Truck Platooning

Efficient And Real-Time Comminication For Autonomous Truck Platooning Using Distributed Systems

Phuc Le Quang
Faculty of Computer Sciences
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
quangphuc.engineer@gmail.com

Quyen Ho
Faculty of Computer Sciences
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
quyen.holethuc@gmail.com

Dao Nguyen
Faculty of Computer Sciences
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
nvdao2k@gmail.com

Stevenson Issac
Faculty of Computer Sciences
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
stevensonetumudon@gmail.com

***Abstract*— This project implements a distributed system for truck platooning using TCP/IP for real-time communication and multithreading in C++ for concurrency in monitoring, event handling, and communication. The state machine based control models ensure system could work as defined mechanisms and easy to management. The system supports platoon formation, communication between trucks, obstacle detection and emergency braking, validated through simulation.**

Keywords—truck platooning, distributed systems, embedded systems engineering, TCP/IP, state machines.

I. MOTIVATION

A truck platoon consists of a group of trucks that are virtually connected and travel closely together using automated driving technology (Bhoopalam et al., 2017). Platooning represents a significant step toward smarter, safer, and more efficient transportation systems. The advantages of truck platooning include reduced costs, lower emissions, and increased fuel efficiency.

The truck platoon relies on embedded systems and distributed computing to enable safe, efficient and intelligent system. These technologies provide real-time control, communication, fault tolerance and security, making autonomous platooning a promising solution for future transportation.

This project aims to design and implement truck platoon using TCP/UDP communication and multithreading in C++. A hybrid TCP/UDP approach facilitates data exchange by striking optimal balance between reliability and speed, while multithreading ensures process real-time data without delays, improving responsiveness. Each truck operates as a separate thread, handling sensor input, control decisions and communication independently.

II. REQUIREMENTS

A. Functional requirements

- Leader Election: The system shall include a mechanism for leader election in a platoon.
- Emergency Handling: The system shall be able to receive emergency signal via UDP and TCP, and act upon the command.
- Truck Communication: The system shall be able to communicate via UDP and TCP.
- Security: The system shall have a secure communication channel to prevent unauthorized use.
- Safe Distance Maintenance: The system shall maintain a safe distance of at least 30 meters between trucks.
- Obstacle Avoidance: The system shall respond to obstacles within 2 milisecond.
- Clock Synchronization: The system shall synchronize clocks of all trucks in the platoon.
- Obstacle Detection: The system shall detect obstacles within a 50-meter range.

B. Non-functional requirements

- Communication Latency: The system shall communicate between trucks with a latency of less than 100 msec.
- Logging and Error Analysis: The system shall log all interactions and errors for future analysis.
- Adaptivity: The system could work with many types of road and traffic conditions.
- Speed Adjustment: The system shall adjust speed based on weather, road conditions and instruction from leading truck.
- Dynamic Addition/Removal: The system shall allow dynamic addition/removal of trucks from the platoon.

III. SKETCH OF APPROACH

A. Leader-Follower Model (Client-Server Architecture)

The system uses a leader-follower model where the leading truck (server) manages high-level control tasks like speed adjustment, braking, and platoon coordination. The following trucks (clients) synchronize their actions with the leader using real-time updates over networked communication channels. Leader-follower control structures are essential for coordinating multiple vehicles in a platoon, ensuring they maintain safe distances while following the leader's commands (Zhang et al., 2021). This architecture enhances system reliability by allowing automatic recovery mechanisms if the leader fails.

B. Approaches to Distributed Communication

Several communication methods were considered to enable real-time data sharing between trucks, each with their use cases, strengths, and limitations.

TABLE I. COMPARISON OF POSSIBLE APPROACHES

Approach	Use Case	Strengths	Weaknesses
DDS (Data Distribution Service)	Vehicle-to-vehicle (V2V) communication, event propagation across platoons.	Built-in **publish-subscribe model**, QoS guarantees.	High system overhead and **complexity in configuration**
Socket Programming (TCP/UDP)	Low-level, high-performance communication between trucks.	Low overhead and **customizable protocol handling**. Great for broadcast activities	Manual error handling required for fault tolerance.
HTTP/REST APIs	Non-critical updates and remote monitoring services.	**Easy integration** with external systems.	**High latency**, not suited for real-time processes

For the platooning system, socket programming (TCP and UDP) was selected due to its flexibility, minimal overhead, and low latency requirements. TCP ensures reliable, ordered delivery of critical messages like braking commands, while UDP is lightweight and useful for fast, periodic updates where occasional packet loss is acceptable (Deyah, and Bhaya, 2017).

C. Method of Implementation

The communication architecture leverages socket programmming as follows:

- TCP: Used for critical messages, such as braking commands or leader updates.
- UDP: Used for non-critical, low-latency updates, such as location and speed broadcasts.

IV. System Architecture

A. Block diagram

This block diagram as shown in Fig. 2 represents the overall architecture of the Truck Platooning System, showing its core components and external entities. Key blocks such as the Emergency Handler, Truck Manager, and Obstacle Processor interact with the Control Logic Mechanism, Safety Mechanism, and Communication Layer for coordinated platoon operations. External interactions, including data logging, event handling, and message broadcasting, ensure system robustness, fault tolerance, and real-time updates across distributed nodes.

B. Context diagram

As shown in Fig. 3, the context diagram shows the interaction between internal system components and external entities within the Truck Platooning System. The Leading Truck manages platoon coordination through the Platoon Server TCP, with the Communication Layer, Truck Manager, and Event Handler facilitating command distribution and emergency handling. The Following Truck includes the Platoon Client TCP to ensure synchronized following behavior. External services like Logging Services and Data Storage interact seamlessly to provide real-time data logging

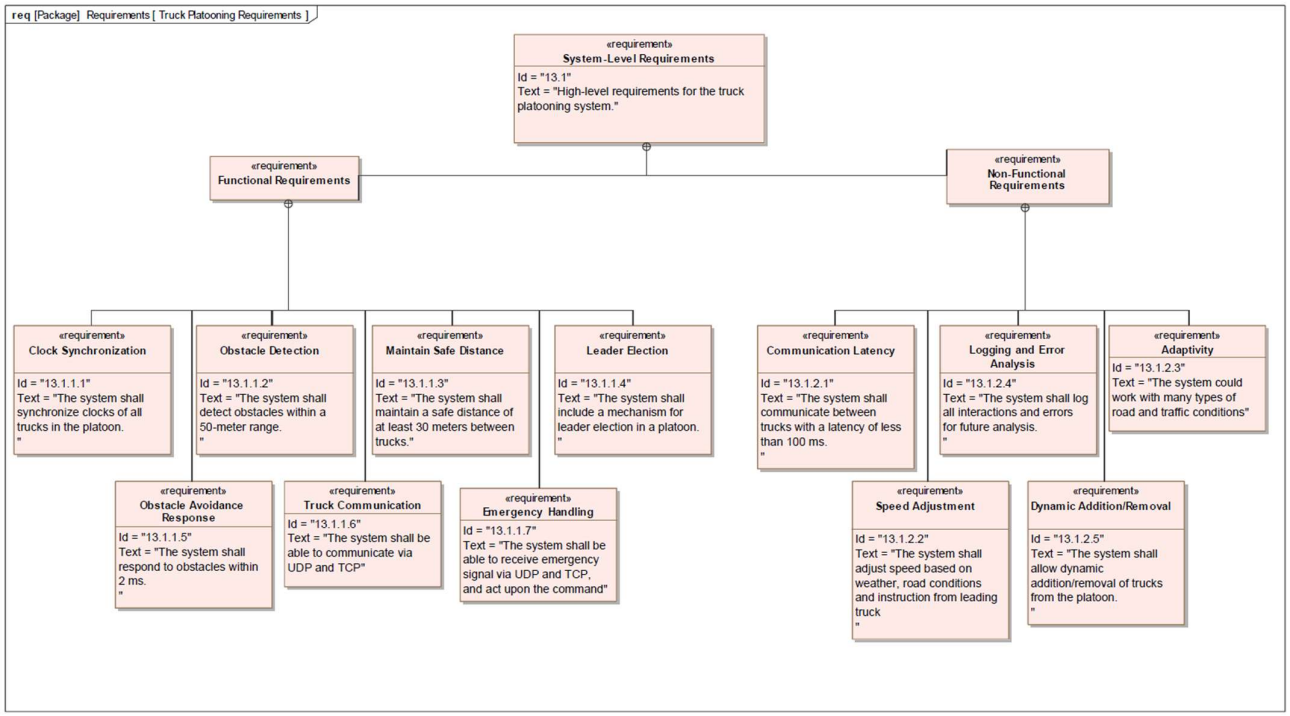

Fig. 1. Requirement diagram for Truck Platooning System

and system monitoring. The design ensures dynamic scalability and fault tolerance through well-defined interfaces and communication flows.

C. Sequence diagram

This part considers describing the main interactions within the system under joining platoon and sending status use cases. In the first one, joining platoon (Fig. 4), five objects/actors interact with each other. This case starts when one truck sends auth request to the leading truck. Once the leading truck has received the request which includes an authenticated key, it forwards the key to Authentication Module and sends back the result to the truck. The truck then continues to send join request, and the leading truck will pass the request to Message Handler. Truck Manager will add truck information to the system database. After that, the leading truck returns join accepted command along with synchronized data. Finally, the truck can join the platooning system.

If the truck has joined the system successfully, it is required to send its status to the leading truck periodically. Fig. 9 shows main interactions for sending status. The member truck transmits information such as position (via GPS location), distance with the front/back trucks and its current speed. When the leading truck receives data, State Manager will process the data, retrieve data from Sensor Processor and adjust speed to maintain safety distance. The final step is following truck receives synchronized data, allowing the entire system to operate in synchronization.

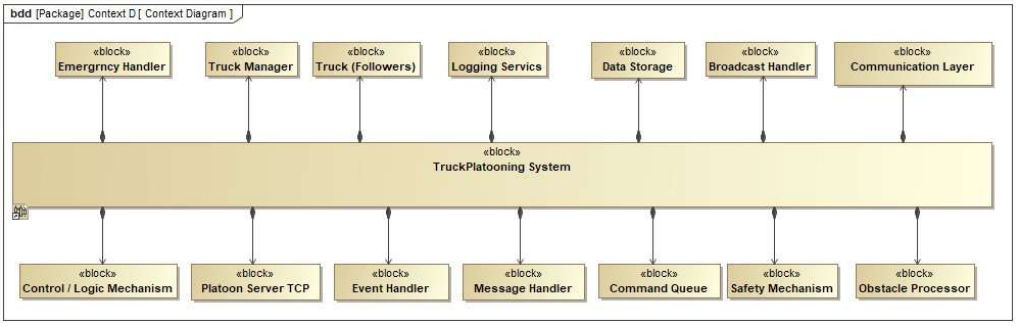

Fig. 2. Block diagram for Truck Platooning System

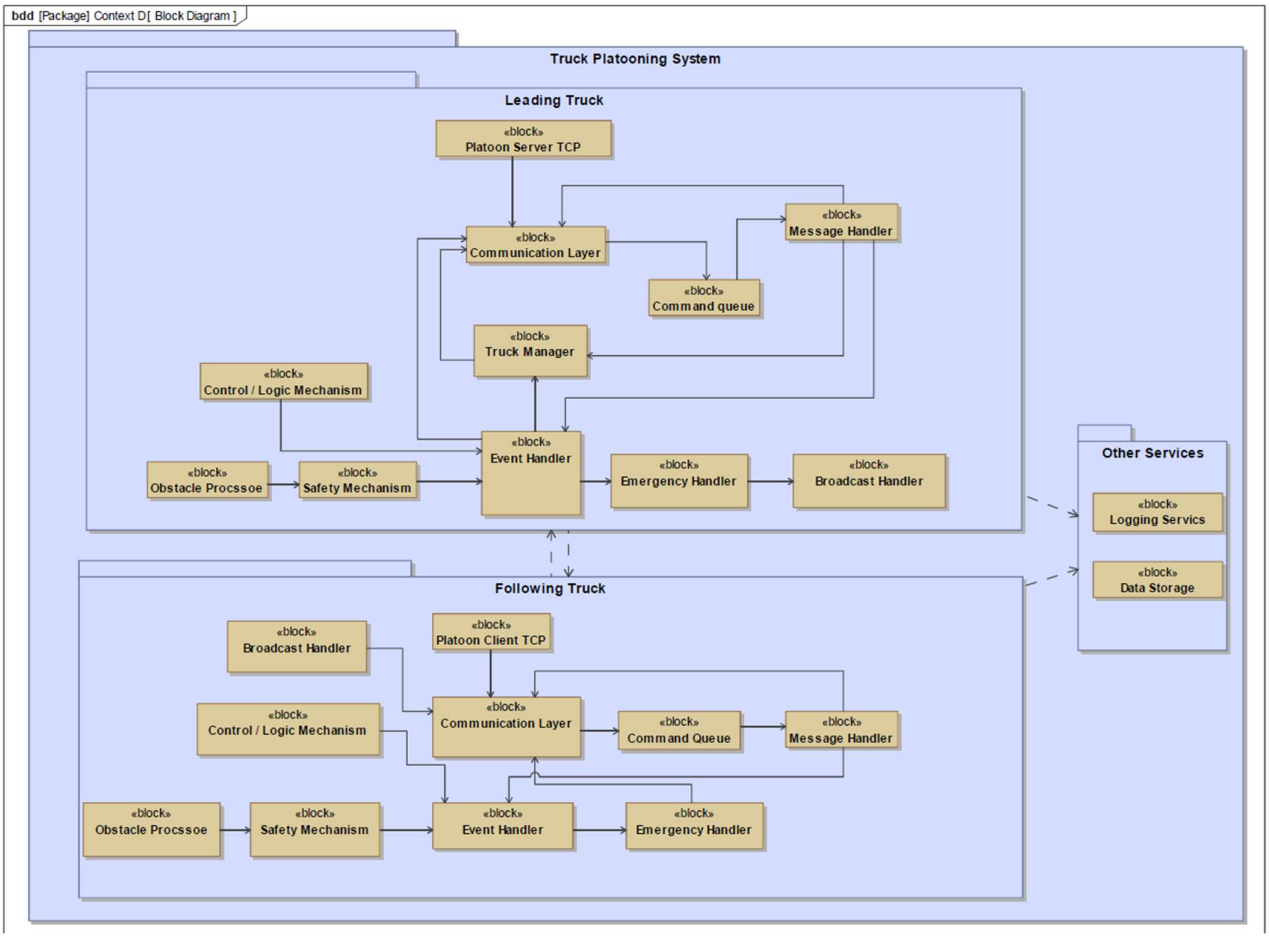

Fig. 3. Context diagram for Truck Platooning System

D. Constraints

As shown in Fig. 5 - 8, the parametric diagram highlights the integration of critical constraints across the system modules for the Truck Platooning System. It links essential parameters such as Safe Following Distance, Platoon Stability, Obstacle Avoidance, and Communication Latency to their corresponding constraint blocks, ensuring that each module adheres to the defined safety and performance requirements. The Control Unit enforces minimum stability and safe distance thresholds, while the Obstacle Detection Module dynamically evaluates proximity to obstacles. Synchronization constraints ensure time-sensitive operations maintain system-wide coherence via logical clock updates, enhancing distributed control and fault tolerance.

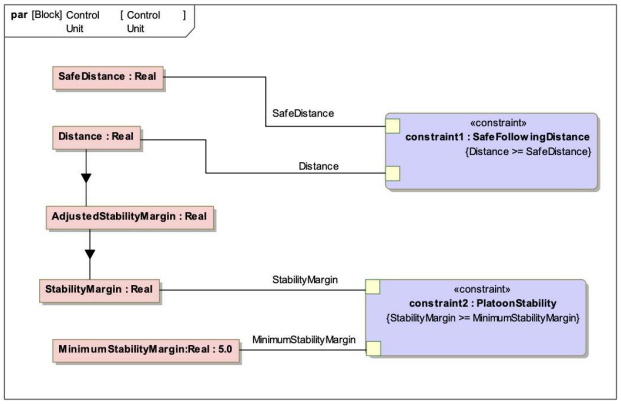

Fig. 4. Parametric Constraints diagram for Control Unit

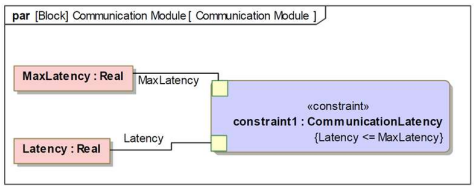

Fig. 7. Parametric Constraints diagram for Communication Latency

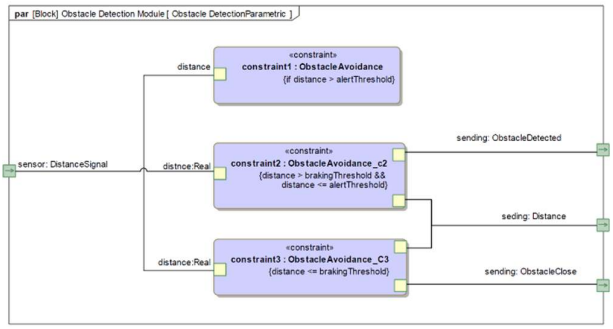

Fig. 5. Parametric Constraints diagram for Obstacle Detection

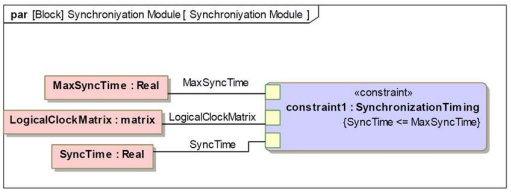

Fig. 8. Parametric Constraints diagram for Synchronization

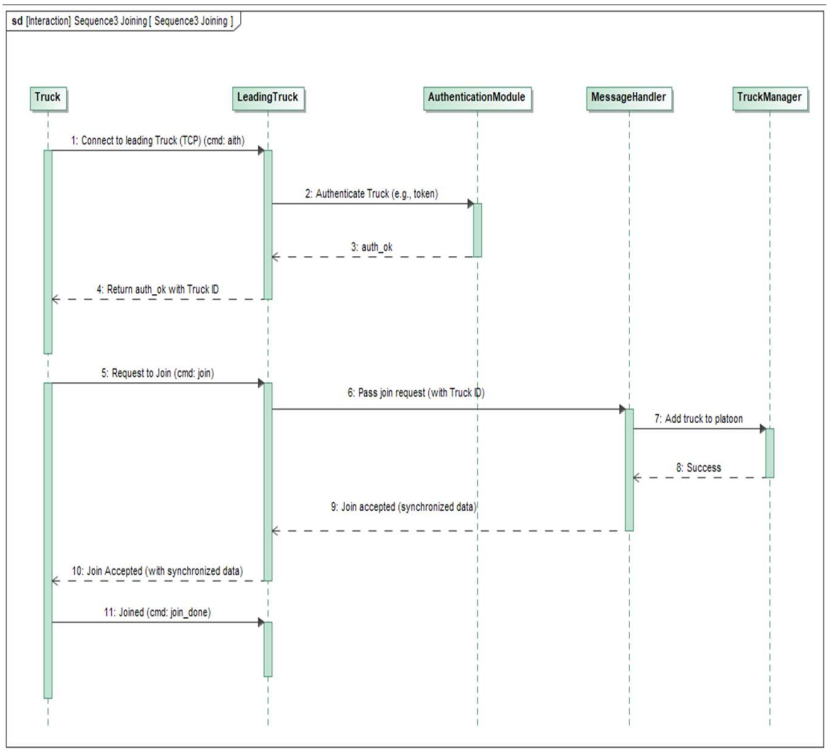

Fig. 6. Sequence diagram of main interaction for joining platoon

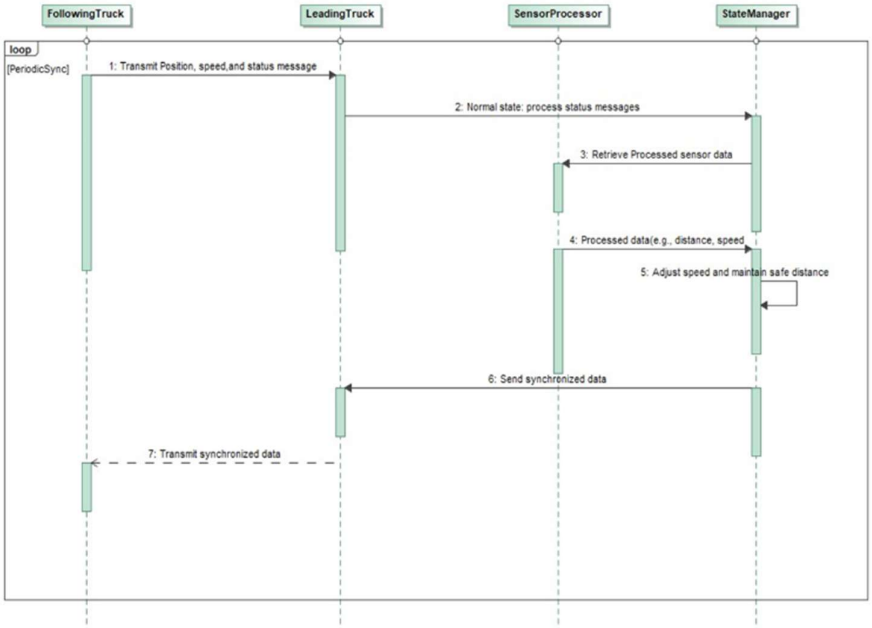

Fig. 9. Sequence diagram of main interaction for sending status

E. Internal blocks

The Internal Block Diagram (IBD) shown in Fig. 10 represents the **Truck Platooning System**, detailing the flow of information and interactions among its key components and external data sources. The **Obstacle Processor** receives real-time obstacle data through its **SensorInput**, triggering alerts when necessary, propagating to the **Safety Mechanism** for emergency response. The **Control Logic** manages decision-making, including speed adjustments and platoon coordination, based on inputs from the **Driver Interface** and **Traffic Data** through the **Communication Handler**. Synchronization signals from the **Clock Synchronizer** ensure consistent, synchronized operations across the platoon. The diagram effectively models the continuous flow and coordination necessary to maintain safe and efficient platoon operations.

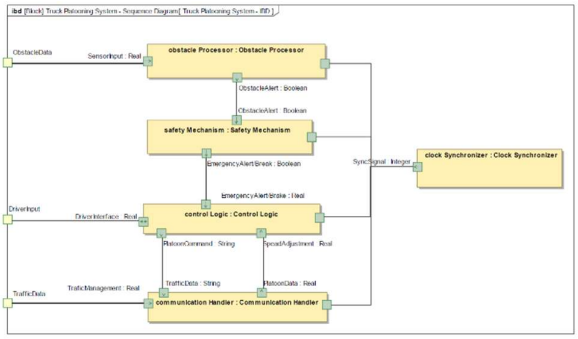

Fig. 10. Internal Block Diagram (IBD) Truck Platooning System

F. System state machines

a) Implementation of a specific state machine

In this part, state machine of obstacle avoidance block which is shown in Fig. 11 below is based on to implement on Arduino board.

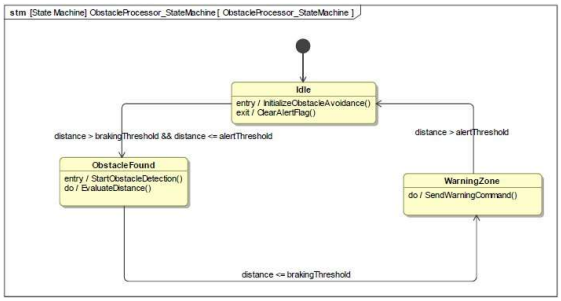

Fig. 11. State machine of obstacle avoidance

The Fig. 11 illustrates that there are three states (IDLE, ObstacleFound and WarningZone) and two thresholds (brakingThreshold and alertThreshold). If the distance is greater than alertThreshold, the system is in IDLE state. If the distance is less than alertThreshold and greater than brakingThreshold, the system is in ObstacleFound state. If the distance is less than brakingThreshold, the system is in WarningZone and sends out a warning message. Next, the connection for Arduino implementation needs to be completed.

From Fig. 12, there are an ultrasonic sensor for collecting the distance data and three LEDs for representing three states (blue for IDLE, yellow for ObstacleFound and red for WaringZone).

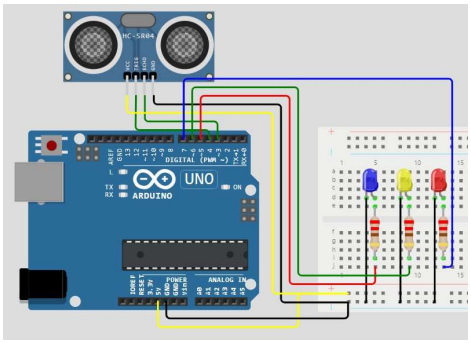

Fig. 12. Arduino connection

b) Scheduling

In this project, GPU Tesla T4 from Google Colab is used to implement the obstacle avoidance block which contains two tasks. Hence, the information of this GPU can be used to specify some parameters for scheduling constraints such as WCET, period, deadlines, priority, and resources that are listed in TABLE II.

TABLE II. SCHEDULING CONSTRAINTS

Task	receive Data	monitorDistance
WCET	338.97us	336.28us
Period	2 ms	2 ms
Deadline	1.5 ms	2 ms
Priority	High	High
Resource	GPU	GPU

WCET is the worst case for the execution time of a task. Google Colab provides a command "nvprof" to get WCET. From TABLE II, there are two tasks that are receiveData with WCET of 338.97us and monitorDistance with WCET of 336.28us. Moreover, the requirements that are defined at the beginning are combined with information in TABLE II to obtain period and deadline parameters. In this case, the requirement defines the response time as 2 ms. Therefore, the periods and the deadlines must be less than or equal to 2 ms. Because the deadline is a crucial element that needs to be considered for obstacle avoidance function, EDF (Earliest Deadline First) is applied to schedule tasks in this component.

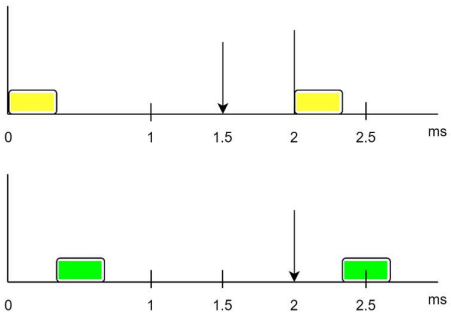

Fig. 13. EDF for obstacle avoidance

In Fig. 13, the task represented by yellow color is receiveData with deadline of 1.5 ms and the task represented by green color is monitorDistance with deadline of 2 ms. Because 1.5 is less than 2, the priority of yellow task is higher than the green task. To determine schedulability of this component, the formula (1) below is used when applying EDF:

$$Schedulability = \sum \frac{Ci}{Ti} \qquad (1)$$

If the schedulability is less than or equal to 1, this set of tasks is schedulable. If this schedulability is greater than 1, this set of tasks is not schedulable. In this case, the parameters from TABLE II are substituted into schedulability formula, and the result is 0.338 which is less than 1. Therefore, the obstacle avoidance component is schedulable.

V. IMPLEMENTATION

A. Identify data, signal, event

TABLE III. TABLE OF DATA, SIGNAL, EVENT OF SYSTEM

Signal/Data	Event
- Truck ID - Acceleration signal - Obstacle detection signal - Braking pressure signal - Steering angle/turning signal - Gear position (N,P,D,R) - The distance between trucks (GPS signal) - Running/Operation time - Speed signal - Fuel level signal - Traffic signal/ road condition ahead	- Authentication - Join/ join accepted/ join done - Leave/ leave start/ leave done - Status - Emergency - Obstacle - Synchronize

B. Implemented use cases

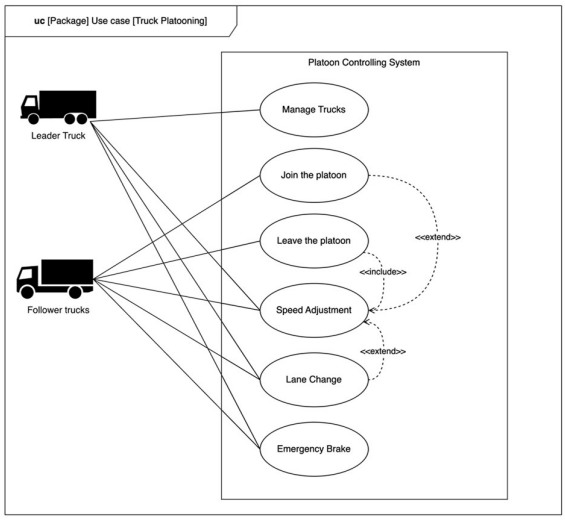

Fig. 14. Implemented use cases

The implementation consists of the interaction between leader truck and follower truck within several use cases such as "Manage Truck", "Join the platoon", "Leave the platoon" and "Emergency Brake". It focuses on simulating specific use cases to indicate the embedded and distributed system characteristics in truck platoon.

C. System state machines

a) Leading truck

State machine of application tasks which handle all events of system such as initialization, idle, normal operation, emergency operation, lost connection and invalid tasks

operation. Additionally, the leading truck has also two more state machines for message server operation and truck management. (See Fig. 18)

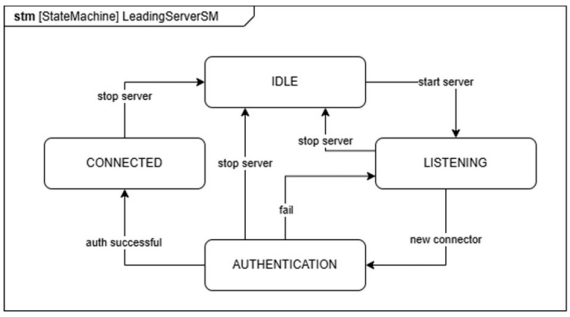

Fig. 17. State machine of communication server

State machine of truck management (Fig. 19) is in charge of managing trucks information in the platoon such as add truck identity to storage, remove trucks from list, provide platoon status for other services. The Fig. 17 shows the state machine of communication server which is considered as a connector to other trucks, it provides protocol to send and receive messages from trucks via TCP port and a specific IP address.

b) Following truck

The state machine indicates the current state of following truck, handles actions and performs control corresponding to each state, as shown in Fig.16.

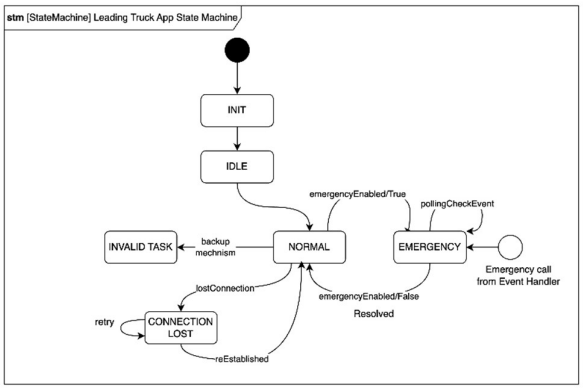

Fig. 18. State Machine of main application of leading truck

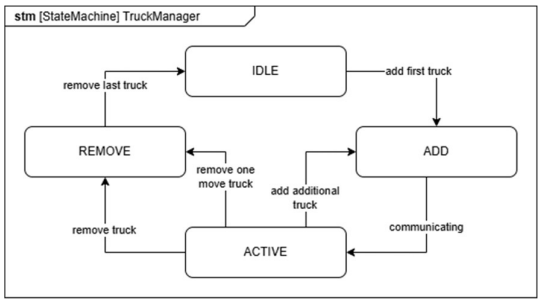

Fig. 19. State machine of truck management

There are eight states in the state machine, and six of them, joining, normal_operation, emergency_brake, speed_up, slow_down and leaving, are when the truck is in the platooning system and maintains its connection with leading (server) truck. Idle state indicates that the truck is not in the platooning and can send request to join, while connection_lost shows that the truck have already joined but failed to connect to leading truck afterward. The truck will return to the previous state when the connection is re-connected again. During normal_operation state, the following performs some tasks in parallel using different threads: listen and receive messages via TCP/UDP from the leading truck, control speed and maintain safety distance or detect and send alert messages when obstacle appears.

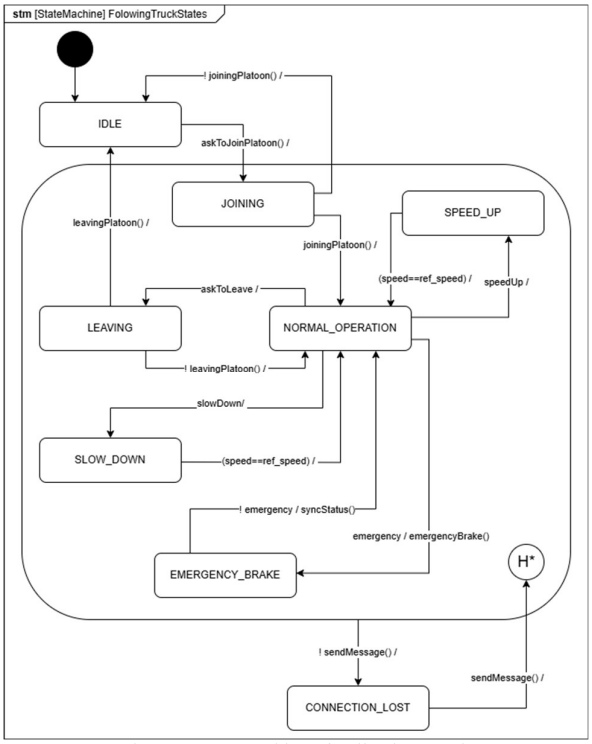

Fig. 16. State Machine of Following Truck

D. Use Case and Activity diagrams

a) Use Case Diagram

The Use Case diagram shown in Fig. 22 outlines interactions between the main external actors (Central Server, Driver, Truck, and Environment) and critical system use cases such as Obstacle Detection, Maintain Safe Distance, Emergency Handling, and others. It provides a high-level overview of key system functionalities and interactions. Furthermore, three Activity Diagrams were generated from the Use Case as shown in Fig. 20, 21, & 23.

b) Obstacle Detection Activity Diagram

- Start Point: System initializes by reading environmental parameters.
- Decision Point: If an obstacle is detected, the distance is evaluated to determine if it's within the safe limit.

- Actions: If unsafe, the system triggers the avoidance mechanism (e.g., braking); otherwise, it sends an update message for monitoring.
- End Point: Completes once the appropriate action is taken.

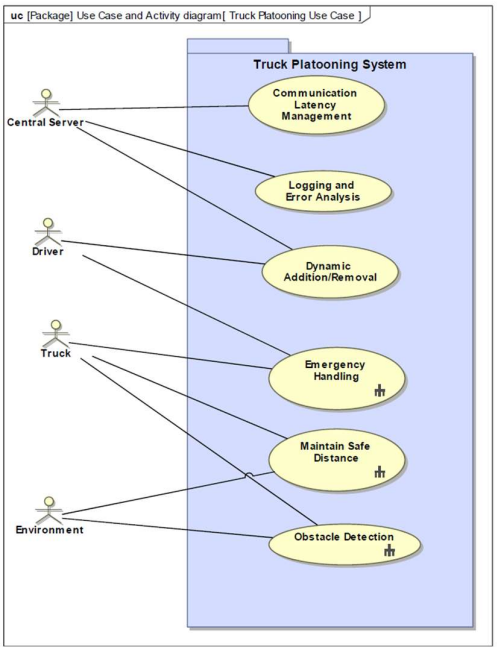

Fig. 22. Use Case diagram for the internal and external system

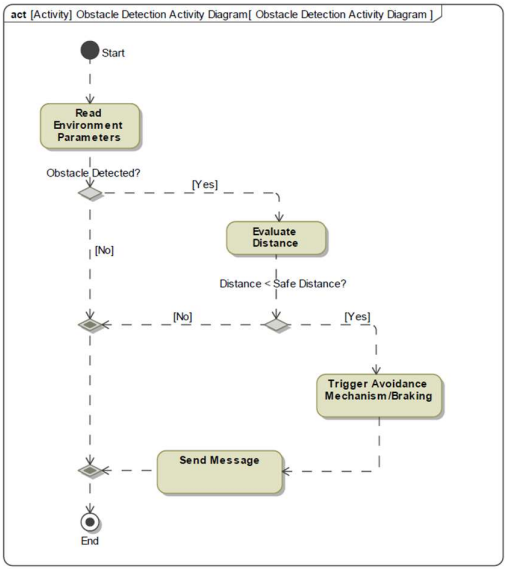

Fig. 23. Obstacle Detection Activity Diagram

c) Maintain Safe Distance Activity Diagram

- **Start Point**: Receives real-time distance information from sensors.
- **Calculations**: Determines whether the current distance is within the safe threshold.
- **Decision Point**: If the distance is too short, the speed is adjusted by decelerating the truck.
- **Synchronization**: Updates the truck's position and sends synchronized data to maintain platoon

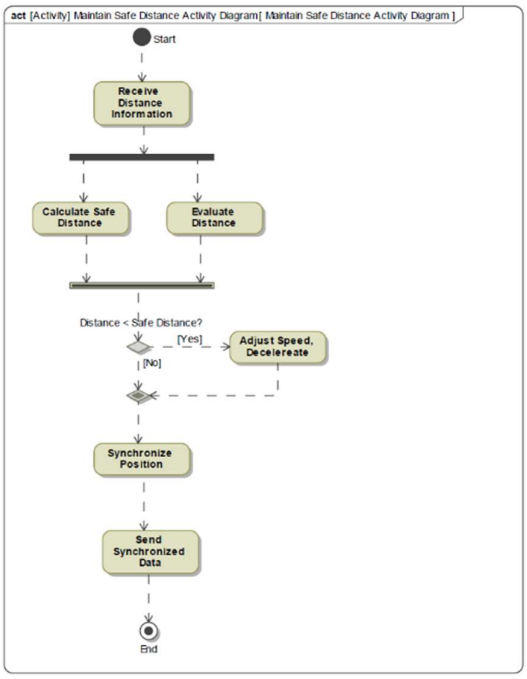

Fig. 21. Maintain Safe Distance Activity Diagram

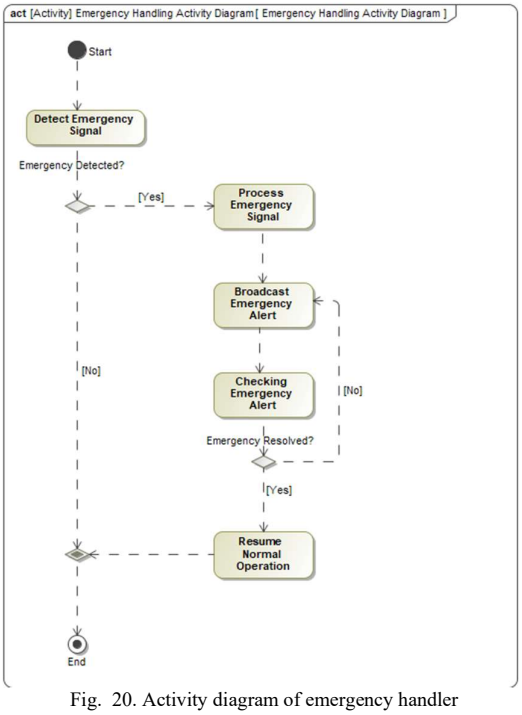

Fig. 20. Activity diagram of emergency handler

stability

d) Leading truck

When leading truck receives emergency signal from message handler or detected by itself, the system operation switches to emergency operation and perform a set of actions such as

broadcast alert via UDP, polling check emergency events, send emergency instructions to following truck via TCP.

e) Following truck

This part focuses on three activity diagrams for following truck namely joining the platoon (Fig. 24), obstacle detection/ avoidance (Fig. 25) and emergency handling (Fig. 26).

When the truck wants to join the platooning system, it must first send the authentication and joining request to the leading truck. If the leading truck accepts the joining request, it will send back the sync data. The following truck must then extract that data and perform calculations to synchronize with the platooning system and join into the system.

Assume that the truck receives distance data from sensor which illustrates three levels of obstacle distance (safe distance, detection distance and warning zone). If the signal indicates warning zone, the truck must then send alert message to leading truck.

Whenever the following truck receives emergency signal via UDP or TCP, it will move to emergency state and handle calculations for controlling speed as well as listen to message to check emergency status.

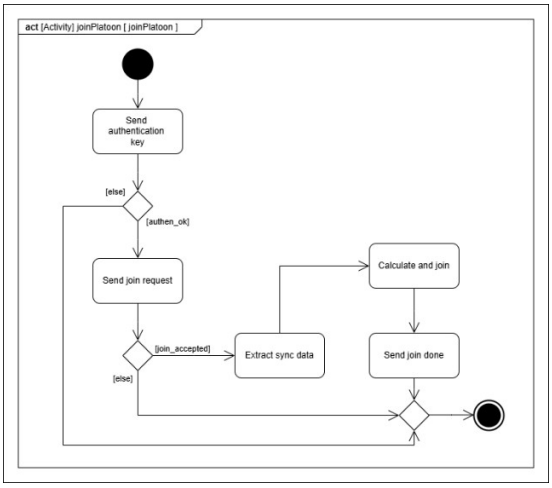

Fig. 24. Activity diagram of joining the platoon for following truck

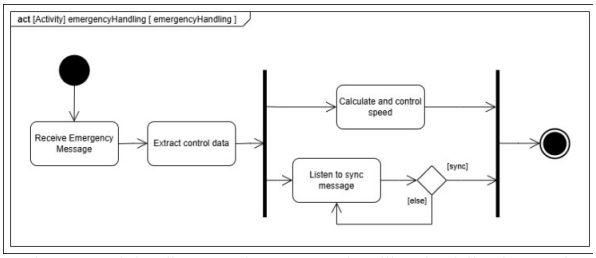

Fig. 25. Activity diagram of emergency handling for following truck

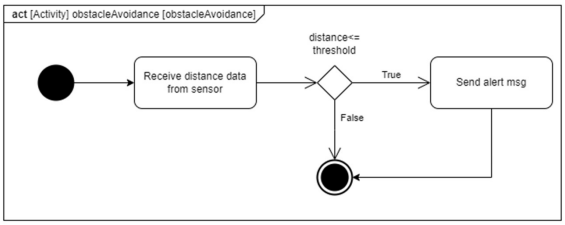

Fig. 26. Activity diagram of obstacle detection/avoidance

E. GPU implementation

- **Parallel Programming Models**

To handle the high computational demand of sensor fusion, obstacle detection, and inter-truck synchronization, the following parallel programming models (shown in TABLE IV) were studied and compared for best application in the system:

- **Implementation of Parallelism**

To handle compute-intensive tasks like obstacle detection and real-time sensor processing, the system uses CUDA for GPU-based parallelism. According to Sanders and Kandrot (2010), GPUs are optimized for data-parallel tasks that require high throughput, such as image processing or matrix computations. This allows thousands of threads to run concurrently, ensuring efficient handling of large datasets.

- **Hardware Choice**

- Multi-Core CPU:
  Intel or AMD processors were used for handling control, communication, and OpenMP-based parallel tasks.
- NVIDIA GPU:
  For real-time image processing, matrix multiplication, and obstacle detection, NVIDIA GPUs were selected due to their compatibility with CUDA and OpenCL. This choice aligns with the choice of parallelism. Sanders and Kandrot laid some emphasis on NVIDIA GPUs, that it provide high computational throughput for real-time applications involving large datasets.

TABLE IV. PROGRAMMING MODELS

Programming Model	Purpose/Features	Use Case Fit
CUDA (GPU Programming)	- High throughput for data-parallel tasks - Uses GPU hardware - Ideal for tasks requiring large-scale parallelism	Suitable for real-time obstacle detection, logical matrix clock synchronization, and simulations involving multiple processes running concurrently.
OpenMP (Multithreading)	- Shared memory model - Uses multi-core CPUs - Good for medium-level parallelism and simpler synchronization	Suitable for prototyping but limited scalability compared to GPUs.
Threads/Task-Based Models (e.g., Pthreads, TBB)	- Low-level thread control - Flexibility in defining how tasks are split and synchronized - Good for CPU-based concurrency	Limited scalability, suitable for small systems with fewer nodes or simpler simulation.

a) Information and architecture of GPU

In this project, the obstacle avoidance component is selected for GPU implementation. Firstly, a suitable programming model must be chosen to utilize GPU power. There are many programming models worldwide such as CUDA, OpenMP, and Pthreads. TABLE IV shows comparisons of features and use cases between these programming models to find the most suitable model.

From TABLE IV, the CUDA programming is considered as the most useful for our project because the set of tasks in the obstacle avoidance component should be run parallelly, and the project may be expanded, hence, many different functions or tasks will be created in the future. With CUDA programming and GPU, tasks can be run on different threads, and cores parallelly.

Secondly, GPU Tesla T4 on Google Colab is used to simulate the obstacle avoidance function. To be able to use this GPU, some information and structure of this GPU must be understood

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   40C    P8              9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Fig. 27. GPU information

The Fig 27. shows that name of GPU is Tesla T4, this GPU has 40 cores or streaming multiprocessors (SMs), CUDA version is 12.2, and the global memory is about 16 GB. The connection between global memory and 40 cores is shown in Fig. 28.

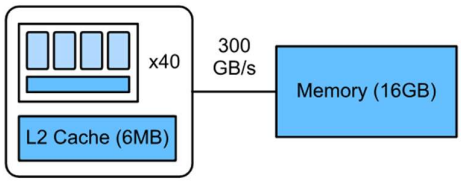

Fig. 28. High-level block diagram of GPU [6]

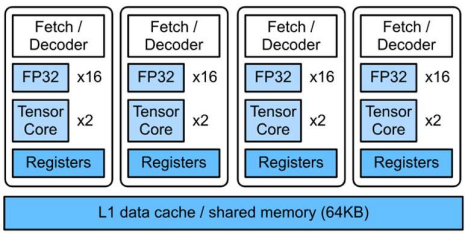

Fig. 29. SM Architecture [6]

This GPU consists of 40 SMs and each SM is similar to a mini processor. 6MB L2 Cache is a shared medium-speed cache used to reduce frequent access to global memory. The architecture of a SM is shown in the Fig. 29.

The Fig. 29 illustrates that each SM contains 64 CUDA cores that execute 32-bit floating-point operations (FP32), and 64KB L1 data cache is a shared memory between these CUDA cores. Generally, this GPU has 40 SMs (mini processors), and each SM consists of 64 CUDA cores. In CUDA programming, a GPU has many grids or kernels (SMs), each grid is a collection of thread blocks (CUDA cores), and each thread block is a group of threads.

b) Implementation

In this simulation, there are two tasks that are executed in two different kernels. These two tasks can run parallelly, and they are independent of each other.

At the beginning of the simulation, two variables are defined for detecting and warning obstacles (alertThreshold = 50 and brakingThreshold = 15) because the requirements defined that obstacles are detected within 50 m and if the distance between trucks and obstacles is less than 15 m, trucks will send warning messages. Then, two kernels are created for two tasks which are receiveData() and monitorDistance(). The first task is used to receive distance data from sensors and the second task is used to determine whether the distance between trucks and obstacles is in dangerous case or not. Next, two streams are declared and created. The two kernels are then assigned to these two streams. Because a stream acts as a mini-processor, each kernel is executed in a separate mini-processor. Therefore, the two tasks can be executed independently and parallelly.

For kernel setting, one thread block is established for each kernel and one thread is created for each block. Then, receiveData() task and monitorDistance() task are assigned to stream 1 and stream 2 respectively. “For” loop is used to simulate the distance data from sensors.

c) Clock logical matrix

In GPU implementation, because there are two tasks and two different threads in two different kernels, the logical matrix must have a size of 2x2.

This matrix is initialized by setting diagonal to 1 and rest to 0, that means the initial clock tick of task 1 in thread 1 is 1 and in thread 2 is 0 because task 1 is only executed on thread 1. In the same way, the initial clock tick of task 2 in thread 2 is 1 and in thread 1 is 0.

In each task in a kernel, there is an event that is used to update the clock tick for each task executed by that kernel

When task 1 is executed on thread 1, the clock tick is updated and increased by 1 for task 1 on thread 1. But clock tick for task 1 on thread 2 cannot be increased because task 1 is only executed on thread 1. Similarly, when task 2 is executed on thread 2, the clock tick is updated and increased by 1 for task 2 on thread 2. But clock tick for task 2 on thread 1 cannot be increased because task 2 is only executed on thread 2. Finally, the matrix is synchronized at the final execution. The clock logical matrix for obstacle avoidance is shown in Fig. 30 below.

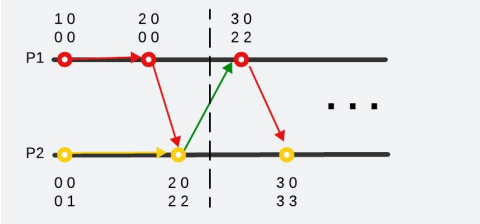

Fig.30. Clock logical matrix

VI. VALIDATION

A. Unit test

TABLE V. TABLE OF UNIT TEST CASES

Application	Test case	Summary
Leading truck	Parse Payload Test	Validates correct extraction of values (e.g., location, speed, status) from a JSON payload in the TruckMessage class.
	Generate UUID Test	Ensures the TruckMessage::generateUUID() method produces unique, 36-character UUIDs.
	Generate Truck ID Test	Confirms the PlatoonServer generates unique, non-empty Truck IDs with the correct prefix “TR00”.
Following Truck	Initialize Truck Data Test	Checks data initialization of following truck.
	Extract Command Data	Checks that commands (e.g., emergency, speed up, slow down) are extract and pass to truck data correctly.
	Control Truck Speed Test	Ensures that speed is controlled properly when having braking.

B. Defect test

TABLE VI. TABLE OF DEFECT TEST CASES

Application	Test case	Summary
Leading truck	Invalid Authentication Key Test	Verifies that the PlatoonServer rejects trucks attempting authentication with invalid keys.
	Duplicate Truck ID Test	Ensures the TruckManager prevents overwriting an existing truck entry when a duplicate Truck ID is added.
	Invalid Command in Truck Event FSM Test	Confirms the TruckEventFSM properly handles invalid commands by throwing appropriate exceptions.
Following Truck	Wrong Data Format Test	Ensures that JSON exception will be thrown if JSON data passing to FollowingTruck::processCommand() method is in wrong format.
	Send Request When No Connection Test	Verifies that data/request can not be sent to platoon server when there is no available socket from server.

C. Component test

TABLE VII. TABLE OF COMPONENT TEST CASES

Application	Test case	Summary
Leading truck	Authentication and Joining	Verify the PlatoonServer can authenticate a truck and add it to the platoon.
	Truck Manager - Add and Remove Truck	Verify the TruckManager block correctly handles the addition and removal of trucks in the platoon.
Following Truck	Connection With Platoon Server Test	Ensures that platoon client connects to server when socket is available.

D. Test result

TABLE VIII. TEST RESULT

```
Running main() from /tmp/googletest-20240910-4595-
56gqyq/googletest-1.15.2/googletest/src/gtest_main.cc
[==========] Running 15 tests from 11 test suites.
[----------] Global test environment set-up.
[----------] 2 tests from TruckMessageTest
[ RUN      ] TruckMessageTest.ParsePayloadAndGetValues
[       OK ] TruckMessageTest.ParsePayloadAndGetValues (0 ms)
[ RUN      ] TruckMessageTest.GenerateUUID
[       OK ] TruckMessageTest.GenerateUUID (0 ms)
[----------] 2 tests from TruckMessageTest (0 ms total)

[----------] 2 tests from PlatoonServerTest
[ RUN      ] PlatoonServerTest.GenerateTruckID
[       OK ] PlatoonServerTest.GenerateTruckID (0 ms)
[ RUN      ] PlatoonServerTest.AuthenticateAndJoinTruck
[       OK ] PlatoonServerTest.AuthenticateAndJoinTruck (3 ms)
[----------] 2 tests from PlatoonServerTest (3 ms total)

[----------] 1 test from AuthenticationDefectTest
[ RUN      ] AuthenticationDefectTest.InvalidKey
[       OK ] AuthenticationDefectTest.InvalidKey (0 ms)
[----------] 1 test from AuthenticationDefectTest (0 ms total)

[----------] 1 test from TruckManagerDefectTest
[ RUN      ] TruckManagerDefectTest.DuplicateTruckID
[       OK ] TruckManagerDefectTest.DuplicateTruckID (0 ms)
[----------] 1 test from TruckManagerDefectTest (0 ms total)

[----------] 1 test from TruckEventFSMTest
[ RUN      ] TruckEventFSMTest.HandleInvalidCommand
[       OK ] TruckEventFSMTest.HandleInvalidCommand (0 ms)
[----------] 1 test from TruckEventFSMTest (0 ms total)

[----------] 1 test from TruckManagerTest
[ RUN      ] TruckManagerTest.AddAndRemoveTruck
[       OK ] TruckManagerTest.AddAndRemoveTruck (0 ms)
[----------] 1 test from TruckManagerTest (0 ms total)

[----------] 1 test from AppStateMachineTest
[ RUN      ] AppStateMachineTest.SwitchToNormalOperation
[       OK ] AppStateMachineTest.SwitchToNormalOperation (0 ms)
[----------] 1 test from AppStateMachineTest (0 ms total)

[----------] 3 tests from FollowingTruckTest
[ RUN      ] FollowingTruckTest.GetCommandFromLeading
[       OK ] FollowingTruckTest.GetCommandFromLeading (0 ms)
[ RUN      ] FollowingTruckTest.TruckInit
[       OK ] FollowingTruckTest.TruckInit (0 ms)
[ RUN      ] FollowingTruckTest.TruckSpeedControlWhenBraking
[       OK ] FollowingTruckTest.TruckSpeedControlWhenBraking(0 ms)
[----------] 3 tests from FollowingTruckTest (0 ms total)

[----------] 1 test from CommandDefectTest
[ RUN      ] CommandDefectTest.CommandParseInvalid
[       OK ] CommandDefectTest.CommandParseInvalid (0 ms)
[----------] 1 test from CommandDefectTest (0 ms total)

[----------] 1 test from FollowingDefectTest
[ RUN      ] FollowingDefectTest.SendRequestWhenNoConnection
[       OK ]FollowingDefectTest.SendRequestWhenNoConnection(3 ms)
[----------] 1 test from FollowingDefectTest (3 ms total)

[----------] 1 test from PlatoonConnectionTest
[ RUN      ] PlatoonConnectionTest.CreateSocket
[       OK ] PlatoonConnectionTest.CreateSocket (0 ms)
[----------] 1 test from PlatoonConnectionTest (0 ms total)

[----------] Global test environment tear-down
[==========] 15 tests from 11 test suites ran. (8 ms total)
[  PASSED  ] 15 tests.
```

VII. INSPECTION

A. Phuc inspects Quyen's implementation

The client program is developed in well-structured design which is easy to investigate, validate and enhance in future. The state machine and activity diagram for following trucks are clear and straightforward, providing a solid foundation for system behavior.

Points could be updated in the next version: The UDP listening services on client side should operate independently to prevent blocking in case of a UDP initialization failure. The message reception mechanism should be handled with a queue of messages instead of waiting for a specific command name.

B. Quyen inspects Phuc's implementation

The server program for leading truck is designed and implemented in a strong object-oriented programming (OOP) approach which is well-organized and easy to understand. Each state machine for the leading truck implementation is handled for a specific purpose, providing flexibility within the system.

Points could be updated in the next version: On the leading truck side, it should provide more specific notifications, such as instructions to speed up or slow down, to the following truck. This will enhance the system's robustness, rather than simply sending emergency and synchronization data.

C. Dao inspects Stevenson's implementation

The diagrams of the system are drawn expertly by using an appropriate tool. Internal block diagrams, component diagrams, constraint diagrams, and general state machines are designed logically and easily to investigate and implement the system or components in code.

Points could be updated in the next version: The requirements should be more specified. Especially, the priority of each task should be defined clearly for scheduling effectively because more functions or tasks will be built in the future, and it may be difficult to synchronize when implementing concurrent and parallel tasks.

D. Stevenson inspects Dao's implementation

The sub-task (Obstacle Detection and Avoidance) was properly implemented. The use of GPU for the task implementation was properly handled, especially by using kernels for distinct functions. This allows the use of GPU advantages in parallel programming. The proper programming model and hardware (CUDA and NVIDIA) was also employed by using the Google COLAB environment for evaluation. The hardware simulation was properly carried out by employing Tinkercad simulation tools.

Points could be updated in the next version: The handling of Obstacles should be done in 2-dimension; The system should be able to handle obstacles getting too close from all directions (e.g., front and side). Also, simulation should be carried out in a proper Embedded System environment (e.g., by choosing appropriate micro-controller).

SUMMARY AND OUTLOOK

A. Summary

The system achieved designed architectures, met requirements and characteristics of an embedded system which is combined with distributed and parallel system. The truck platooning project is likely to work as expected such as join, leave convoy, broadcast emergency via TCP and UDP as a backup channel. The system could communicate between objects via TCP protocol with predefined data frames in JSON format.

B. Challenges and Outlook

The current implementation successfully establishes a functional and robust system. However, there are several enhancements and future directions could be taken into account:

- Enhanced communication protocol with security: Implement high-level authentication mechanism in communication between trucks such as encrypt, decrypt message, authentication key,
- Advanced fault recovery mechanisms: Adapting to self-healing mechanisms that could improve system resilience.
- Integration with MQTT to enhance latency and ensure connectivity: Ensuring low-latency, high-reliability message between trucks. Synchronizing messages across multiple protocols.
- Integration with 5G and cloud-based management: The system could benefit from 5G communication, which offers ultra-low latency and high bandwidth connectivity, enabling cloud-based computing for distributed processing closer to the trucks.

APPENDIX

Appendix A – Source Code

a) Distributed and Parallel Systems (DPS):

Link: https://github.com/thisisquangphuc/fhdo-distributed-system/tree/master

b) Embedded Software Engineering (ESE):

Link: https://github.com/thisisquangphuc/fhdo-winter24-ese

Appendix B – GitHub Overview

a) Repository: **fhdo-distributed-system**

Total: 47 files, 3713 codes, 706 comments, 693 blanks, all 5112 lines.

TABLE IX. PROJECT OVERVIEW

Path	Files	Line of code
.	47	3,713
. (Files)	3	115
gpu	5	1,124
src	36	2,224
src (Files)	7	494
src/communication	8	507
src/control	8	873
src/libs	2	168
src/libs/dotenv	2	168
src/utils	11	182
tests	3	250

b) Repository: **fhdo-winter24-ese**
Total: 2 files, 97 codes, 8 comments, 12 blanks, all 117 lines.

TABLE X. ESE SOURCE CODE OVERVIEW

Path	Files	Line of code
.	2	97
. (Files)	1	14
src	1	83

Appendix C – Contribution

TABLE XI. TABLE OF CONTRIBUTION

Contributor	Task
Phuc Le Quang	Manage GitHub repository, branches, perform merging and pull requests from branches. *Distributed and Parallel System:* Design system architecture, identify data, signal, events are required for the interaction between trucks. Design state machines, activity diagrams for model-based specification Simulation – program server side, leading truck. *Embedded Systems Engineering:* Specification of the analysis model with context diagram, block diagram and describe main interactions using sequence diagram for leading truck. Implement test driven based development in validation phase; design and run test cases include unit test, defect test, component test.
Quyen Ho	*Distributed and Parallel System:* Identify data, signal, events are required for the interaction between trucks. Simulation – program client side, following truck. *Embedded Systems Engineering:* Define state machines, activity diagram for following truck. Implement test driven based development in validation phase; design and run test cases include unit test, defect test, component test.
Dao Nguyen	*Distributed and Parallel System:* Compare programming models for GPU implementation Implement GPUs program which demonstrates parts of the algorithm. Implement clock synchronization by logical matric clock. *Embedded Systems Engineering:* Show partly the implementation level of one block including the state machine behavior using Arduino simulation. Specify scheduling constraints, computation time, implement tasks scheduling.
Stevenson Issac	*Distributed and Parallel System:* Develop appropriate distributed and parallel architecture with requirements and characteristics. Compare implementation approaches, programming models. Refine method of implementation, hardware. Investigating GPU implementation. *Embedded Systems Engineering:* Applying SysML application to draw diagrams of system. Refining the architecture with internal block diagrams, component diagrams, constraint diagrams, general state machine for discussed approach.

TABLE XII. NUMBER OF COMMITS OF MEMEBERS

Contributor	Commits
Phuc Le Quang	37 (DPS), 4(ESE)
Quyen Ho	36 (DPS)
Dao Nguyen	5 (DPS), 1 (ESE)
Stevenson Issac	2 (DPS)

Appendix D – Folder Structure

TABLE XIII. PROJECT SOURCE CODE STRUCTURE

```
├── CMakeLists.txt
├── README.md
├── build
├── include
├── src
│   ├── app.cc
│   ├── app.h
│   ├── communication
│   │   ├── comm_init.cpp
│   │   ├── comm_manager.cpp
│   │   ├── comm_manager.h
│   │   ├── comm_msg.h
│   │   ├── platoon_client.cpp
│   │   ├── platoon_client.h
│   │   ├── platoon_server.cpp
│   │   └── platoon_server.h
│   ├── control
│   │   ├── event_manager.cpp
│   │   ├── event_manager.h
│   │   ├── following_truck.cpp
│   │   ├── following_truck.h
│   │   ├── monitor.cpp
│   │   ├── monitor.h
│   │   ├── trucks_manager.cpp
│   │   └── trucks_manager.h
│   ├── error.h
│   ├── error_handling.cpp
│   ├── following.cpp
│   ├── libs
│   │   └── dotenv
│   │       ├── LICENSE
│   │       ├── dotenv.c
│   │       └── dotenv.h
│   ├── main.cpp
│   ├── notes.txt
│   └── utils
│       ├── config.cpp
│       ├── config.h
│       ├── env.h
│       ├── json.h
│       ├── keygen.h
│       ├── logger.cpp
│       ├── logger.h
│       ├── tcp_utils.cpp
│       ├── tcp_utils.h
│       ├── utils.cpp
│       └── utils.h
├── tests
│   ├── CMakeLists.txt
│   ├── build
│   ├── command_test.cpp
│   └── following_test.cpp
└── workflow.txt
```

AFFIDAVIT

We (Phuc Le Quang, Quyen Ho, Dao Nguyen, Steveson Issac) herewith declare that we have composed the present paper and work ourselves and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the

same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.

ACKNOWLEDGMENT

The authors would like to thank Prof. Stefan Henkler for valuable guidance throughout the project. We also appreciate the support of Fachhochschule Dortmund for providing computational resources.

REFERENCE

[1] Bhoopalam, A. K., Agatz, N., & Zuidwijk, R., 2017. *Planning of truck platoons: A literature review and directions for future research*. Transportation Research Part B Methodological (107), pp. 212–228.

[2] Zhang, Y., Wang, Y., Wang, J., & Chen, W., 2021. *Leader-Follower Coordination for Vehicular Platoons: A Distributed Control Perspective.* International Journal of Control, pp. 320-335.

[3] Deyah, W. and Bhaya, W., 2017. *Balancing between TCP and UDP to Improve Network Performance*. ARPN Journal of Engineering and Applied Sciences 12 (22), pp 6410-6415.

[4] Sanders, J., & Kandrot, E., 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional.

[5] Mu Li, 2020. GPU Architecture: gitHub repository. Accessed: Feb. 03. 2025. [Online]. Available: https://github.com/d2l-ai/d2l-tvm-colab/blob/master/chapter_gpu_schedules/arch.ipynb

[6] Mu Li, 2020. chapter_gpu_schedules: gitHub repository. Accessed: Feb. 03. 2025. [Online]. Available: https://github.com/d2l-ai/d2l-tvm-colab/blob/master/chapter_gpu_schedules/arch.ipynb