need to be able to drive independently of the system.

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:

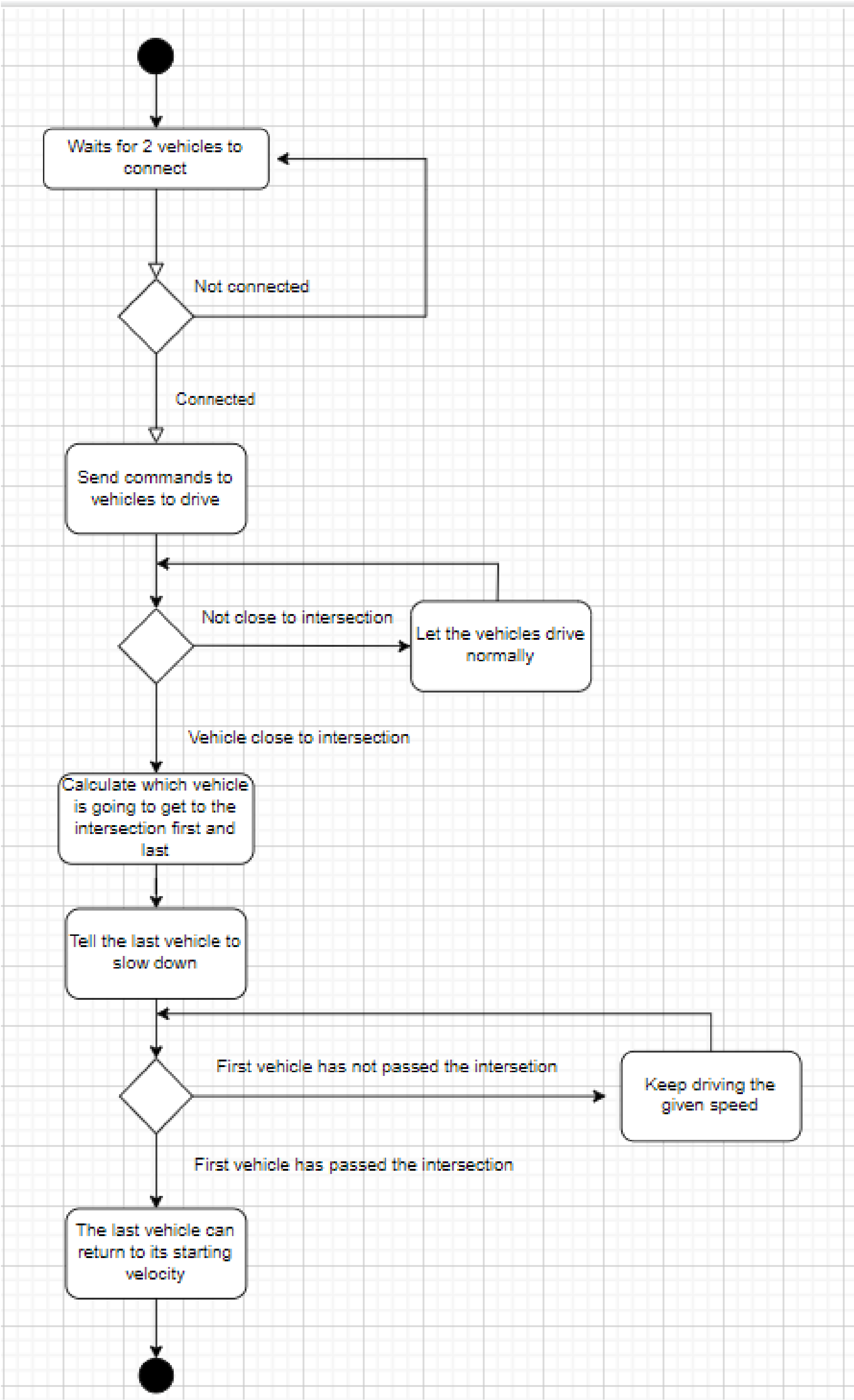

Figure 5.1: Flow diagram for the server. This diagram specificly shows the flow of the semi physical demonstration.

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.

```python
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:

```csharp
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;
  ...
}
```

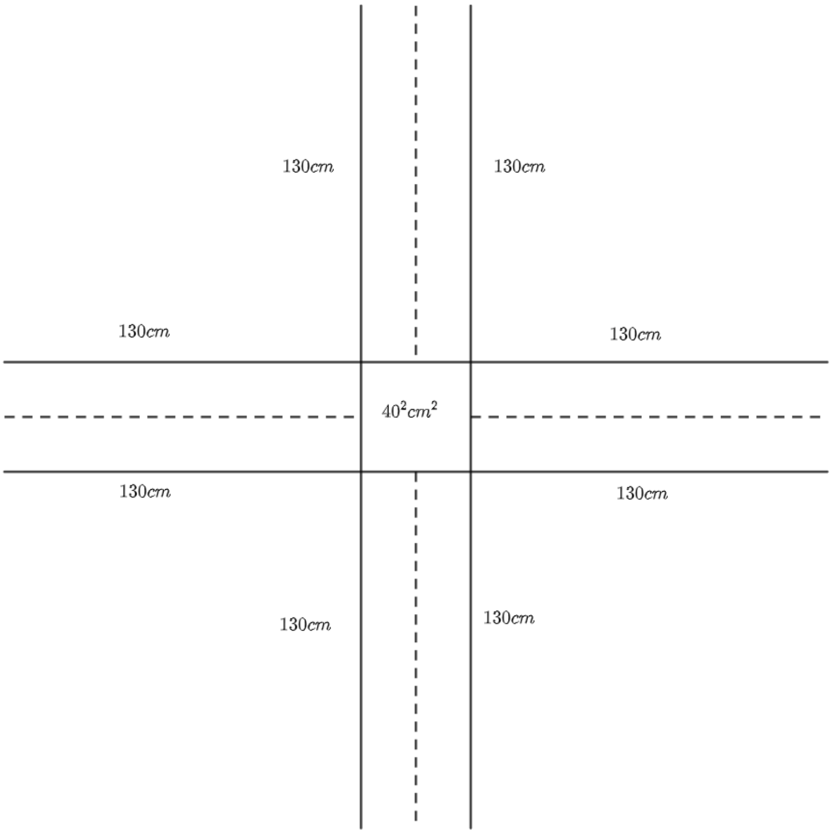

Figure 5.2: This figure shows two roads each with two lanes and a total length of 300cm. The overlapping part forms a square which is the intersection. This configuration was heavily considered when creating representational models on the server, and also used for the demo as a result.

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 (IBM Cloud Education, 2020).

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 APEC Delegation, 2014).

Bibliography

Accenture. (2021). *Earnings report - 2021*. Retrieved March 21, 2022, from https://newsroom.accenture.com/news/accenture-reports-very-strong-fourth-quarter-and-full-year-fiscal-2021-results.htm

Accenture. (2022). *About accenture*. Retrieved March 18, 2022, from https://www.accenture.com/no-en/about/company-index

Atlassian. (2019, August 13). *Webhooks*. Retrieved May 16, 2022, from https://developer.atlassian.com/server/jira/platform/webhooks/

Chinese APEC Delegation. (2014). *White paper of internet of vehicles (iov)*. Retrieved March 18, 2022, from http://mddb.apec.org/Documents/2014/TEL/TEL50-PLEN/14_tel50_plen_020.pdf

Fette, I., & Melnikov, A. (2011). The websocket protocol. https://doi.org/10.17487/RFC6455

IBM Cloud Education. (2020, June 3). *Artificial intelligence (ai)*. Retrieved March 18, 2022, from http://ibm.com/cloud/learn/what-is-artificial-intelligence

IBM Cloud Education. (2021, April 6). *Rest apis*. Retrieved May 16, 2022, from https://www.ibm.com/cloud/learn/rest-apis

Influxdata. (n.d.). *Influxdb is the time series platform for developers*. Retrieved May 16, 2022, from https://www.influxdata.com/products/influxdb-overview/

Levinson et al. (2021). *Shockwaves*. Retrieved May 21, 2022, from https://eng.libretexts.org/Bookshelves/Civil_Engineering/Fundamentals_of_Transportation/06%3A_Traffice_Control/6.01%3A_Shockwaves

Long, J., Gao, Z., Ren, H., & Lian, A. (2008). Urban traffic congestion propagation and bottleneck identification. *Science in China Series F: Information Sciences*, *51*(7), 948. https://doi.org/10.1007/s11432-008-0038-9

Microsoft. (2022a, March 26). *Overview of asp.net core signalr*. Retrieved May 16, 2022, from https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-6.0

Microsoft. (2022b, March 26). *Websockets support in asp.net core*. Retrieved May 16, 2022, from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-6.0

Naranjo, J. E., Yang, X., Luo, S., Gao, K., Qiao, T., & Chen, X. (2019). Application of data science technologies in intelligent prediction of traffic congestion. *Journal of Advanced Transportation*, *2019*(7), 2915369. https://doi.org/10.1155/2019/2915369

Nurse, J. R., Creese, S., & De Roure, D. (2017). Security risk assessment in internet of things systems. *IT Professional*, *19*(5), 20–26. https://doi.org/10.1109/MITP.2017.3680959

Samset, T., Bakken, M., Sandberg, H., & Nguyen, E. (2020, May 19). *Self driving vehicle: A prototype using raspberry pi* (tech. rep.). Høyskolen i Kristiania.

Sinha, S. (2021, September 22). *State of iot 2021: Number of connected iot devices growing 9% to 12.3 billion globally, cellular iot now surpassing 2 billion*. Retrieved March 18, 2022, from https://iot-analytics.com/number-connected-iot-devices/

Vaagaasar, A. L., & Skyttermoen, T. (2021). *Prosjektveilederen: For deg som vil skape verdi* (1st ed.). Cappelen Damm Akademisk.

Zivkovic, C., Guan, Y., & Grimm, C. (2020). *Iot platforms, use cases, privacy, and business models: With hands-on examples based on the vicinity platform* (1st ed.). Springer, Cham.

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?

Sprint planning for sprint 4 (21.03 - 04.06)

- Oppgaver fra backlog
    - **Disposisjon til bacheloroppgave**
    - **Teori til bacheloroppgave**
    - **Få server til å sende kommando til flere biler.**

    Mulige oppgaver:
    - **Enhetstesting**
    - **Simulering**

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.