Truck Platooning

Distributed and Parallel Systems

1st Furkan Ali Yurdakul
Fachhochschule Dortmund
Dortmund, Germany
furkan.yurdakul001@stud.fh-dortmund.de
Matriculation number: 7215046
Contribution: 25%

2nd Bruno Hyska
Fachhochschule Dortmund
Dortmund, Germany
bruno.hyska@stud.fh-dortmund.de
Matriculation number: 7216772
Contribution: 25%

3rd Ian Murnane
Fachhochschule Dortmund
Dortmund, Germany
ian.murnane@stud.fh-dortmund.de
Matriculation number: 7216662
Contribution: 25%

4th Elbek Bakiev
Fachhochschule Dortmund
Dortmund, Germany
elbek.bakiev001@stud.fh-dortmund.de
Matriculation number: 7216249
Contribution: 25%

***Abstract*—This paper proposes a model-based approach for the specification and implementation of a distributed and parallel truck platooning system. The system comprises multiple trucks that interact and communicate with each other to form a platoon, controlled by a lead truck. We identify the required data/signal/events for communication and control behavior of the trucks. The model-based specification is mapped to code including the use of OpenMP. We develop a distributed and parallel architecture to meet system requirements and choose the appropriate parallel programming model. The simulation demonstrates the platoon's ability to drive together, detect obstacles, elect a leader, and detect node failure. The system shows its stability in cases of communication failure and its ability to maintain the distance between the trucks. Our work highlights the potential of model-based specification and implementation for developing distributed and parallel systems in truck platooning.**

Keywords—Truck platooning, data transmission, leader, follower, distributed, parallel, coordinates, speed, direction, network traffic control, functionality, requirements.

I. Motivation

The transportation sector is a critical component of the global economy, and the freight industry is a key player in this field. However, the freight industry faces several challenges, such as rising fuel costs, environmental concerns, and road safety issues. Emerging technologies, such as truck platooning, have the potential to address these challenges and revolutionize the transportation industry (Farokhi & Johansson, 2014).

Truck platooning, which involves a group of trucks traveling in close formation, is an area of research that is gaining momentum due to its potential benefits. Truck platooning has the potential to improve fuel efficiency, reduce carbon emissions, and increase road safety. However, developing a robust and efficient truck platooning system is a complex and challenging task.

The motivation for this project is to develop a model-based approach to address the challenges associated with truck platooning. Our approach involves the development of a distributed and parallel truck platooning system that leverages state-of-the-art communication protocols, control behavior, and parallel programming models.

The proposed system's potential impact is significant, as it can increase the efficiency and sustainability of the freight industry while reducing its environmental impact. It can also enhance road safety by reducing the number of accidents caused by driver fatigue or distraction. By developing a distributed and parallel truck platooning system that addresses the associated challenges, we can contribute to the development of a more sustainable, efficient, and safe transportation industry.

II. Sketch of Approach

Our team has developed a highly effective, scalable, and reliable process to create a truck platooning system. This process includes several important steps that ensure the system meets the required standards and specifications.

Firstly, we define the necessary information, signals, and actions required for the trucks to communicate with each other. We also create a detailed plan for how the system will work, using specific rules and models to define the requirements. This helps ensure that the system is reliable and efficient.

Once we have identified the necessary interactions and communications, our team focuses on determining the relevant control behaviors for the trucks. We ensure that the distance to the preceding truck is maintained, and the system remains stable and robust even in the face of communication failures. Here, we employ state machine diagrams to model these behaviors and ensure that we can accurately capture the various scenarios that the truck platooning system may encounter.

With the data and control behaviors identified, we proceed to map our model-based specifications to code using OpenMP. Initially, we implement a "pure" functional behavior that we will refine to an OpenMP implementation. By using OpenMP, we are confident that we can develop a highly efficient, scalable, and robust system that will deliver the required performance while remaining highly adaptable to the changing needs of the transportation industry.

Our approach also involves developing an appropriate distributed and parallel architecture that can meet the requirements and characteristics necessary for the truck platooning system to function correctly. We compare and evaluate different approaches and models for distributed communication and interaction, as well as node-specific parallel implementation.

In the end, we simulate our implementation using a Python environment and the Tkinted library, demonstrating the developed use cases, including trucks joining a platoon, leader election, driving in a convoy, obstacle detection, and node failure detection. Our team specifies and implements an appropriate environment for these use cases and demonstrates how our distributed and parallel system handles them.

This document includes additional research that has not yet been implemented, which can be found in the Annex section.

III. System Design

The system design phase is a crucial aspect of developing an effective and efficient truck platooning system. In this chapter, we will discuss the functional requirements, communication protocol, architecture, state machine diagram, and control behavior. These components are essential for ensuring reliable and scalable communication between the trucks. We will also discuss our approach to distributed and parallel architecture, including our implementation of the OpenMP library and state machine diagrams for modeling control behavior. This enables the trucks to maintain a safe distance and remain stable, even in the event of communication failures.

A. Functional requirements

The functional requirements for the truck platooning system define the specific functionality that the system must have in order to achieve its objectives. These requirements detail the expected behavior of the system in different scenarios.

The following table lists the functional requirements for the truck platooning system, categorized by priority, with the use of MoSCoW method (see chapter VII.A).

TABLE I. Functional Requirements

Label	Priority	Description
Development		
FR-D-1	MUST	Architecture must be concurrent, distributed and parallel
FR-D-2	SHOULD	Developed using Pure functional behavior
FR-D-3	COULD	Threading handled by OpenMP
Prerequisites		
FR-P-1	MUST	All trucks have a driver
FR-P-2	COULD	Network layer is abstracted for simplification and demonstration
FR-P-3	SHOULD	The system environment is a simplified ideal-world for demonstration purposes. This includes: No other vehicles on the test track Simple road system with 90° intersections No traffic control, signs or speed limits
High Level		
FR-H-1	MUST	Trucks can form a platoon. A platoon is defined as 2 or more vehicles consisting of a leader and followers (members)
FR-H-2	MUST	A truck in a platoon will autonomously follow a member vehicle at a specified distance. This incorporates autonomous steering and autonomous acceleration and braking.
Membership		
FR-M-1	MUST	Trucks can request to join platoon
FR-M-2	SHOULD	Join requests must be able to be accepted or rejected
FR-M-3	SHOULD	Platoon data will only be shared with members
FR-M-4	COULD	Membership can be revoked by the member
FR-M-5	MUST	The leader is the vehicle at the front
FR-M-6	MUST	Trucks can only join at the end of the queue
Communication		
FR-C-1	MUST	Followers must publish a regular heartbeat message at a specified interval
FR-C-2	MUST	All vehicles should publish their state including when they brake
Shared Data		
FR-S-1	MUST	Members list, specifying leader
FR-S-2	COULD	Destination
FR-S-3	COULD	Route (not implemented in demo system)
FS-S-4	MUST	Data is encoded into a message format
Fault Handling		
FR-F-1	SHOULD	Members will be removed when communication has timed-out
FR-F-2	SHOULD	Vehicles can couple and de-couple as required

To ensure the system meets its goals of increasing fuel efficiency and improving road safety, the functional requirements are organized into several categories. The Development requirements specify the technical aspects of the system, such as its architecture and threading. The Prerequisites requirements outline the necessary conditions for the system to operate, such as the presence of a driver in each truck. The High-Level requirements describe the core functionality of the platooning system, including the ability for trucks to form platoons and autonomously follow other vehicles. The Membership requirements detail the rules for joining and leaving a platoon, while the Communication and Shared Data requirements outline the messaging protocols and data exchange between platoon members. the Fault Handling requirements address how the system will handle errors and unexpected events.

B. Communication

The truck platooning system is designed to operate as a distributed network of autonomous vehicles, where information must be exchanged between instances to ensure safe and efficient platooning. To achieve this, a robust and reliable communication protocol is crucial. In this chapter, we will explore the communication protocol used in the system, and why we have chosen AWS IoT (MQTT) as the most suitable solution. We will also discuss the specific data that needs to be shared between instances, and how this data is structured to ensure efficient communication and synchronization.

1) Communication protocol

For the communication protocol of the truck platooning system, we conducted research into various options, including HiveMQ, ZeroMQ, and AWS IoT, which operates using the MQTT protocol.

a) *HiveMQ*

HiveMQ is a message broker that uses the MQTT protocol, designed specifically for machine-to-machine communication. MQTT is a lightweight protocol, making it well-suited for low-bandwidth and high-latency environments. HiveMQ provides a scalable and robust platform for handling millions of messages per second, which is crucial for truck platooning applications. However, we found that it had limitations when it came to integrating with other platforms and services, which led us to explore other options (Light, 2015).

In HiveMQ, messaging is done through a connection to the broker. The publisher sends a message to the broker using the hostname and port, while the subscriber receives the message by connecting to the same host and port. To publish a message, one can use the following code:

```
import ssl

from paho import mqtt
import paho.mqtt.client as paho
import paho.mqtt.publish as publish

# creating a set of 2 test messages that will
be published at the same time
msgs = [{'topic': "paho/test/multiple",
'payload': "test 1"}, ("paho/test/multiple",
"test 2", 0, False)]

# use TLS for secure connection with HiveMQ
Cloud
sslSettings =
ssl.SSLContext(mqtt.client.ssl.PROTOCOL_TLS)

# put in your cluster credentials and
hostname
auth = {'username': "<username>", 'password':
"<password>"}
publish.multiple(msgs, hostname="<hostname>",
port=<port>, auth=auth,
                 tls=sslSettings,
protocol=paho.MQTTv31)
```

To subscribe to a message, one can use the following code:

```
import ssl
import paho.mqtt.client as paho
import paho.mqtt.subscribe as subscribe
from paho import mqtt

def print_msg(client, userdata, message):
    print("%s : %s" % (message.topic,
message.payload))

sslSettings =
ssl.SSLContext(mqtt.client.ssl.PROTOCOL_TLS
)

auth = {'username': "<username>",
'password': "<password>"}
subscribe.callback(print_msg, "#",
hostname="<hostname>", port=<port>,
auth=auth, tls=sslSettings,
protocol=paho.MQTTv31)
```

While HiveMQ's MQTT protocol is built on top of TCP and requires a stable network connection, it may take some time to deliver and fetch information from the broker. For a truck platooning system, a fast and stable data exchange platform is required. Additionally, having an extra node as a broker in the messaging system of the platoon may require more time to exchange information between the trucks. This is where ZeroMQ comes in as a solution to connect the trucks in the platoon.

b) *ZeroMQ*

ZeroMQ, is a messaging library that provides high-performance messaging patterns for distributed systems. It offers a simple and lightweight interface for messaging, making it easy to use and integrate with other libraries and platforms and unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker (ZeroMQ, 2022).

The ZeroMQ (ZMQ) protocol utilizes sockets to represent connections between trucks. One connection can be initialized with the help of two different sockets. For the truck platooning project, a PAIR socket is suitable as it can operate as both a publisher and a subscriber at the same time within a single program execution (ZeroMQ, n.d.).

```
import zmq
context = zmq.Context()
socket = context.socket(zmq.PAIR)
```

Implementing the code for the leader and followers can be done identically using the PAIR socket. The only difference between the two is that the leader truck should bind a port in TCP while the follower truck connects to the same port to establish a connection. After a successful search, the leader opens a port and the follower connects to that port.

For the leader truck, the following code initializes the port:

```
socket.bind("tcp://*:%s" % '<port_number>')
```

And for the follower truck, the following code connects to the port:

```
socket.connect("tcp://localhost:%s" %
'<port_number>')
```

Once the connection is established, the two sockets are ready for messaging. Data can be sent and received in various formats including integers, strings, bytes, and JSON.

While ZMQ provides a simple and efficient way for communication between trucks, it lacks the advanced features and security protocols required for truck platooning, making it unscalable, unreliable and unsuitable for our needs. This is where AWS IoT with its built-in security features and MQTT protocol proves to be a better solution.

c) *AWS IoT*

AWS IoT is a fully managed service provided by Amazon Web Services that enables secure and scalable communication between Internet of Things (IoT) devices and the cloud. It provides a reliable and secure way to connect, manage, and interact with IoT devices, and allows the devices to securely communicate with other devices and cloud services (Amazon, 2023).

The communication protocol used by AWS IoT is MQTT, which is a lightweight and efficient protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. MQTT is based on the publish-subscribe model, in which devices publish messages to a broker that routes them to the appropriate subscribers. This enables the devices to decouple the production and consumption of data and simplifies the architecture of the system.

To implement AWS IoT in our project, we will use the AWS IoT Device SDK for C++, which provides a simple and consistent programming model for interacting with the AWS IoT service. This SDK supports the MQTT protocol, and provides features such as message buffering, automatic reconnect, and device shadow synchronization. By using this SDK, we can quickly and easily connect our platoon vehicles to the AWS IoT service, and leverage its features for secure and reliable communication.

First, the SDK needs to be fetched from the git repository of AWS IoT. This SDK can then be added to our project to make use of it (Amazon, 2023).

The following example could be used to Publish/Subscribe to a topic within AWS IoT using the SDK (Amazon, 2022):

```
    // Set up the MQTT client
    MqttClient client("my-client-id", "my-
endpoint", "my-certificate", "my-private-
key", "my-root-ca");

    // Connect to AWS IoT
    client.Connect();

    // Subscribe to a topic
    client.Subscribe("my/topic", [](const
std::string& topic, const std::string&
payload) {
        std::cout << "Received message on
topic " << topic << ": " << payload <<
std::endl;
    });

    // Publish a message to the topic
    client.Publish("my/topic", "hello,
world");

    // Wait for messages to arrive
    std::this_thread::sleep_for(5s);

    // Disconnect from AWS IoT
    client.Disconnect();
```

In this example, you need to replace the placeholders my-client-id, my-endpoint, my-certificate, my-private-key, and my-root-ca with the appropriate values for your AWS IoT setup.

The MqttClient class provides a high-level interface to the MQTT protocol for connecting to AWS IoT, subscribing to topics, and publishing messages. The Connect method establishes a connection to AWS IoT, and the Subscribe method registers a callback function to handle incoming messages on the specified topic. The Publish method sends a message to the specified topic. Finally, the Disconnect method closes the connection to AWS IoT.

In conclusion, after exploring various communication protocols and brokers, we found that AWS IoT was the most suitable choice for our distributed system. Its scalability, reliability, security, and ease of integration make it an ideal platform for large-scale IoT deployments. With AWS IoT, we were able to build a robust and efficient communication system that enables safe and efficient truck platooning operations.

C. Architecture

To develop an efficient and safe platooning system, an appropriate distributed and parallel architecture must be designed from a programming perspective. This architecture should support real-time communication, and be fault-tolerant, scalable, flexible, and performant. One way to do this is to use a message-passing architecture that enables fast and reliable communication between the different trucks in the platoon. This can be achieved through the use of message-oriented middleware(MQTT) (MQTT, 2023), which provides a lightweight and efficient means of transmitting data in distributed systems. In addition, using a message-passing architecture enables us to make use of parallel programming models like OpenMP, which can be used to parallelize tasks and make efficient use of shared memory systems.

OpenMP is a popular programming model that can be used to write multi-threaded and shared-memory parallel programs in C++. It is supported by most modern compilers and is widely used in scientific computing and other fields that require high-performance computing. OpenMP provides a set of directives that enable the programmer to specify which parts of the code can be executed in parallel and a set of library functions that provide synchronization and communication mechanisms between threads. OpenMP is particularly well-suited to parallelize loops and other iterative constructs, which makes it a good fit for platooning algorithms that require real-time data processing and decision-making.

Another requirement for a platooning architecture is fault tolerance. Accomplishing this objective, the platooning system should be designed with redundancy, checkpoints, and error detection mechanisms, such as heartbeats. OpenMP provides the ability to implement fault-tolerance mechanisms, such as checkpointing and task migration, which can help to ensure that the system remains operational even in the face of unexpected events.

The system should be designed with modular and flexible software architectures, such as microservices or service-oriented architectures. OpenMP can also help to improve scalability by enabling the program to take advantage of multiple cores and processors. OpenMP also supports dynamic task scheduling, which enables the program to balance the workload across threads dynamically, which helps to optimize resource usage and improve the overall efficiency of the system.

What is also worth mentioning is, performance optimization is a critical requirement for a platooning system, which is often required to operate in real-time and in demanding scenarios. Having this in mind, the platooning system should be optimized for high-performance computing, and designed to minimize the time and resources required for message passing and computation. OpenMP can be used to parallelize computationally intensive tasks, which helps to reduce the overall computation time and improve system performance.

1) Distributed architecture

Distributed architecture is a computing model where different computing resources are spread across a network and work together to perform a specific function. In a distributed architecture, the computing resources can be physical or virtual and are connected to each other via a communication network. The computing resources in a distributed architecture work collaboratively to achieve a common goal and can be added or removed dynamically to scale the system up or down.

In the context of truck platooning, distributed architecture refers to the use of multiple computing resources spread out across the vehicles, which are connected to each other via a network. Each vehicle in the platoon has its own computing resource, which communicates with the computing resources of the other vehicles.

To enable the communication between the computing resources, a cloud-based service, such as AWS IoT Core, can be used. AWS IoT Core enables devices to connect securely to the cloud and exchange data with other devices and applications. The computing resources on the vehicles publish their information to a specific MQTT topic, which is then subscribed to by the other computing resources in the platoon(see Figure 1).

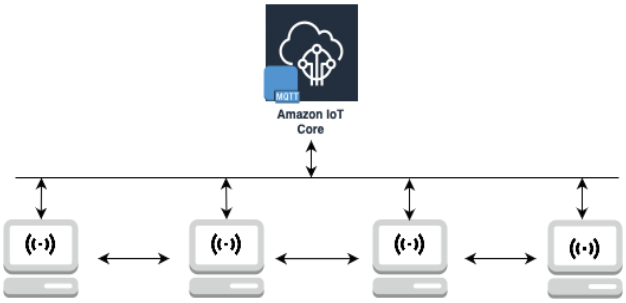

Figure 1: Expected distributed structure for demonstration

The platoon vehicles share a standardized data set, including information on their position, speed, direction, and status:

```
{
   "id": string,
   "x": double,
   "y": double,
   "isBraking": bool,
   "speed": double,
   "direction": int,
   "joined": time_t,
   "received": time_t,
}
```

The "id" field represents a unique identifier for each vehicle, while "x" and "y" represent the vehicle's current position. The "isBraking" field indicates whether the vehicle is currently braking, while "speed" and "direction" represent the vehicle's current speed and direction. "joined" and "received" indicate the time at which the vehicle joined the platoon and the time at which the data was received, respectively. By sharing this data, the platoon vehicles can maintain a synchronized and coordinated behavior.

2) Parallel architecture

Parallel architecture is a computing model where multiple computing resources work together to execute a program or a task in parallel. In parallel computing, the processing of a task is divided into smaller parts that can be executed concurrently by multiple computing resources, such as CPUs or GPUs. This allows the task to be completed faster and more efficiently.

OpenMP provides a set of directives and libraries that can be used to implement parallelism in shared memory systems, such as multi-core processors. OpenMP allows developers to specify which parts of their code should be parallelized and how the processing should be divided among multiple processors.

In the context of truck platooning, parallel architecture and OpenMP can be used to improve the efficiency and safety of the platoon by processing and analyzing the data collected from the vehicles more quickly. For example, if a fleet of trucks is equipped with sensors that collect data on factors such as speed, location, and traffic conditions, the data can be divided into smaller parts and processed concurrently on multiple processors using OpenMP. This allows for faster analysis of the data, which can be used to optimize the operation of the platoon.

OpenMP can also be used to simulate the behavior of the truck platoon. By simulating the platoon using multiple processors, the simulation can run faster, allowing fleet managers to test and evaluate different scenarios more quickly. This can help to identify potential safety issues and optimize the performance of the platoon.

a) OpenMP implementation

The code provided below is a C++ that uses OpenMP to run parallel sections of code. One of the sections runs a VehicleControl function, and another section opens a connection to AWS IoT via MQTT and publishes the *vehicleModel*. (OpenMP, 2021).

Firstly, the VehicleControl class makes use of OpenMP in the main loop. Within this loop, the critical section of code ensures that only one thread can access and modify the vehicleModel at any given time, avoiding any data race or concurrent modification issues. By parallelizing the process of updating the vehicleModel, we can take advantage of multiple cores and make the code run faster.

```
VehicleControl::VehicleControl(Document
&vehicleModel, ThreadSafeQueue
&threadSafeQueue) {
. . .
    while (true) {
        #pragma omp critical
        {
            // get the position
            position =
make_pair(vehicleModel["x"].GetDouble(),
vehicleModel["y"].GetDouble());

            // update the position and
direction
            direction =
changeDirectionAtBoundary(position,
direction);
            position = move(position,
direction);
            // write to the vehicleModel

vehicleModel["x"].SetDouble(position.first);

vehicleModel["y"].SetDouble(position.second);
            double speed =
```

```
abs(direction.first) + abs(direction.second);

vehicleModel["speed"].SetDouble(speed);
            vehicleModel["is-
braking"].SetBool(speed != SPEED_LIMIT * 2);
            // add direction
            double angle =
atan2(direction.second, direction.first) *
180 / M_PI;
            if (angle < 0) angle += 360;

vehicleModel["direction"].SetInt(static_cast<
int>(angle));
        }
        // tell the AWS section to publish
the vehicleModel
        threadSafeQueue.push("update");


this_thread::sleep_for(chrono::milliseconds(I
NTERVAL_TIME_MS));
    }
}
```

Secondly, the Communication class also makes use of OpenMP to print debug messages to the console. By using a critical section of code, we can ensure that multiple threads do not write to the console at the same time, causing confusion or incorrect debug messages.

```
Communication::Communication(ThreadSafeQueue
threadSafeQueue,
std::shared_ptr<Mqtt::MqttConnection>
connection, String clientId, String topic) {
    _connection = connection;
    _topic = (&topic)->c_str();

    auto onConnectionCompleted =
[&](Mqtt::MqttConnection &, int errorCode,
Mqtt::ReturnCode returnCode, bool) {
        if (errorCode) {
            fprintf(stdout, "Connection
failed with error %s\n",
ErrorDebugString(errorCode));

connectionCompletedPromise.set_value(false);
        }
        else {
            #pragma omp critical
            fprintf(stdout, "Connection
completed with return code %d\n",
returnCode);


connectionCompletedPromise.set_value(true);
        }
    };
. . .
}
```

It is worth noting that OpenMP was used extensively throughout the project to optimize performance and improve efficiency. The two code snippets presented here are just a few examples of how OpenMP was used to enhance the functionality of the system.

D. Control behavior

The behavior of the trucks in the platooning network is a critical component of our system's functionality. To ensure the safe and efficient operation of the platoon, we have implemented a control behavior that governs the actions of each truck based on its role within the platoon. This control behavior is defined using state machine diagrams and activity diagrams that detail the sequence of actions and states that each truck can transition through as it operates within the platoon. By utilizing these diagrams, we can ensure that each truck operates in a safe and predictable manner, minimizing the risk of collisions or other incidents. In this section, we will explore the details of this control behavior and how it is implemented within our system.

1) State Machine Diagram

State machines are useful for modeling systems that have a complex sequence of states and transitions between those states. In the case of Leader Truck and Follower Truck, state machines can be used to model the behavior of these vehicles as they move together on the road.

A state machine can help us to define and control the behavior of the Leader Truck and Follower Truck in a coordinated manner, based on the different states they can be in and the transitions between those states.

The diagram below(see Figure 2) shows the states and transitions for a leader truck in the platoon. It includes states such as "Idle", "Waiting Follower", "Sharing Data", and "Platooning". The transitions indicate how the leader truck moves between these states based on various events.

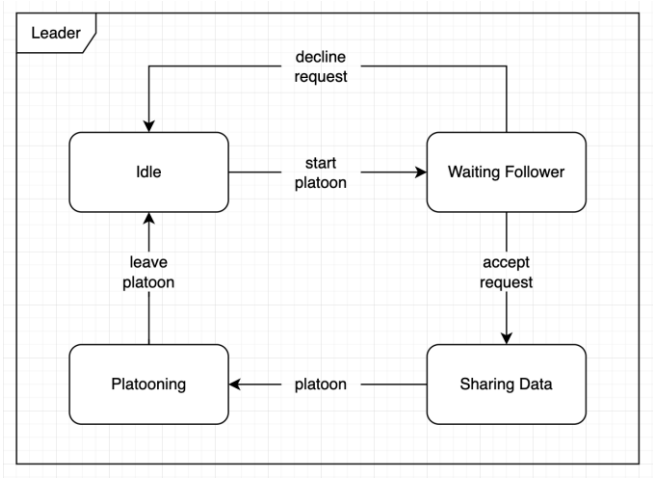

Figure 2 Leader Truck State Machine Diagram

States descriptions:

Idle: The initial state of the leader truck, where the truck is not currently in a platoon and is waiting for a follower truck to request to join the convoy.

Waiting Follower: The state where the leader truck is waiting for a follower truck to request to join the convoy. The truck will remain in this state until a follower truck request to join the convoy is received and accepted. The leader could decline a request due to various reasons. If it happens truck returns to the *Idle* state.

Sharing data: The state where data sharing between the leader truck and follower truck is carried out. In this state data between two trucks is synchronized to let them start the platooning.

Platooning: The state where the leader truck is currently leading a platoon. In this state, the truck is constantly transmitting and receiving data to and from other trucks in the platoon, while also using its sensors to monitor the road conditions. It may happen that the leader truck leaves a platoon. If it happens the second truck that was driving after the leader becomes a leader and the platoon continues to exist and move.

The following diagram(see Figure 3) shows the states and transitions for a follower truck in the platoon. It includes states such as "Idle", "Waiting for response", " Sharing Data", and " Platooning ". The transitions indicate how the follower truck moves between these states based on various events.

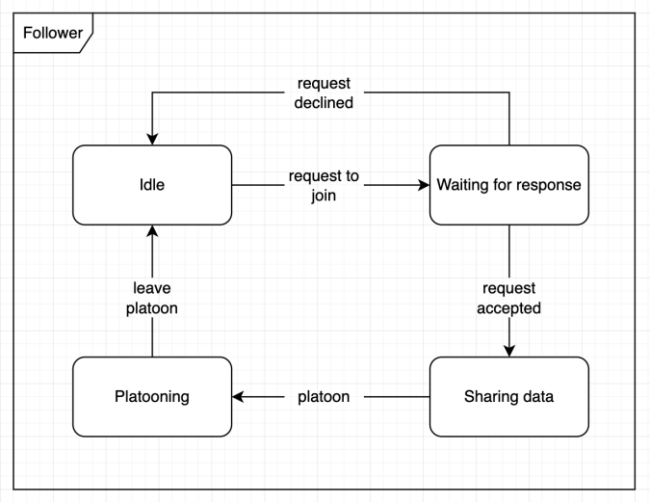

Figure 3 Follower Truck State Machine Diagram

Idle: The initial state of the follower truck, where the truck is not currently in a platoon and is waiting for a driver to request to join the convoy or a single leader truck that wants to start a platoon.

Waiting for response: The state where the follower truck is waiting for a leader truck to accept the request to join the convoy. The truck will wait until a leader truck either accepts or declines a request to join the convoy. The leader could decline a request due to various reasons. If it happens truck returns to the *Idle* state.

Sharing data: The state where data sharing between the leader truck and follower truck is carried out. In this state data between two trucks is synchronized to let them start the platooning.

Platooning: The state where the follower truck is currently following a platoon. In this state, the truck is constantly transmitting and receiving data to and from other trucks in the platoon, while also using its sensors to monitor the road conditions.

The Truck Platooning system relies on manual input to start the platoon, request, and acceptance for trucks to join the platoon, data sharing for smooth platooning, leader change when necessary, and disbanding the platoon when the drivers desire. The state machine diagrams for the follower and the leader provide a visual representation of the states and transitions that occur during platooning. The system flow and the state machine diagrams help to understand the functioning of the Truck Platooning project.

2) Activity diagram

The activity diagram for controlling the distance between trucks in a platoon(see Figure 4) shows how the trucks can maintain a safe distance from the vehicle in front. The system uses sensors to constantly check the distance to the preceding truck. The safe distance is evaluated based on the time taken by the follower truck to cover the distance between itself and the leader truck. If the distance is less than the safe distance threshold, the truck gradually decreases its speed. The process repeats until the safe distance is reached, ensuring that the distance to the preceding truck is guaranteed.

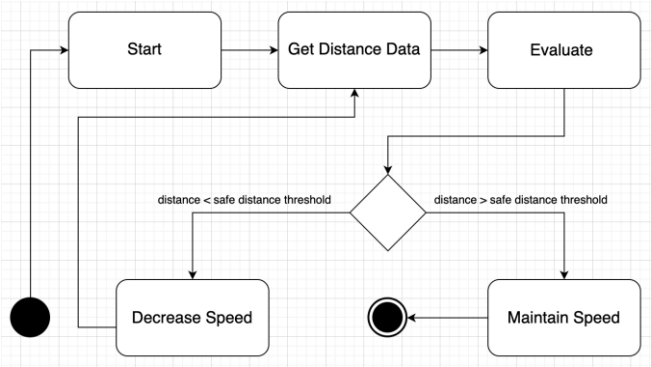

Figure 4 How the distance is controlled by trucks (Activity Diagram)

Start: The truck is in motion and needs to maintain a safe distance from the vehicle in front.

Get Distance Data: The truck's sensors constantly check the distance to the vehicle in front.

Evaluate: If the distance is greater than the safe distance threshold, the truck maintains its current speed.

Decrease Speed: If the distance is less than the safe distance threshold, the truck gradually decreases its speed.

Get Distance Data Again: After decreasing its speed, the truck checks the distance to the vehicle in front again.

Evaluate: If the distance is still less than the safe distance threshold, the truck further reduces its speed and repeats this process until the distance is within the safe threshold.

Maintain Speed: If the distance is within the safe threshold, the truck maintains its current speed.

End: The truck continues to monitor the distance and adjust its speed as necessary.

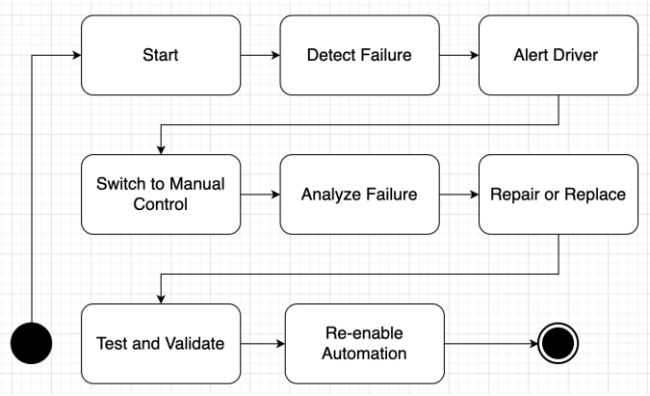

Figure 5 System behavior in cases of failure (Activity Diagram)

The activity diagram for system behavior in cases of failure(see Figure 5) outlines the process by which the Truck Platooning system handles failures. The system is designed to be robust and stable in cases of communication failure, where the trucks might not be able to share data with each other. If a failure is detected, the system triggers an alarm and alerts the driver to take manual control of the vehicle. The driver takes over control of the vehicle and disables the automated platooning system. Once the system is repaired or replaced, it is tested and validated before being re-enabled. This ensures that the system is stable and functional before it is used again.

Start: The Truck Platooning system is in operation and functioning normally.

Detect Failure: If the system detects a failure, it triggers an alarm and enters the "Failure" state.

Alert Driver: The system alerts the driver about the failure and prompts them to take manual control of the vehicle.

Switch to Manual Control: The driver takes over control of the vehicle and disables the automated platooning system.

Analyze Failure: The system performs a root cause analysis of the failure to determine the reason for the system failure.

Repair or Replace: Based on the analysis, the system is repaired or replaced as necessary.

Test and Validate: The system is tested and validated to ensure that it is functioning properly before being used again.

Re-enable Automation: Once the system is tested and validated, the driver can re-enable the automated platooning system and continue operation.

End: The system resumes normal operation with the automated platooning system functioning properly.

The distance control diagram shows how the system ensures the distance to the preceding truck is guaranteed, while the failure handling diagram outlines the steps taken to maintain system stability and reliability in cases of failures. These visual representations make it easier to comprehend complex systems and ensure that they function effectively in real-world situations.

IV. Implementation

The implementation chapter outlines the technical details of the truck platooning project. It includes the code implementation of the system, as well as the visualization of the platooning results using a Python script with the Tkinter library (FH-Dortmunders, 2023).

A. Code

This section is crucial to understand how our system works and how each module interacts with one another. As part of this discussion, we present two diagrams that illustrate the functional behavior of our code: a class diagram and an activity diagram.

1) Class diagram

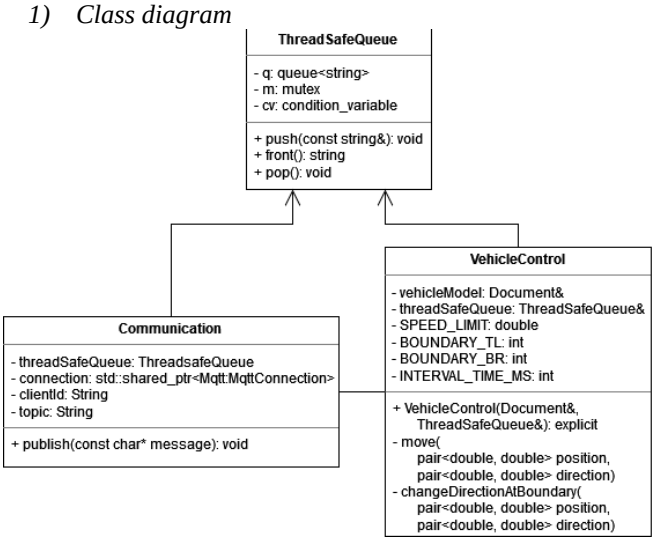

Figure 6: Class diagram of the main code (C++)

The class diagram in Figure 6 illustrates the relationships and dependencies between the three classes of the system: VehicleControl, Communication, and ThreadSafeQueue.

VehicleControl represents the vehicle model and control system and is responsible for performing the necessary calculations and logic to move the vehicle. It makes use of an instance of ThreadSafeQueue to communicate with the Communication class.

Communication is responsible for handling the MQTT communication protocol and sending messages to a remote server. It also makes use of an instance of ThreadSafeQueue to receive messages from the VehicleControl class.

ThreadSafeQueue is a simple class that provides a thread-safe implementation of a queue data structure. Both VehicleControl and Communication use an instance of ThreadSafeQueue to safely exchange messages between the two classes.

See chapter VII.C.1) for the functional program that makes use of these classes.

2) Activity diagram

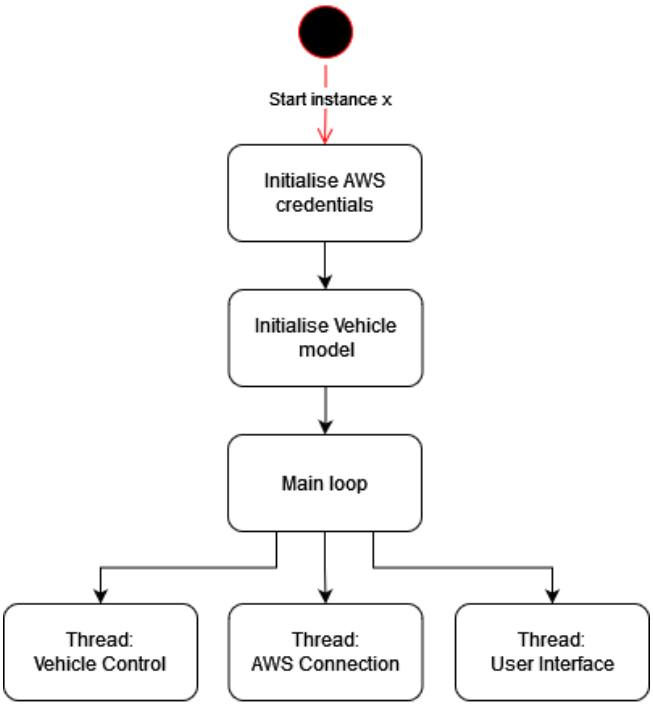

Figure 7: Activity diagram of the main code (C++)

In Figure 7 we can see the activity diagram where the program starts by creating an instance, and then it initializes the necessary AWS credentials and the vehicle model. After that, it enters the main loop where it creates three threads to handle the Vehicle Control, AWS Connection, and User Interface tasks.

The first thread is the Vehicle Control thread, which controls the movement and direction of the vehicle within the platoon. This thread is implemented using OpenMP, a programming framework for parallel processing, to improve the efficiency of the code. In this thread, the vehicleControl object is created, which takes in the vehicle model and thread-safe queue objects as inputs.

The AWS Connection thread handles the communication with AWS IoT. It uses the thread-safe queue to ensure that communication between the threads is synchronized and secure. This thread is responsible for exchanging information between the platoon vehicles and the cloud server.

The third thread is the User Interface thread, which allows the user to provide manual inputs to request joining or leaving the platoon. This thread also uses the thread-safe queue for communication with other threads.

See chapter VII.C.1)**Error! Reference source not found.** for the functional program that shows the code as the activity diagram states.

B. Visualization

The visualization part starts with fetching information from the cloud. The trucks in the platoon use AWS IoT to publish their data to the cloud. The visualization module is represented in Python code which subscribes to the cloud, gets information about every truck, and updates a plot according to the received data.

The code which below represents creating a canvas:

```
from tkinter import *

WIDTH = 800
HEIGHT = 650

win = Tk()
win.title("Truck Canvas")
win.geometry("%sx%s" % (WIDTH, HEIGHT))
canvas = Canvas(win, width=WIDTH,
height=HEIGHT, background="white")
```

The Tkinter canvas is a graphical user interface element that provides a drawing area for the script to display the trucks' positions. The canvas is created using the Tkinter library, which is a built-in Python library for creating GUI applications. The canvas is initialized with a fixed size and a white background.

Pictures of the truck are placed on the canvas in the following way:

```
image = PhotoImage(file="truck01.png")
resized_image = image.subsample(4, 4)
canvas.create_image(100, 100, anchor=CENTER,
image=resized_image, tags="truck01")
canvas.pack()
```

To display the truck images on the canvas, a PNG image of a truck is loaded using the Tkinter *PhotoImage* class. The image is then resized using the subsampling method to make it smaller, and it is assigned to a *resized_image* variable. The resized image is then used to create a *create_image* object on the canvas, which is positioned at a specific coordinate on the canvas.

The following example code is used to establish a connection between the visualization module and the cloud:

```
mqtt_connection =
cmdUtils.build_mqtt_connection(on_connection_
interrupted, on_connection_resumed)
    connect_future =
mqtt_connection.connect()
    connect_future.result()
    message_topic =
cmdUtils.get_command(cmdUtils.m_cmd_topic)

    subscribe_future, packet_id =
mqtt_connection.subscribe(
        topic=message_topic,
        qos=mqtt.QoS.AT_LEAST_ONCE,
        callback=on_message_received)
    subscribe_result =
subscribe_future.result()
```

The first line creates an MQTT connection object using *cmdUtils.build_mqtt_connection* function and passes two callback functions *on_connection_interrupted* and *on_connection_resumed* as arguments. These callback functions are triggered when the connection is lost or re-established:

```
def on_connection_resumed(connection,
return_code, session_present, **kwargs):
    print("Connection resumed. return_code:
{} session_present: {}".format(return_code,
session_present))

    if return_code ==
mqtt.ConnectReturnCode.ACCEPTED and not
session_present:
        print("Session did not persist.
Resubscribing to existing topics...")
        resubscribe_future, _ =
connection.resubscribe_existing_topics()

resubscribe_future.add_done_callback(on_resub
scribe_complete)

def on_resubscribe_complete(
resubscribe_future):
    resubscribe_results =
resubscribe_future.result()
    print("Resubscribe results:
{}".format(resubscribe_results))

    for topic, qos in
resubscribe_results['topics']:
        if qos is None:
            sys.exit("Server rejected
resubscribe to topic: {}".format(topic))
```

The *on_message_received* function is a callback function that gets called whenever a message is received on the subscribed topic. The function first parses the JSON data from the message and stores it in a *trucks_dicts* dictionary. The dictionary maps each truck's ID to its position and other data. The function then updates the position of the corresponding truck on the canvas using the *canvas.coords* function. The *canvas.coords* function takes the ID of the truck, the new x and y positions of the truck, and updates the position of the truck on the canvas.

```
trucks_dicts = {}

def on_message_received(topic, payload, dup,
qos, retain, **kwargs):
    data = json.loads(payload.decode('utf-
8'))

    if not trucks_dicts.get(data['id']):
        trucks_dicts[data['id']] = {'num':
len(trucks_dicts) % 6 + 1}

canvas.coords(f"truck0{trucks_dicts[data['id'
]]['num']}", data["x"], data["y"])
```

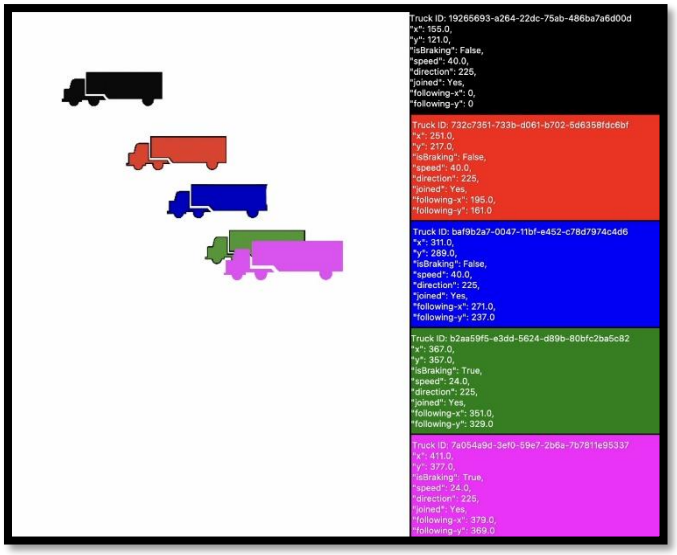

Figure 8: Demonstration of truck platooning

In Figure 8 we can see the end result of using tkinter to visualize our data. The trucks are visible on the left side, and through color coding we match the incoming data shown on the right side.

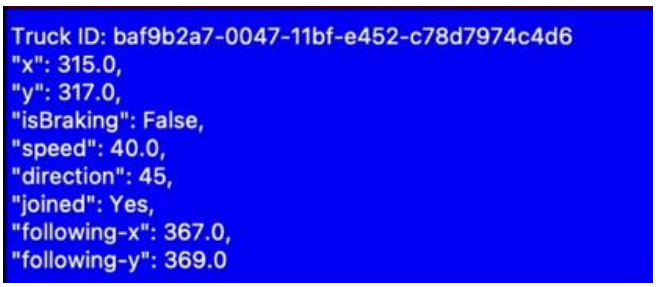

Figure 9: Incoming data per truck

As we explained earlier about our way of a distributing system we share a specific set of data, which can be seen on the right side of the canvas. See Figure 9 for a zoomed image of the canvas.

V. Conclusion

This project successfully demonstrates the potential for platooning of autonomous vehicles through the use of AWS IoT and C++ programming. The developed solution allows vehicles to join and leave platoons, adjust their speed and direction based on platoon data, and send messages to and from the AWS cloud.

During the development process, the team encountered various challenges and worked through them to produce a functional solution. However, it is worth noting that there is still room for improvement and expansion of the system. For instance, the team considered exploring the use of GPU for data parallelism, which could have improved performance in message queuing or calculating paths after joining a platoon.

This project highlights the potential for advanced technologies, such as AWS IoT and C++ programming, to facilitate the growth and development of autonomous driving systems. It also showcases the importance of continuous improvement and the exploration of new ideas to push the boundaries of what is possible.

A. Acknowledgment

The authors would like to extend their gratitude to Prof. Dr. Stefan Henkler and M. Eng. Noura Sleibi who provided classes about Distributed and Parallel Systems. Specifically, Prof. Dr. Stefan Henkler who supported us throughout the journey of this project.

VI. References

Amazon. (2022, December 9). *Sample apps for the AWS IoT Device SDK for C++ v2*. Retrieved from GitHub: https://github.com/aws/aws-iot-device-sdk-cpp-v2/tree/main/samples

Amazon. (2023). *AWS IoT Core - Developer Guide.* Retrieved from Amazon: https://docs.aws.amazon.com/pdfs/iot/latest/developerguide/iot-dg.pdf#what-is-aws-iot

Amazon. (2023, January 27). *AWS IoT Device SDK for C++ v2*. Retrieved from GitHub: https://github.com/aws/aws-iot-device-sdk-cpp-v2

Farokhi, F., & Johansson, K. H. (2014, January 21). *A Study of Truck Platooning Incentives.* Retrieved from arxiv: https://arxiv.org/pdf/1310.5534.pdf

FH-Dortmunders. (2023, February 19). *DPS-Project*. Retrieved from GitHub: https://github.com/bakievelbek/DPS-project

Light, R. (2015, September 28). *Paho Python - MQTT Client Library Encyclopedia.* Retrieved from HiveMQ: https://www.hivemq.com/blog/mqtt-client-library-paho-python/

MQTT. (2023). *MQTT: The Standard for IoT Messaging*. Retrieved from MQTT: https://mqtt.org/

OpenMP. (2021). *OpenMP 5.2 API Syntax Reference Guide.* Retrieved from OpenMP: https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf

Wikipedia. (2022, December 20). *MoSCoW method*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/MoSCoW_method

ZeroMQ. (2022). *Documentation.* Retrieved from ZeroMQ: https://zeromq.org/get-started/

ZeroMQ. (n.d.). *Chapter 1 - Basics.* Retrieved from ZeroMQ: https://zguide.zeromq.org/docs/chapter1/

VII. Annex

A. MoSCoW

The MoSCoW method is a prioritization technique used to categorize requirements or features into four groups: Must have, Should have, Could have, and Won't have (Wikipedia, 2022). The method is designed to help project teams and stakeholders to identify and focus on the most critical features or requirements, and to make informed decisions about which features to prioritize.

The four categories are defined as follows:

- Must have: Requirements or features that are critical to the success of the project and must be included in the final product. These are essential features that are necessary for the product to work as intended, and failure to deliver these features could result in the project being deemed a failure.
- Should have: Requirements or features that are important but not critical to the success of the project. These features are desired and would be beneficial to have but are not essential for the product to function properly.
- Could have: Requirements or features that are desirable but not necessary. These features are nice-to-have and can be implemented if there is time and budget available but are not a priority.
- Won't have: Requirements or features that are not included in the project, either because they are not important or because they are not feasible.

Using the MoSCoW method, project teams can prioritize their requirements or features and allocate their resources accordingly. This ensures that the most important features are implemented first, and that the project stays on track and within budget.

B. Contribution

All authors contributed to the writing and reviewing of the final draft, also to the implementation of the project and approved the submitted version. Each author contributed equal hours of work, which results into 25% of contribution from each author.

C. Code

1) Functional programming C++

```
#include <aws/crt/UUID.h>
#include "rapidjson/document.h"
#include "rapidjson/prettywriter.h"
#include "rapidjson/stringbuffer.h"

#include <chrono>
#include <conio.h>
#include <ctime>
#include <iostream>
#include <omp.h>
#include <mutex>
#include <vector>

#include "CommandLineUtils.h"
#include "Communication.h"
#include "ThreadSafeQueue.h"
#include "VehicleControl.h"

using namespace Aws::Crt;
using namespace rapidjson;
using namespace std;

// find the last vehicle in the platoon
String getLastVehicleId(vector<Document> &vehicles) {
    uint32_t lastJoinedDate = 0;
    String lastJoinedId = "";

    for (auto &doc : vehicles) {
        uint32_t joined = doc["joined"].GetUint();
        if (joined > lastJoinedDate) {
            lastJoinedDate = joined;
            lastJoinedId = doc["id"].GetString();
        }
    }

    return lastJoinedId;
}

int main(int argc, char *argv[]) {
    srand((uint32_t) time(NULL)); // Seed the random number generator

    ApiHandle apiHandle;

    /************************ Parse Arguments ****************************/
    Utils::CommandLineUtils cmdUtils = Utils::CommandLineUtils();
    cmdUtils.RegisterProgramName("basic_pub_sub");
    cmdUtils.AddCommonMQTTCommands();
    cmdUtils.RegisterCommand("key", "<path>", "Path to your key in PEM format.");
    cmdUtils.RegisterCommand("cert", "<path>", "Path to your client certificate in PEM format.");
    cmdUtils.AddCommonProxyCommands();
    cmdUtils.AddCommonTopicMessageCommands();
    cmdUtils.RegisterCommand("client_id", "<str>", "Client id to use (optional, default='test-*')");
    cmdUtils.RegisterCommand("port_override", "<int>", "The port override to use when connecting (optional)");
    cmdUtils.AddLoggingCommands();
    const char **const_argv = (const char **)argv;
    cmdUtils.SendArguments(const_argv, const_argv + argc);
    cmdUtils.StartLoggingBasedOnCommand(&apiHandle);
```

```
String topic = cmdUtils.GetCommandOrDefault("topic", "test/topic");
String clientId = cmdUtils.GetCommandOrDefault("client_id", String("test-") + Aws::Crt::UUID().ToString());

auto connection = cmdUtils.BuildMQTTConnection();

// initialise this vehicles model
String vehicleLabel = UUID().ToString();
Document vehicleModel;
vehicleModel.SetObject();
Value id(vehicleLabel.c_str(), vehicleModel.GetAllocator());
Value x(rand() % 301 + 150);
Value y(rand() % 301 + 150);
Value isBraking(false);
Value speed(0.0);
Value direction(0);
Value joined(0);
Value followingX(0);
Value followingY(0);
vehicleModel.AddMember("id", id, vehicleModel.GetAllocator());
vehicleModel.AddMember("x", x, vehicleModel.GetAllocator());
vehicleModel.AddMember("y", y, vehicleModel.GetAllocator());
vehicleModel.AddMember("is-braking", isBraking, vehicleModel.GetAllocator());
vehicleModel.AddMember("speed", speed, vehicleModel.GetAllocator());
vehicleModel.AddMember("direction", direction, vehicleModel.GetAllocator());
vehicleModel.AddMember("joined", joined, vehicleModel.GetAllocator());
vehicleModel.AddMember("following-x", followingX, vehicleModel.GetAllocator());
vehicleModel.AddMember("following-y", followingY, vehicleModel.GetAllocator());

cout << endl << "Starting with ID: " << vehicleLabel << endl;
cout << "Processors " << omp_get_num_procs() << endl << endl;

omp_set_nested(1);
ThreadSafeQueue threadSafeQueue;
vector<Document> vehicles;
String following = "";

#pragma omp parallel sections default(none) shared(cout, threadSafeQueue, connection, clientId, topic, following,
vehicles, vehicleModel) num_threads(3)
{
    // VEHICLE CONTROLLER
    #pragma omp section
    {
        #pragma omp critical
        cout << "* Starting VehicleControl in Thread: " << omp_get_thread_num() << endl;

        auto vehicleControl = VehicleControl(vehicleModel, threadSafeQueue);
    }

    // AWS CONNECTION
    #pragma omp section
    {
        #pragma omp critical
        cout << "* Starting AWS IoT communication in Thread: " << omp_get_thread_num() << endl;

        // open the connection to AWS IoT (MQTT)
        // event based messages will be returned in the treadSafeQueued
        this_thread::sleep_for(chrono::seconds(1));
        Communication communication = Communication(
                threadSafeQueue,
                connection,
                clientId,
                topic
```

```
    );

    while (true) {
        // block until AWS/Update message received
        string message = threadSafeQueue.front();
        threadSafeQueue.pop();

        // publish this vehicle model to AWS - triggered from VehicleControl
        if (strcmp(message.c_str(), "update") == 0) {
            // convert the vehicle model json to a string
            StringBuffer buffer;
            Writer<StringBuffer> writer(buffer);
            #pragma omp critical
            vehicleModel.Accept(writer);

            // publish to the AWS channel
            communication.publish(buffer.GetString());
            continue;
        }

        // process AWS messages from other vehicles

        // parse string->json
        Document doc;
        doc.Parse(message.c_str());  // other vehicles message

        // if following the truck in this message, note its position and direction
        if (following == doc["id"].GetString()) {
            vehicleModel["following-x"].SetDouble(doc["x"].GetDouble());
            vehicleModel["following-y"].SetDouble(doc["y"].GetDouble());
        }

        // add received timestamp
        // TODO: Using server time here is preferred as it will cause critical issues if a machines time is wrong
        auto timestamp = chrono::duration_cast<chrono::seconds>(
                chrono::system_clock::now().time_since_epoch()).count();
        doc.AddMember("received", timestamp, doc.GetAllocator());

        // existing - update the record
        bool updated = false;
        for (auto &vehicle: vehicles) {
            if (strcmp(vehicle["id"].GetString(), doc["id"].GetString()) == 0) {
                vehicle.Swap(doc);
                updated = true;
                break;
            }
        }
        // new - add it
        if (!updated) {
            cout << "+ " << doc["id"].GetString() << endl;
            vehicles.push_back(std::move(doc));
        }
    }
}

// USER INTERFACE
#pragma omp section
{
    #pragma omp critical
    cout << "* Starting User Interface in Thread: " << omp_get_thread_num() << endl;

    this_thread::sleep_for(chrono::seconds(3));
```

```
cout << endl << "Press J <enter> to join, or L <enter> to leave" << endl;
char c;
while(true) {
   c = (char) getchar();

   // request to join
   if (c == 'j' || c == 'J') {
      time_t now = time(nullptr);
      uint32_t unix_timestamp = static_cast<uint32_t>(now);

      if (vehicleModel["joined"].GetUint() == 0) {
         #pragma omp critical
         {
            String lastJoinedId = getLastVehicleId(vehicles);

            if (lastJoinedId == "") {
               // nothing found, you're the leader!
               cout << "You're the platoon leader" << endl;
               following = "";
            }
            else {
               // vehicle found, follow it
               following = lastJoinedId;
               for (auto &doc : vehicles) {
                  if (doc["id"].GetString() == following) {
                     vehicleModel["x"].SetDouble(doc["x"].GetDouble());
                     vehicleModel["y"].SetDouble(doc["y"].GetDouble());
                     break;
                  }
               }
               cout << "following: " << following << endl;
            }

            vehicleModel["joined"].SetUint(unix_timestamp);

            cout << "Platoon joined" << endl;
         }
      }
      else {
         #pragma omp critical
         cout << "Already in a platoon" << endl;
      }
   }

   // leave
   if (c == 'l' || c == 'L') {
      if (vehicleModel["joined"].GetUint() == 0) {
         #pragma omp critical
         cout << "Not in a platoon" << endl;
      }
      else {
         #pragma omp critical
         {
            vehicleModel["joined"].SetUint(0);
            vehicleModel["x"].SetDouble(250);
            vehicleModel["y"].SetDouble(350);
            vehicleModel["following-x"].SetDouble(0);
            vehicleModel["following-y"].SetDouble(0);
            following = "";
            cout << "Platoon left" << endl;
         }
      }
```

```
            }
          }
        }
    }
}
```

2) Functional programming Python

```
import command_line_utils
import json
import sys
import threading
from awscrt import mqtt
from tkinter import *

WIDTH = 980
HEIGHT = 800

truck_colors = {
    1: 'black',
    2: 'red',
    3: 'blue',
    4: 'green',
    5: 'magenta',
    6: 'yellow'
}

win = Tk()
win.title("Truck Canvas")
win.geometry("%sx%s" % (WIDTH, HEIGHT))
canvas = Canvas(win, width=WIDTH, height=HEIGHT, background="white")

image = PhotoImage(file="images/truck01.png")
resized_image = image.subsample(4, 4)  # larger number is smaller
canvas.create_image(100, 100, anchor=CENTER, image=resized_image, tags="truck01")

canvas.coords("truck01", -200, -200)

image2 = PhotoImage(file="images/truck02.png")
resized_image2 = image2.subsample(4, 4)  # larger number is smaller
canvas.create_image(100, 100, anchor=CENTER, image=resized_image2, tags="truck02")
canvas.coords("truck02", -200, -200)

image3 = PhotoImage(file="images/truck03.png")
resized_image3 = image3.subsample(4, 4)  # larger number is smaller
canvas.create_image(100, 100, anchor=CENTER, image=resized_image3, tags="truck03")
canvas.coords("truck03", -200, -200)

image4 = PhotoImage(file="images/truck04.png")
resized_image4 = image4.subsample(4, 4)  # larger number is smaller
canvas.create_image(100, 100, anchor=CENTER, image=resized_image4, tags="truck04")
canvas.coords("truck04", -200, -200)

image5 = PhotoImage(file="images/truck05.png")
resized_image5 = image5.subsample(4, 4)  # larger number is smaller
canvas.create_image(100, 100, anchor=CENTER, image=resized_image5, tags="truck05")
canvas.coords("truck05", -200, -200)

image6 = PhotoImage(file="images/truck06.png")
resized_image6 = image6.subsample(4, 4)  # larger number is smaller
canvas.create_image(100, 100, anchor=CENTER, image=resized_image6, tags="truck06")
canvas.coords("truck06", -200, -200)
```

```
canvas.pack()

trucks_dicts = {}

received_count = 0
cmdUtils = command_line_utils.CommandLineUtils("PubSub - Send and recieve messages through an MQTT connection.")
cmdUtils.add_common_mqtt_commands()
cmdUtils.add_common_topic_message_commands()
cmdUtils.add_common_proxy_commands()
cmdUtils.add_common_logging_commands()
cmdUtils.register_command("endpoint", "<str>", "endpoint", True, str)
cmdUtils.register_command("key", "<path>", "Path to your key in PEM format.", True, str)
cmdUtils.register_command("cert", "<path>", "Path to your client certificate in PEM format.", True, str)
cmdUtils.register_command("client_id", "<str>", "Client ID to use for MQTT connection (optional, default='test-*').")
cmdUtils.register_command("topic", "platoon/channel", "topic", True, str)
cmdUtils.register_command("port", "<int>", "Connection port. AWS IoT supports 443 and 8883 (optional, default=auto).",
                          type=int)

cmdUtils.get_args()
received_all_event = threading.Event()


# Callback when connection is accidentally lost.
def on_connection_interrupted(connection, error, **kwargs):
    print("Connection interrupted. error: {}".format(error))


def on_connection_resumed(connection, return_code, session_present, **kwargs):
    print("Connection resumed. return_code: {} session_present: {}".format(return_code, session_present))

    if return_code == mqtt.ConnectReturnCode.ACCEPTED and not session_present:
        print("Session did not persist. Resubscribing to existing topics...")
        resubscribe_future, _ = connection.resubscribe_existing_topics()

        resubscribe_future.add_done_callback(on_resubscribe_complete)


def on_resubscribe_complete(resubscribe_future):
    resubscribe_results = resubscribe_future.result()
    print("Resubscribe results: {}".format(resubscribe_results))

    for topic, qos in resubscribe_results['topics']:
        if qos is None:
            sys.exit("Server rejected resubscribe to topic: {}".format(topic))


# Callback when the subscribed topic receives a message
def on_message_received(topic, payload, dup, qos, retain, **kwargs):
    data = json.loads(payload.decode('utf-8'))
    if not trucks_dicts.get(data['id']):
        trucks_dicts[data['id']] = {'num': len(trucks_dicts) % 6 + 1}

    y_start = (trucks_dicts[data['id']]['num'] - 1) * 160
    y_end = trucks_dicts[data['id']]['num'] * 160

    color = truck_colors[trucks_dicts[data["id"]]["num"]]

    canvas.create_rectangle(600, y_start, 980, y_end, outline="black", fill=f"{color}", width=2)

    text = f"""
        Truck ID: {data['id']}
        "x": {data["x"]},
```

```
        "y": {data["y"]},
        "isBraking": {data["is-braking"]},
        "speed": {data['speed']},
        "direction": {data['direction']},
        "joined": {'Yes' if data["joined"] else 'No'},
        "following-x": {data['following-x']},
        "following-y": {data['following-y']},
    """
    canvas.create_text(750, y_start + 80, text=text, fill='white', tags="main_text")

    canvas.coords(f"truck0{trucks_dicts[data['id']]['num']}", data["x"], data["y"])


if __name__ == '__main__':
    mqtt_connection = cmdUtils.build_mqtt_connection(on_connection_interrupted, on_connection_resumed)
    connect_future = mqtt_connection.connect()

    connect_future.result()
    print("Connected!")

    message_topic = cmdUtils.get_command(cmdUtils.m_cmd_topic)
    print("Subscribing to topic '{}'...".format(message_topic))
    subscribe_future, packet_id = mqtt_connection.subscribe(
        topic=message_topic,
        qos=mqtt.QoS.AT_LEAST_ONCE,
        callback=on_message_received)
    subscribe_result = subscribe_future.result()
    print("Subscribed with {}".format(str(subscribe_result['qos'])))

    win.mainloop()
```