DATA3900 - Bachelorproject
Final report

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

Total pages: 30

Last updated:
May 18, 2022

Abstract

Short 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. Our project was done for Accenture, and lasted from January 2022 to May 2022. We have tried to develop a solution for traffic management with self-driving cars and server communication. A physical demonstration of our proposed solution is done with the use of a Raspberry Pi computer working as a car. In this project we have documented the functionality of our system, and our process of making it.

The report is split into 7 chapters. Introduction, research areas, process documentation, implementation, results, discussion and conclusion. At the end we have added a Moscow analysis, sprint overview and our project journal as appendices. Technical terms are in 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

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, Application Development Analyst at Accenture AS, External Supervisor

1.1.4 Client

Accenture AS

Accenture is an international IT consulting firm that operates in 200 different countries across the world. The main office is located 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

In 2020, Accenture supervised a bachelor student-group with the goal of developing a self-driving car. The group developed a model-sized car that uses artificial intelligence to analyze data from sensors and cameras, and makes decisions based on the data. Accenture wants to build further on this project, by exploring the addition of a centralized communication system for the vehicles.

The product owner of the project states various reasons that make it relevant for Accenture. According to Accenture, Norway is one of the countries that is most ready to start utilizing self-driving cars, although not all factors that need to be in place are ready. Self-driving cars alone will also likely not be able to solve all the problems related to today's traffic-problems. Accenture wants to explore if combining self-driving cars with an autonomous management system could provide greater value.

==================

Accenture has in many years offered final year students at OsloMet and Høyskolen i Kristiania interesting projects for their bachelor thesis. In 2020, a group of students from Høyskolen i Kristiania were 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 centrilized communication system.

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

1.2.2 Significance

Accenture's values coincide with the project, seeing as they wish to build their knowledge in the form of 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 goal of the project is to end up with a prototype that can be shown to interesents and demonstrate how self-driving cars could be combined with a centralized system. For this we will use the self-driving car that Accenture already owns, as a result of the previous bachelor's-thesis done with them. The conditions are that the product must be possible to present, either as a physical or digital showcase. 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 cars. 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 thesis question:

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

Process documentation

2.1 Development method

The task given to us was open because we had few restrictions, and it was up to our group to decide which technologies to use. This meant that work methodology had to focus more on flexibility rather than planning. We therefore chose to use an agile work method. Agile work method focuses 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 work was a good fit for our project.

We chose to use inspiration from two light frameworks, kanban and scrum. Scrum is an agile, light framework which helps people and teams work together. Scrum describes a set of meetings, roles and tools. Under is a figure describing a scrum work process:

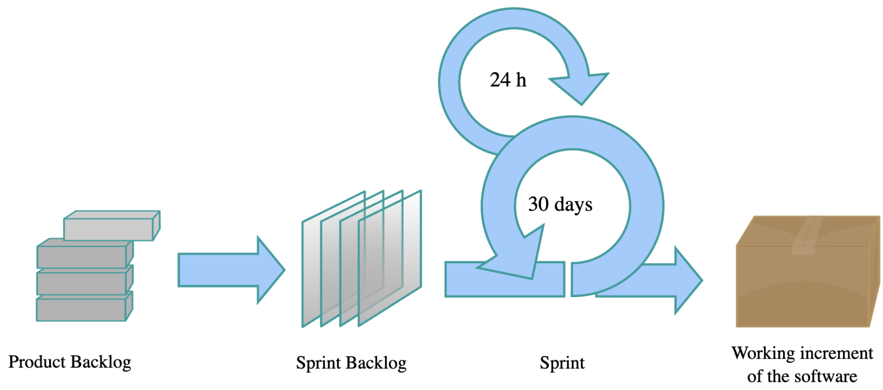

Figure 2.1: Illustration of a generic scrum process. Here the sprints are in 30 day increments

A sprint is an essential part in 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 a sprint planning and ends with a sprint retro perspective. For our project we found it most viable to plan in increments of two weeks. We chose two weeks increments because we felt it was an even balance between work and planning. As mentioned we also had meetings with our client Accenture every 2 weeks which fitted well with the time increment.

Our group also made use of the meetings in the Scrum framework. The meetings are sprint planning, sprint retroperspective and daily standups. Sprint planning is a meeting or event which starts before a sprint. At the sprint planning the teams agree on a goal for the sprint, and the tasks from the backlog that should be worked on that contribute to the goal. Example of report from a sprint planning meeting is attached. The backlog is a list of functionality the product should contain. In addition we wrote down the tasks for the specific sprints in a digital document. These tasks were to be finished by the next sprint. Here is our sprint planning documents that shows our goals for each sprint:

(Legge til figur av sprint planning documentet vårt)

After a sprint we would have a sprint retro perspective where we would discuss what went well in that sprint, and what could have been done better. This helped us reflect over the prior week and adjust accordingly, if necessary. We would also discuss if the task asigned in the sprint meetings were finnished or needed more work. This helped us figure out if we were on track with our initial plan.

In addition to the weekly meetings we also had daily standups. Daily standups is a short meeting, usually around five minutes, where each person answers three questions. What did you do last time? What are you doing 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 where we did not feel the necessity to have specified roles, because we usually worked together on our project. 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.

Our implementation of Kanban was to use a Kanban board as the backlog. A Kanban board is used to visualize where a task is in the work process. Here is an example from our project:

We have four columns that represent which phase a task is in. The backlog is the tasks in the to-do column. When we are working on a task we drag it over to the In progress column. After a task is finished it goes to the "Review in progress", where we review the task. If we concluded that the task wad finnished in the reviewing it gets dragged over to the "Done column". The git project tool also provides prioritation of tasks from high to low. We also took advantage of this. You can also connect the tasks to a specific branch so the task automaticly gets finnished when merging the branch into master. 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 Canban board thoughout the whole prosess. This is

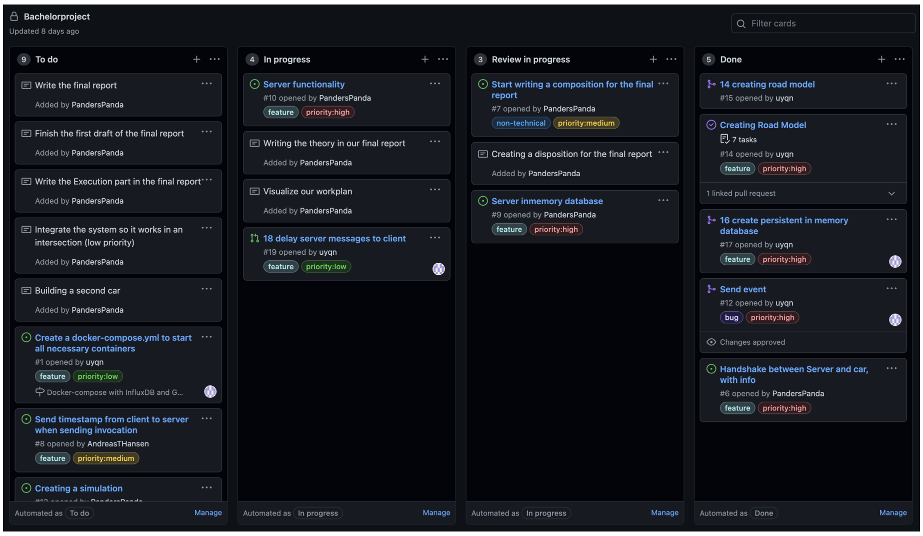

Figure 2.2: Extract of our kanban from Github.

because the backlog was changing a lot, and therefore the canban board needed many modifications to be up to date. In addition, we were able to keep track of the tasks by having frequent meetings.We figured out that it was more beneficial to focus on one framwork, which in our case was scrum.

2.2 Tools and technologies

The circumstances surrounding Covid 19 ment that we were not able to meet our supervisors in person at the start, however many of the restrictions cleared up at the later part of the project. Luckily the group was still able to meet physically a few times a week. We still had to use a wide range of tools for communication.

- 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
- 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 to work on the project together were important. Here is 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.

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

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

Project planning and documentation was also an important part of the project. Here is a few tools we used for the project planning:

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

Implementation

3.1 Process phases

We chose to split our process into four phases:

Planning and research phase	week (x-x)
API-solution with time series database	week(x-x)
Signal-R	week (x-x)
Demonstration of our system	week(x-x)

3.2 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 direction of the project.

The first choice to make was to either build an AI for the vehicles from scratch or use the AI from the group prior. Building the AI from scratch ment that we could make an AI which integrated our system from the start, however with the time constraint of the project we decided that it would take less time to use the product from the group prior. In addition, our group had more prior experience with networking than with raspberry Pi and AI. We therefore chose to use the AI from the project prior. We also had to decide which technologies and programming languages to use.

In this phase we did not have access to the vehicle made by the prior bachelor group and neither the code. In addition there was a need for project planning and research before we could start developing our IoT-system. The topics needed to be researched was:

- Research what causes traffic jams and solutions to fix it.
- Research IoT-systems and how they function with vehicles.
- Research different planning methods that will fit our project

We also used the pre-project phase to get to know Accenture, their guidelines and their workspace since we were going to work a their office.

3.2.1 Choice of programming languages

For the programming languages, we chose to use python for the client and C# for the server. The group prior had used python for their raspberry Pi vehicle. This meant that using python made it easier to extend the code from the project prior. Our group also had experience with networking in python, therefore making python a prime choice of programming language for the client. C# supports multiple ASP.NET Core libraries which was useful since we were going to work with networking. The server needs to be as efficient as possible and C# is considered a fast programming language. We also had some prior knowledge coding in C# as well.

3.2.2 Internet of Things

The Internet Of Things refers to physical objects that communicate with the use of sensors, cameras, software or other technologies that connect and exchange 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**). The field of IoT has also been evolving in recent years due to other technologies becoming more accessible, such as machine learning and the 5G network.

You need not look further than to the smart-home consumer market to find applications in your life, of devices communicating to solve problems. It could also be applied in climate surveillance systems, energy or transportation. The benefits that an internet of connected devices could add billions in value to industries across the world, and to the global economy. 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, and it is a central theme of our thesis. 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, with the hopes of making a solution that can be scaled up at a later point.

3.2.3 Preventing traffic congestions for a one lane road

A traffic congestions also known as traffic jams are when a long line of vehicles are moving slowly or have stopped moving completely. Traffic jams are annoying and distrupts nearby local environments with sound and gas emissions. There are many factors which can cause a traffic congestion such as: poorly designed roads, not wide enough roads, traffic light patterns, and accidents. In conclusion, an event that distrupts the traffic flow can cause a traffic congestion.

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

This scenario will lead to the vehicles behind needing to slow down as well, distrupting the traffic flow. To prevent this, the vehicles behind has to deacrease their velocity before they reach the destination of where the event happened.

The reasoning for this is because then the vehicles does not need to decrease their velocity as drasticly.

After a few dicussions in our group we came up with an idea of how the interactions between the server and the cars would be. First the car has to connect to the server and give information about its current speed, weight, width and lenght. The server would keep track on all the cars positions at the road. Since there is only a single road there was only one dimention to worry about. The cars would send information to the server if their velocity had changed. This would trigger an event on the server where it would tell all the cars behind the car that triggered the event to change their velocity to slow down accordingly. The velocity the cars would have to change to deppended on their distance to the car slowing down and their current velocity. Here is a flow chart of the demo for this solution:

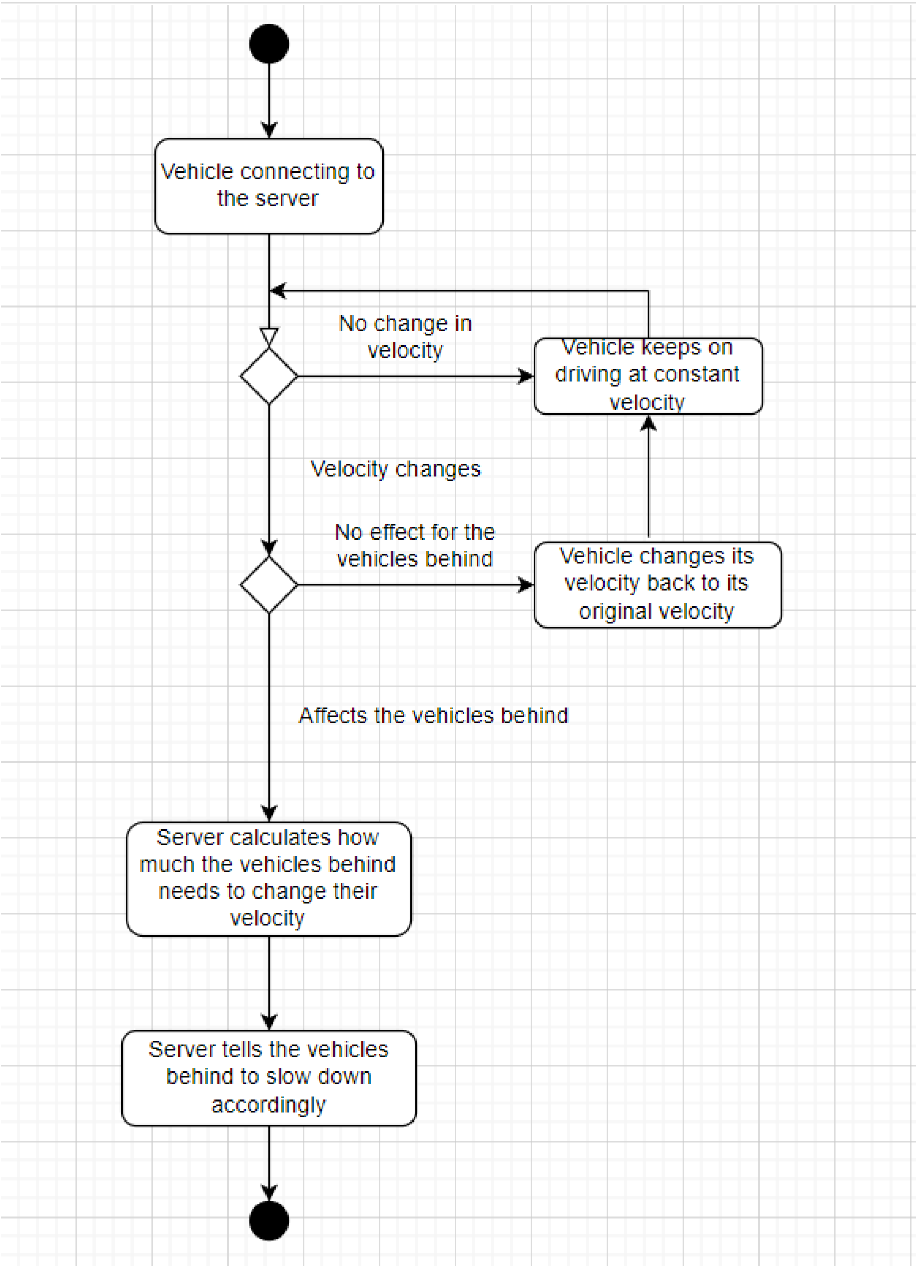

Figure 3.1: Flow diagram for the demo for the first solution.

3.2.4 Prioritization method

MoSCoW method is a prioritization technique used in project managment to prioritize requirements. The word is split into four sections: The "Mo" stands for must have and, is the number one priority requirements for our project. Theese are necesarry for the success of our project. "S" stands for should have. Theese requirements should be incuded in our project, however not mandetory. The "Co" stands for could have. The requirements in this section are requirements that are extras we want to include in our project, however not prioritized. At last the "W" stands for will not have this time. The requirements in the "W" section is maybe for a later group if someone wants to further build on our project.

Since we were unsure how many use cases we finish within the time contraint of the project period, we thought a prioritazion method was a good fit. Under is the visualization of the MoSCoW method.

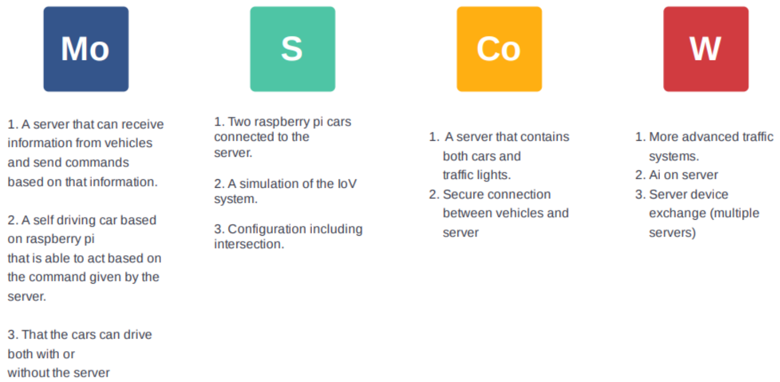

Figure 3.2: Visualuzation MoSCoW method. Everything under "Mo" are requirements we must have in our project, under the "S" sections are requirements we should have, and under "Co" are requirements we should have, however not necessary. Under "W" are requirements we will not have this time

3.3 Phase 2: REST API

The aim of the project was to connect Raspberry Pi devices to a system. Thus, the group decided that the best way to establish this connection was to implement a RESTful API on the server. In addition, the question on how the data should be stored was also arises during this phase.

Representational state transfer (REST) *application programming interface* (API) provides a way for client and server to establish communication through *hypertext transfer protocol* (HTTP). 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**).

Due to the time sensitivity of the application we are trying to build, it was therefore necessary to choose an appropriate type of database for our system. In this case, we chose to incorporate the time series database. InfluxDB is a time series database created by InfluxData. It provides a SQL-like syntax for querying resources that is quick and scalable, and most importantly free. Moreover, InfluxDB client library, using the influxDB v2 API, provides both ease of install and use for a multitude of langauages (**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 its velocity, acceleration and position to the server. Client should first POST itself to the server. The server will then add the vehicle to the influxDB to keep track of the vehicle's information. Then, the vehicle should be able to perform a GET request to the server to retrieve its information.

The client at this phase would be able to send a PATCH request to the server in order to update its information. In addition, the server will add a new entry into the database whenever it receives this request. The idea was that the client and server would be able to continuously communicate to each other such that the server could determine the behaviour of all connected clients.

However, a RESTful API server were not able to perform all its required task that the group wanted the server to do. Firstly, the server were only able to communicate with one client at the time, i.e. the client that sends a request. What the group wanted at this stage was that based on a request the server should also be able to send its own request to all the other clients. In order to acheive this, the server had to send an unprompted response to other clients that is not requesting a resource from the server, which was not possible with our current architecture.

A new solution had to be in place in order to achieve our goal. Other solutions were proposed during this stage.

Long polling

Polling is the idea that the server pushes resources to the client. There are mainly two types of polling; short- and long polling.

In short polling, a client requests a resource from the server and the server responds with nothing if the resrouce is not available. The client will then send a new request in a short amount of time and the cycles repeats until the client receives the resource it has requested.

Long polling is similar to short polling, however, the server does not send anything back before the resource is available. That is, the client sends a request to the server and the server is holding 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 on a seperate thread and instruct the server to hold onto this request until it had further instructions to the requesting client.

However, implementing a method on the server to block the response introduced more complication to the project. Also, with the asynchronous nature of the controllers implemented on the server it would also mean that the server will consume a lot of the processor which also means that the performance of the server will be heavily deteriorated.

Implement REST API on the client

Another solution was to implement the client itself as a REST API server on its own. However, in order to achieve this, each client needed to also send the server its host and port information to the server. Also, the server has to be implemented as a client in order to connect to the vehicles.

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, 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.

3.4 Phase 3: SignalR

After exhaustive discussions on how to solve the two-way communication discussed in Phase 2: REST API, the group agreed that websockets would be a good solution to our problem. Websockets is a protocol that provides a bidirectional communication between clients and server by establishing a single TCP connection in both direction (**rfc_websockets**). Hence, 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.

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.

3.5 Phase 4 - Making a demo

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.

3.5.1 Building a 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.

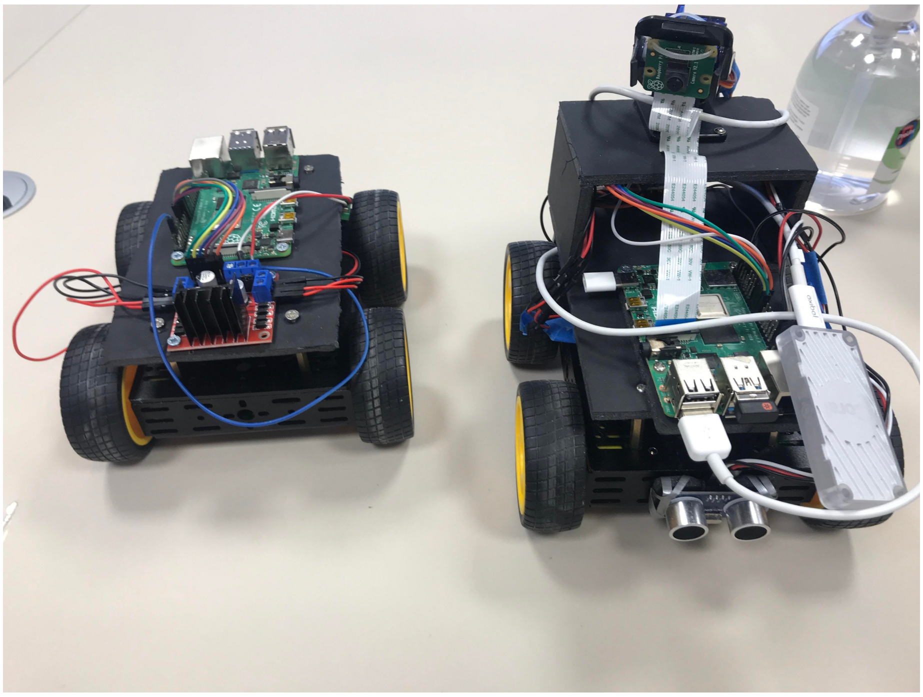

Figure 3.3: Cars with and without camera, right to left

3.5.2 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 direc-

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

tions 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 3.4: 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:

Figure 3.5: Graph of velocity as a function of power with linear regression

The formula we ended up with was as follows: $P = 0.4516v + 36.189$, 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.

3.5.3 Construction of 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 the limit. The server works in a way that the demo would not start until both vehicles had connected to it. Theoreticly 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.

(Picture of the demo where the vehicles crashes)

To make the demo hundred percent accurate were not possible. 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 we assumed in the server. However we were able to get a consistent demo with enough margins. We made vast margins by making a buffersone around the vehicles. The buffersone was about twenty centimeters.

(Picture of a succesful demo! Yay)

3.6 Phase 2: REST API

The aim of the project was to connect Raspberry Pi devices to a system. Thus, the group decided that the best way to establish this connection was to implement a RESTful API on the server. In addition, the question on how the data should be stored was also arises during this phase.

Representational state transfer (REST) *application programming interface* (API) provides a way for client and server to establish communication through *hypertext transfer protocol* (HTTP). 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**).

Due to the time sensitivity of the application we are trying to build, it was therefore necessary to choose an appropriate type of database for our system. In this case, we chose to incorporate the time series database. InfluxDB is a time series database created by InfluxData. It provides a SQL-like syntax for querying resources that is quick and scalable, and most importantly free. Moreover, InfluxDB client library, using the influxDB v2 API, provides both ease of install and use for a multitude of langauages (**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 its velocity, acceleration and position to the server. Client should first POST itself to the server. The server will then add the vehicle to the influxDB to keep track of the vehicle's information. Then, the vehicle should be able to perform a GET request to the server to retrieve its information.

The client at this phase would be able to send a PATCH request to the server in order to update its information. In addition, the server will add a new entry into the database whenever it receives this request. The idea was that the client and server would be able to continuously communicate to each other such that the server could determine the behaviour of all connected clients.

However, a RESTful API server were not able to perform all its required task that the group wanted the server to do. Firstly, the server were only able to communicate with one client at the time, i.e. the client that sends a request. What the group wanted at this stage was that based on a request the server should also be able to send its own request to all the other clients. In order to acheive this, the server had to send an unprompted response to other clients that is not requesting a resource from the server, which was not possible with our current architecture.

A new solution had to be in place in order to achieve our goal. Other solutions were proposed during this stage.

Long polling

Polling is the idea that the server pushes resources to the client. There are mainly two types of polling; short- and long polling.

In short polling, a client requests a resource from the server and the server responds with nothing if the resrouce is not available. The client will then send a new request in a short amount of time and the cycles repeats until the client receives the resource it has requested.

Long polling is similar to short polling, however, the server does not send anything back before the resource is available. That is, the client sends a request to the server and the server is holding 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 on a seperate thread and instruct the server to hold onto this request until it had further instructions to the requesting client.

However, implementing a method on the server to block the response introduced more complication to the project. Also, with the asynchronous nature of the controllers implemented on the server it would also mean that the server will consume a lot of the processor which also means that the performance of the server will be heavily deteriorated.

Implement REST API on the client

Another solution was to implement the client itself as a REST API server on its own. However, in order to achieve this, each client needed to also send the server its host and port information to the server. Also, the server has to be implemented as a client in order to connect to the vehicles.

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, 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.

3.7 Phase 3: SignalR

After exhaustive discussions on how to solve the two-way communication discussed in Phase 2: REST API, the group agreed that websockets would be a good solution to our problem. Websockets is a protocol that provides a bidirectional communication between clients and server by establishing a single TCP connection in both direction (**rfc_websockets**). Hence, 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.

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.

3.8 Results

We observed troughout multible demonstrations that the vehicles accelerated from 80 cm/s to 55 cm/s when the server intervnted. Without the servers intervention, one of the vehicles would have had to completely stop at the intersection which would be a higher change in velocity. We concluded with that in this specific scenario, the server had increased traffic flow. However it is hard to say anyhting regarding more complex scenarios with more vehicles.

3.8.1 Self evaluation

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

We were able to satisfy most of the product goals and requirements in the end, however some extra hours were needed towards the end. Because of the time contsraint we found it difficult to focus on scalibility. We chose to mainly focus on getting a working demonstration of our IoT-system. In terms of scalibility, we can add more vehicles, but the roadmodel does not fully support functionalites for having more roads than we currently have in our demonstration.

One challenge in the process was that our group were not able to meet as a whole as much as we had wished because of work. If we had worked more thoghout 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 finnish all the requirements in "must have" and "should have" section, but none of the "can have" section. All in all the group was satisfied with the process.

3.8.2 Possible improvements

Although we were able to forfill the product requirements there are room for improvement. 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 a future bachelor project. In addition there are also improvements that can be done with the accuracy of the demo, specificly by doing more calibration tests and make 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.

Making a product that i viable 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.

3.9 Conclusion

3.9.1 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.

Product documentation

4.1 Preface

The product documentation is a technical description of the product.

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

4.2 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:

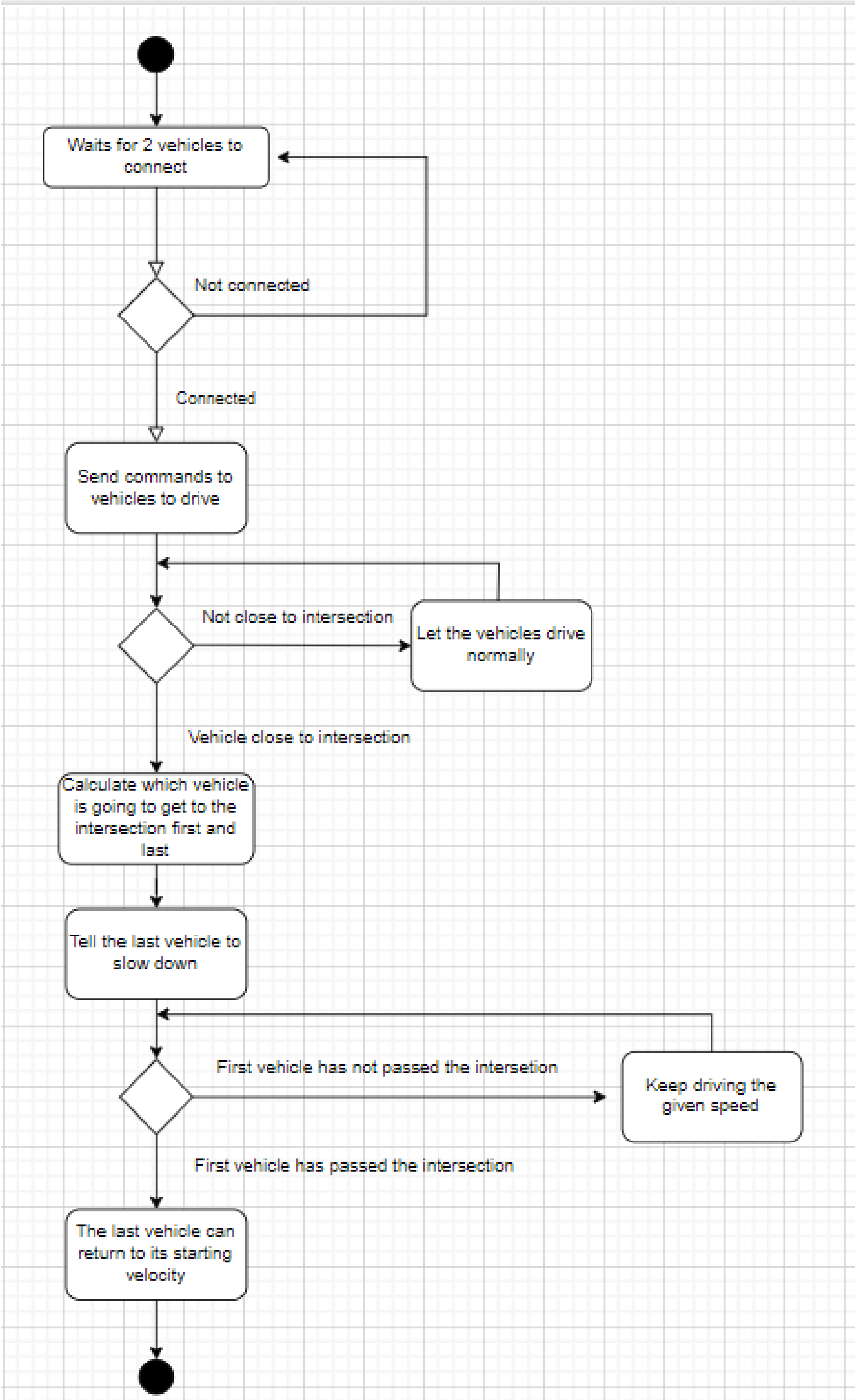

Figure 4.1: Flow diagram for the server. This diagram specificly shows the flow of the demo

4.3 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.

4.4 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.

Discussion

5.1 Evaluation of process

In our process documentation we have described the way the project progressed from beginning to end. In this section we will discuss positive and negative takeaways, with the goal of aiding further work on the project. This will include our approach to work methodology, communication and team building.

5.1.1 Work methodology

As previously mentioned, we chose to apply an agile work methodology based on the fact that our project seemed to be prone to many changes. We included rituals like daily stand-ups, sprint planning meetings and retrospective meetings after sprints. Doing this helped our team understand what difficulties the other members were facing, and gave us time to discuss our obstacles daily.

There were also some elements of agile development that we did not incorporate, either because of lack of experience or lack of time. As explained in the work of Skyttermoen, T. and Vaagasar, A.L.; agile development teams regularly have meetings with the product owner, who give feedback on the strengths and weaknesses of the deliveries (Skyttermoen & Vaagaasar, 2017, pp.120). Our group only had a few meetings with the product owner during the project period, to update him on how far along we were. We did, however, have biweekly meetings with our external supervisors at Accenture who provided us feedback on the work we had done. Because we had the freedom to choose the type of solution we wanted, as long as it met the requirements that were set to us, we didn't need the product owner to be in our biweekly meetings.

5.2 Evaluation of results

5.3 Further work

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.

5.3.1 Scalability

When the knowledge and research has come further regarding AI, there may be a possibility for such an IoT system to be scaled for the real world. There is a need for less traffic in the cities and our results from this project shows that an IoT-system where cars can communicate through a server will increase traffic flow. Scalability is therefore important which we had in mind while coding the server and the client.

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 as of now. 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.

5.3.2 Extendability to the real world applications

Edge computing

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.

5.3.3 Risk management

Conclusion