DATA3900 - Bachelorproject
Final report

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

Total pages: 21

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

Process documentation

1.1 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.

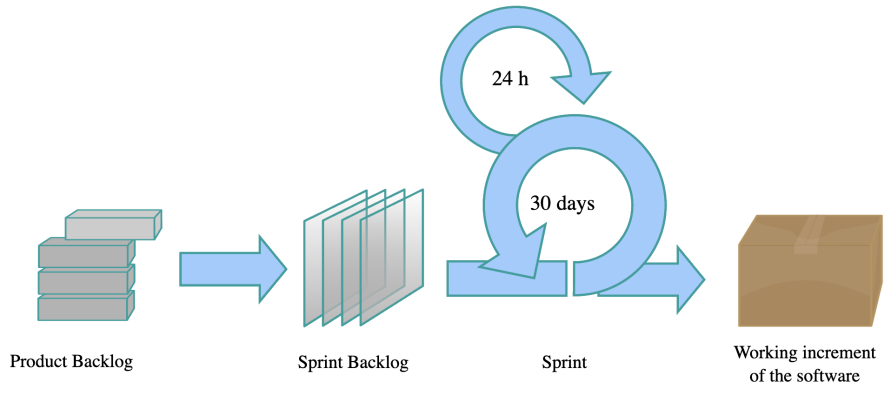

Figure 1.1: Illustration of a generic scrum process.

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 test it. If the testing is a success it gets dragged over to the "Done column". 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 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.

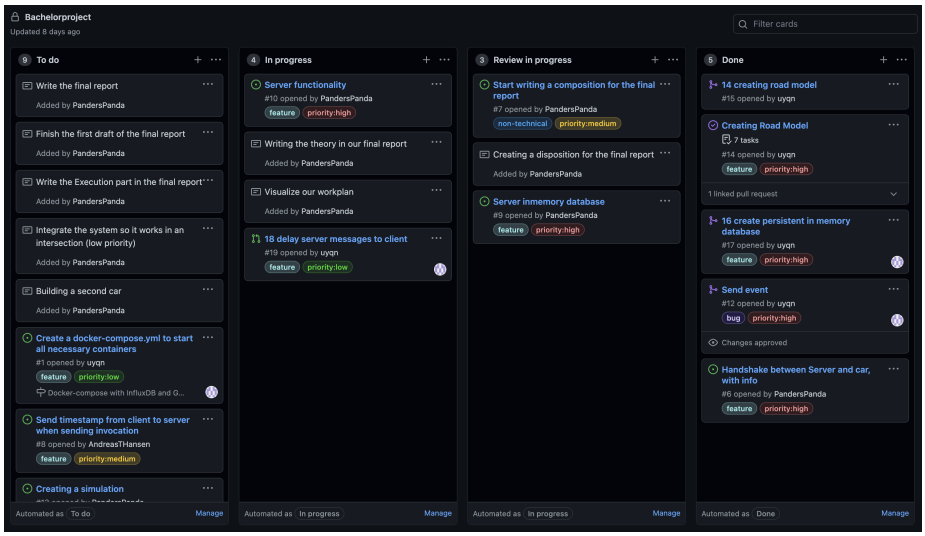

Figure 1.2: Extract of our kanban from Github.

1.2 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

2.1 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)

2.2 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 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.

2.2.1 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.

2.2.2 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 (Sinha, 2021). 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 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, with the hopes of making a solution that can be scaled up at a later point.

2.2.3 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.

2.2.4 MoSCoW method

MoSCoW method is a prioritization technique used in project managment to prioritize requirements. Since we were unsure how many use cases we finish within the time contraint of the priject period, we thought a prioritazion method was a good fit.

The "Mo" stands for must have ans is the number one priority requirements for our project. "S" stands for should have, the "Co" for could have, and "W" 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.

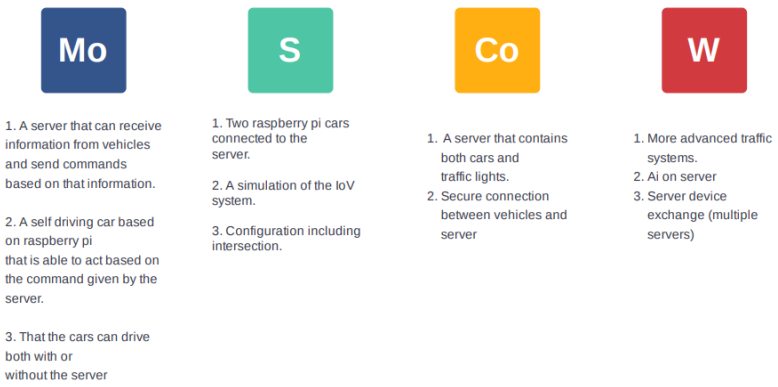

Figure 2.1: MoSCoW method

2.3 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 discussed 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 (IBM Cloud Education, 2021).

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 (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 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.

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.4 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 (Fette & Melnikov, 2011). Hence, using websockets both client and server can transfer data whenever they see fit. However, Microsoft, 2022b 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, 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).

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.

2.5 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.

2.5.1 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.

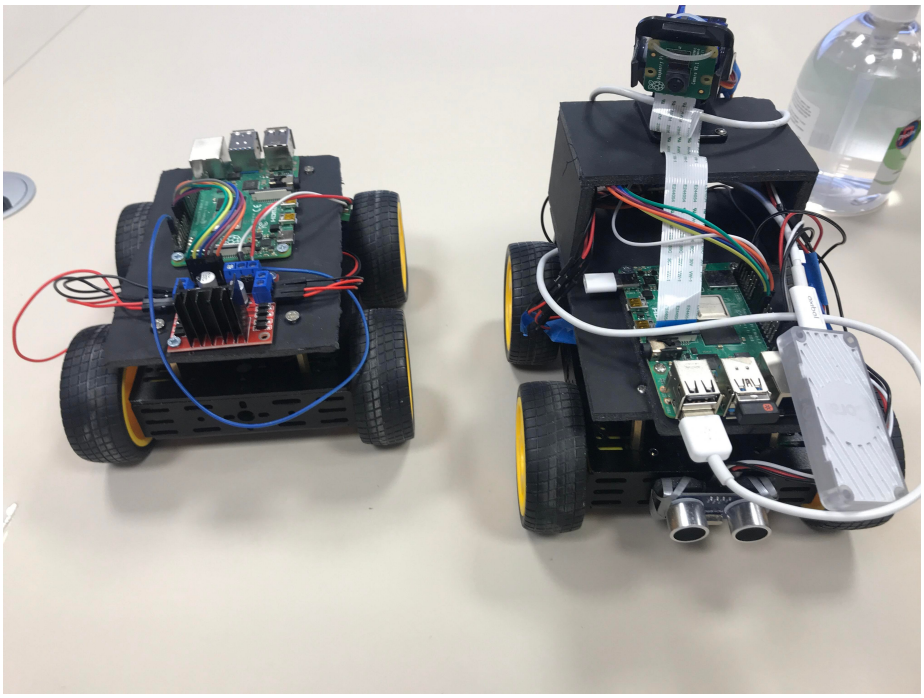

Figure 2.2: Cars with and without camera, right to left

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 2.1: Test text

2.5.2 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 directions by the server. In this test the vehicles were given a specific velocity and driving distance by the server. When the vehicles arrived at their destination the server would tell them to stop. The car's drove in a straight line.

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

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

Figure 2.3: Graph of velocity as a function of power

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

Figure 2.4: Graph of velocity as a function of power with linear regression

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

To test the solution we have worked on, we made a physical demonstration with two cars that meet at an intersection, as part of the product documentation. We want to test that a combination of a centralized communication system

and artificial intelligence can improve traffic flow. What we wanted to observe was if the velocity of the vehicles were not drasticly changed and therefore not distrupting the traffic flow.

2.5.3 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)

2.6 Results

We observed troughout multible demonstrations that the vehicles accelerated from 80 cm/s to 55 cm/s when the server intervined. 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.

Product documentation

3.1 Product specification

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.

3.1.1 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.

3.1.2 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`.

3.1.3 Patch

After initialization and handshake elaborated in 3.1.1 Initialization and 3.1.2 Handshake and listener respectively, the Raspberry Pi vehicles starts to drive into an intersection simultaneously. Throughout the journey the cars are continuously patching to the server, by calling the client's `async def send\_patch` method.

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

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

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

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

```
    ...
  }
}
```

The snippet above shows that the method first creates a new vehicle using the information provided by the `Client`. However, since this new vehicle does not contain all the information, such as the travel plan, the method first update the existing vehicle in the database in order to refresh the vehicle with the available information. It then fetch the same vehicle that was stored in the handshake, mentioned in 3.1.2 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.

3.1.4 VehiclesHubDatabase

Bibliography

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. (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/

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

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/