Truck Platooning System

Formation of truck platoon

1st Sheikh Muhammad Adib Bin Sh Abu Bakar
Fachhochschule Dortmund
ME. Embedded Systems Engineering
sheikh.binshabubakar001@stud.fh-dortmund.de

2nd Luis Fernando Rodriguez Gutierrez
Fachhochschule Dortmund
ME. Embedded Systems Engineering
luis.rodriguez001@stud.fh-dortmund.de

3rd Aditya Kumar
Fachhochschule Dortmund
ME. Embedded Systems Engineering
aditya.kumar001@stud.fh.dortmund.de

4th Leander Hackmann
Fachhochschule Dortmund
ME. Embedded Systems Engineering
leander.hackmann001@stud.fh-dortmund.de

5th Mykyta Konakh
Fachhochschule Dortmund
ME. Embedded Systems Engineering
mykyta.konakh001@stud.fh-dortmund.de

***Abstract*—Truck platooning significantly improves fuel efficiency through reduced aerodynamic drag, leading to cost savings and environmental benefits. Enhanced traffic flow, safety improvements, and decreased driver fatigue contribute to a more efficient and sustainable long-haul freight transportation system. This motivation leads us to design a prototype truck platooning system.In this paper we document our truck platooning system from analysis, system design , system implementation (at simulation level) to the verification and validation of the system prototype.**

***Index Terms*—truck platoning, server client communication, truck architecture**

I. Introduction

The emergence of distributed and parallel systems in the technical landscape has improved fault tolerance, efficiency, and dependability of systems, as well as tackling the issues of scalability, performance, and more [1]. One such implementation of distributed and parallel systems in the domain of mobility and logistics is truck platooning. Truck platooning is a driving technique in which a convoy of trucks travels close to one another while being linked by advanced driver assistance systems and wireless technologies as visualized in Figure 1. The convoy's lead truck, which is equipped with sensors and communication systems, sets the pace and directs its progress, while the following trucks imitate its actions, including braking and acceleration. This helps with traffic flow optimization, enhances road safety, increases fuel efficiency and cost savings, and reduces environmental impact [2].

II. Analysis

We start the project by analyzing the basic requirements for a truck platooning system. There are several factors one should consider, such as a reliable communication system to ensure smooth coordination among the vehicles in the platooning system, the ability to maintain a consistent speed and a minimum distance between vehicles to ensure safe operation, efficient fuel consumption, and the ability to adapt to changes in traffic conditions. It is important to develop a system that can accurately detect and respond to the movements of the leader and follower trucks in real-time and should be able to respond to emergencies [2].

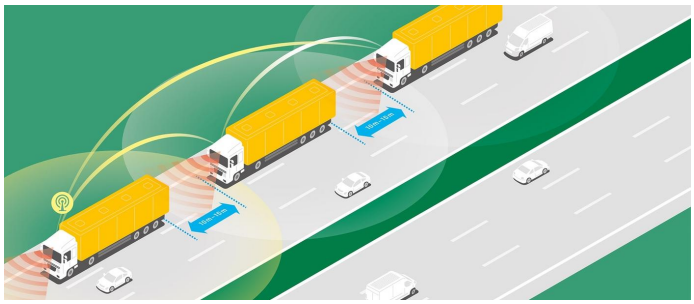

Fig. 1. Truck Platooning [3]

Thus, based on this, we can develop a use case for a truck platooning system that enhances safety and efficiency in long-haul transportation. As depicted in Figure 2, we establish two main actors that can interact with the system: the truck and the communication server. The communication server is linked to the use cases in the context of communication, whereas the truck is associated with all of the defined use cases. The system consists of the following use cases:

- Truck Network Formation
- Message Passing
- Leader Selection
- Collision Avoidance
- Move in Platoon
- Connect To Follower

Scenario for Implementation: On analysis we come up with a scenario that potentially covers all the defined use cases. We consider a situation where a platoon of trucks with a leader and followers already exists and is connected over a communication server. In case of an error in the leader truck, we want to make sure that we can assign the role of leader to the truck that is in the first position of the platoon. In the following section, we will discuss the design and implementation of the system.

Fig. 2. System Use case

III. Design and Implementation: System

The platooning system consists of mainly two components: the truck, which can either be a leader or a follower, and the communication channel. The following block definition diagram is shown in Figure 3, which represents various system components and their relationships in a truck platooning system. There are two main subsystems: the instance of the truck and the server. The truck consists of other subsystems such as the controller unit, communication unit, collision avoidance unit, and human-machine interface. This portioning allowed us to develop the system prototype more efficiently because two parts could be developed in parallel. The upcoming section provides a detailed explanation of specific parts of the system and their functionalities.

IV. Design and Implementation: Truck

The truck consists of 3 main components, Controller, Driver interface and Communication module as shown in Figure 4. We treat Truck as a whole component that will be run through an '.exe' file. All the main components of the Truck are executed as threads to ensure concurrency, this way we can have some parallelism in the implementation. This behavior is visualized in Figure 5. Those components will be explained in detail in the incoming sub-section.

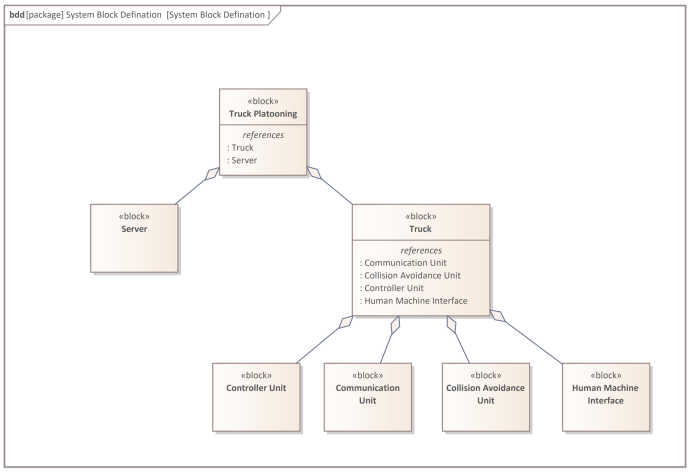

Fig. 3. System Architecture

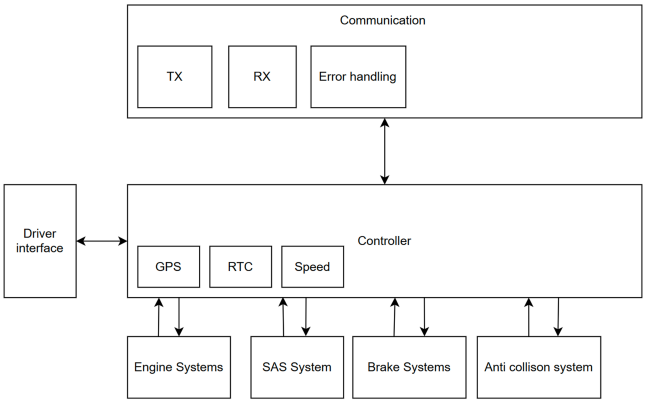

Fig. 4. Truck Architecture

A. Controller

The controller is the main component that controls the behavior of the truck, and the other component or sub system is the output or input to the controller. The behavior is modeled as in Figure 6 where it has 5 main states, Initial, waiting, leader, follower and system stop state. The state machine is implemented using switch case as in Listing 2

The reason it has leader and follower state is because the truck can change its roles while still active or moving. The leader and follower state have its own internal state as depict in Figure 7 and Figure 8, where both have a moving state, error handling state and emergency stop state and only follower state have align state because it need to keep align with its leader.

Based on Figure 6, Figure 7 and 8, most of the transition is influenced by an event or a signal. For instance, from waiting state, it will change to leader state if and only if it receives a leader signal and and it will change to follower state if and only if it receives follower signal. Those signals could be produced by the controller itself or another component within the suck such as communication module. The controller behaves differently in each state. In initial state, the controller

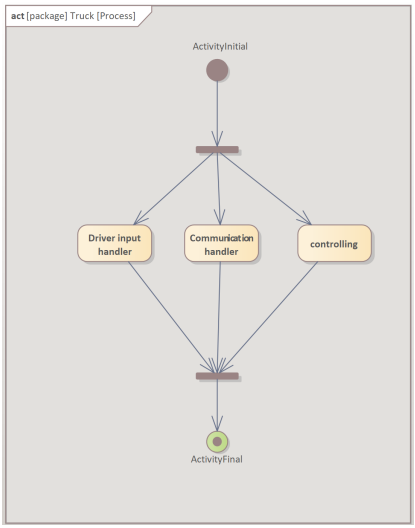

Fig. 5. Main process run by the truck

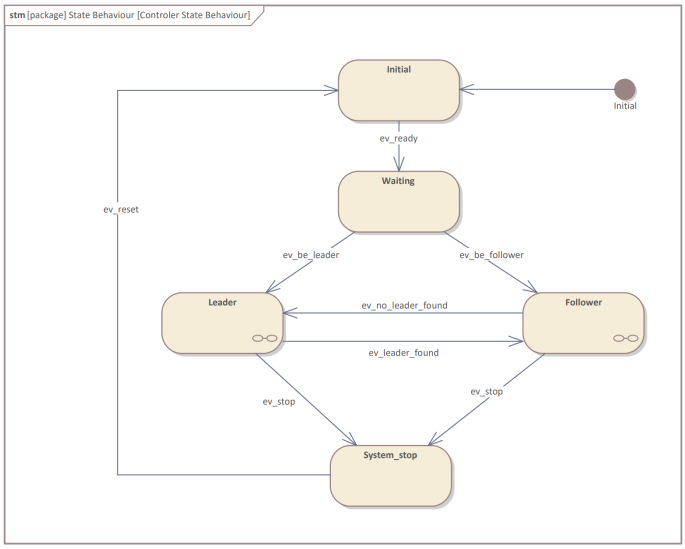

Fig. 6. Controller State Machine

will initialize all subsystems and produce a ready signal after that as visualized in Figure 9. The signal will trigger the transition from initial state to the waiting state. In a waiting state, the role of the truck will be decided by finding the leader. If a leader is found, it will set it role to follower and produce follower signal otherwise, it will be a leader and produce leader signal as shown in Figure 10. Find a leader means that the truck tries to find another nearest front truck.

The event from waiting state will decide whether it will enter the leader state or follower state. Both internal states will start with moving state as in Figure 11 and **??**. As leader, it will always send its movement set either by a driver or the system itself to the follower. Every new movement it increments its logical clock that is also shared to its follower and a routine to check for new event is executed.

As follower, it will always wait for a new movement or

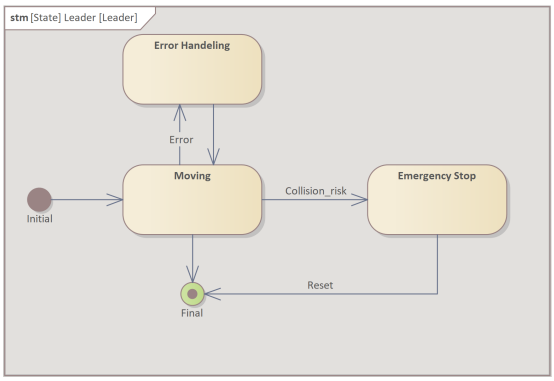

Fig. 7. Internal state machine of leader state

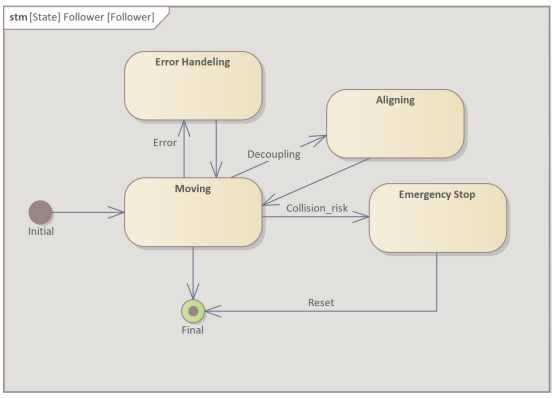

Fig. 8. Internal state machine of follower state

an event from its leader. If none of them is received within a set time out, which is handled by a watchdog timer, a no_leader_found signal will be produced. Thus, the transition from follower state to the leader state will be triggered as in Figure 6. This shows the flexibility of trucks' role. In case the stop signal is received, the controller shall change its state to system stop state. For instance, when some of the vital component is not working that lead to Hazard (not visualized in this paper). For that reason, in the stop state, the controller tries to reset the truck.

B. Driver Interface

The driver interface is the submodule that will allow us to interact with the system "Truck". This interface is a basic user input using the keys "WASD" to increase or decrease the speed, in the case of "AD" key, it is just to select the direction of the truck, like a videogame. This to make the interaction as natural as possible. For the implementation of this user input the library "conio.h" was used, this because this library has a function to threat the input of the keyboard like an interruption. It will continue checking if an input was received from the keyboard, if this was the case then we will read this input and then send the information to the controller. However, the input of the user will only be allowed in the case where the truck has the role of "LEADER" as in Figure

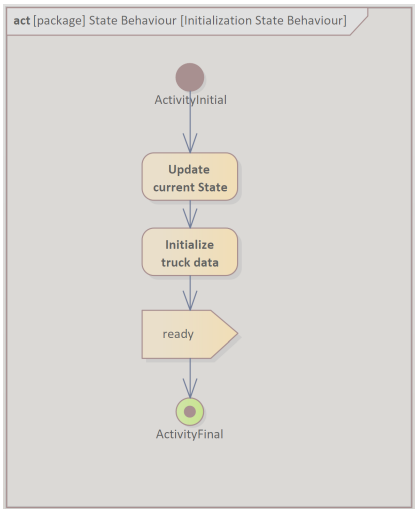

Fig. 9. Initial state behaviour

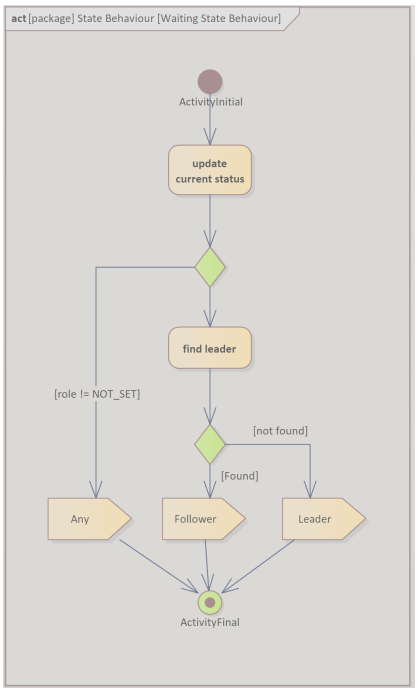

Fig. 10. Waiting state behaviour

13, in the other case, the input will simply be ignored. This behaviour is implemented as in Listing 1.

Listing 1. Driver interface only available for leader

```
if(self_truck->role == LEADER){
    inputChar = _getch();
}
else{
    inputChar;
}
```

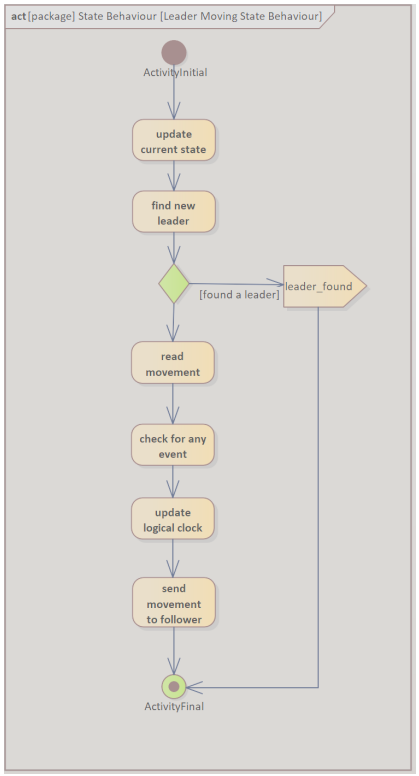

Fig. 11. leader state behaviour

C. Communication Module

The communication module is responsible for the communication between the truck and the server. This component will be explained more detailed in section V where it will acts as a client.

D. Collision Avoidance Subsystem

Another subsystem of the truck that are handled by the controller is collision avoidance. This subsystem is to prevent any Hazard due to collision between the truck and the environment. One of the functionality of the system is to measure the distance of the obstacle around the truck and the truck itself. If the distance is below a threshold, the system shall raise a warning flag. For that reasons, we named this functionality distance validation. The behaviour of this functionality is modelled as in Figure 15. There 4 distance there are measured in parallel as visualized in Figure 14. The red signal representing the distance is below threshold.

For safety reason, those distance shall validated in parallel that lead us to use GPU for parallel computation. In that case, we use Googlecolab GPU where the ”update activation value” activity from Figure 15 is the kernel. this computation consist of 3 array, sensor input, threshold and output. An example of the function use case is depicted in Figure

V. Design and Implementation: Communication

The implementation of the communication system required the selection of a suitable communication technique prior to

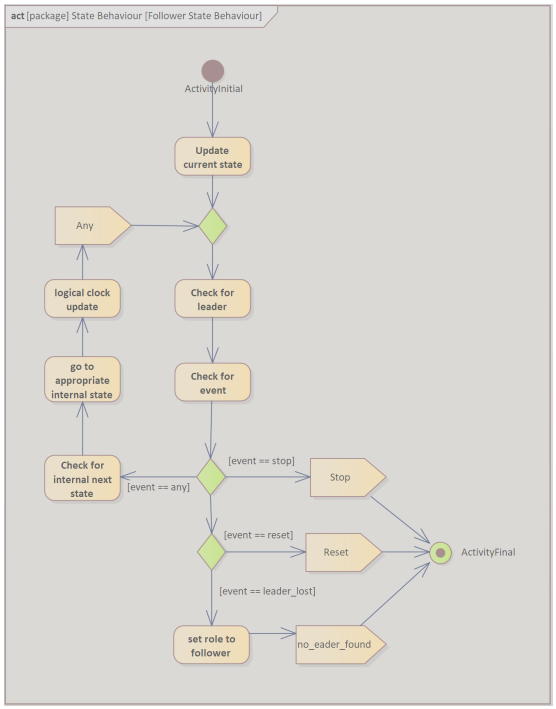

Fig. 12. Follower state behaviour

Fig. 13. Basic input and output of the controller

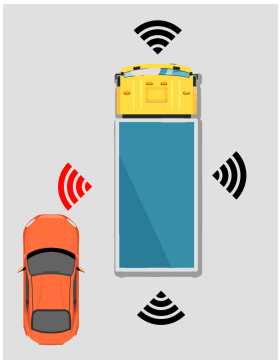

Fig. 14. Truck's sensor position

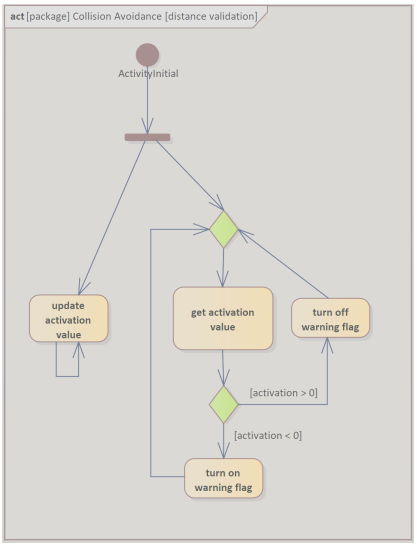

Fig. 15. Distance validation activity

implementation. Because each Truck needs to be able to communicate with every other Truck, the resulting communication scenario is a N-to-N scenario. This had a major impact on the elicitation of a suitable communication technique as the technique needs to support such a case. Since the whole project is mostly written in C++ and executed on Windows machines, the usage of a TCP/IP connection by using the winsock2.h library was an obvious choice.

The communication scenario and the working principle of the winsock2.h library had a major influence on the resulting network topology. Because of the N-to-N scenario and the lacking possibility of realizing a "discovery procedure" (initialization phase of nodes of a wireless network) with the TCP/IP protocol, a client server architecture of the network was chosen.

A. *Implementation of the server*

Because of the working principle of the server of being independent to the rest of the simulation, it was developed as an own application. After some experiments with the winsock2.h library, the needed key features of the server were found out:

- Accepting new connections.
- Receiving messages from the clients.
- Forwarding messages to their destination.
- Supplying all connected clients with an overview of all reachable clients.

With these needed features in mind, a simple activity diagram of the servers' working procedure was developed. The idea was to use a sequential way of processing the packets. After the initialization of the server and especially the socket that all clients try to connect to, it polls the socket for new incoming connections. If there is a connection available, it is accepted, and the resulting socket is stored into a vector. For this vector, the struct "SocketClientID" was created. It contains two fields:

- clientSocket [SOCKET]: used to store the socket of the client.
- ID [int]: used to associate the trucks' ID of the client to the socket.

When the connection is accepted, the value of the field "ID" is left at the default of 0 as there is at this point no information

```
Sensor position: front          back          left          right
Sensor input:    9.000000       8.000000      0.500000      6.000000
treshold:        3.000000       3.000000      0.800000      0.800000
output:          6.000000       5.000000      -0.300000     5.200000
validation:      true           true          false         true

left side is to close with obstacle !!!!
```

Fig. 16. Distance validation

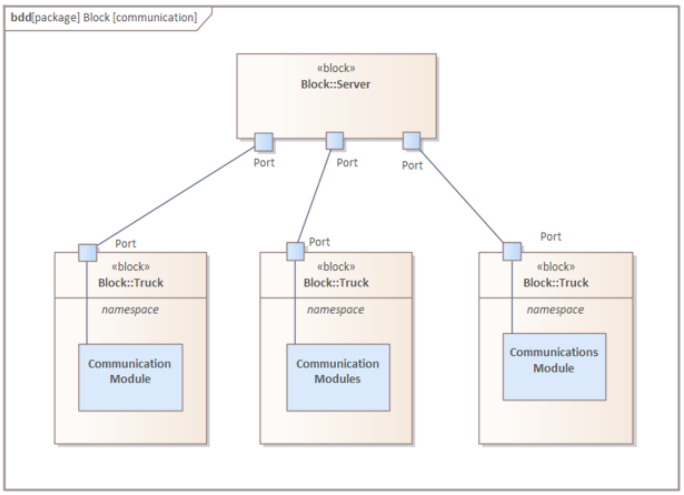

Fig. 17. Architecture of the communication system

which ID the truck has. This information gets filled in with the first received packet as the packet contains the ID of the sender.

After polling for new connections, the server checks if there is at least one active connection. If not, it continues to poll for new connections. Otherwise, the server polls all sockets of the elements of the vector for incoming packets. When there is a packet received, the field "ID" of the vector's element is checked, if there is already an associated ID. If not, the sender ID is extracted from the message and written to the field. In any case, the message is then stored into a buffer. After all messages are received and stored, the server proceeds with the forwarding of the messages. This incorporates the extraction of the destination ID from the message to search for the receivers' socket in the vector that contains the sockets of all connected clients together with their ID. If the matching socket was found, the message gets forwarded. As a last step, a message that contains all the IDs of all connected clients is created and sent to all clients.

These steps were implemented as separate functions called "initialize", "checkAndAcceptConnection", "getMessagesFromAllSockets", "forwardPackets" and "sendClientIDVector". In the main loop, they are called in this sequence to ensure the proper operation of the server. The modularity allowed a structured development and made it simple to expand the

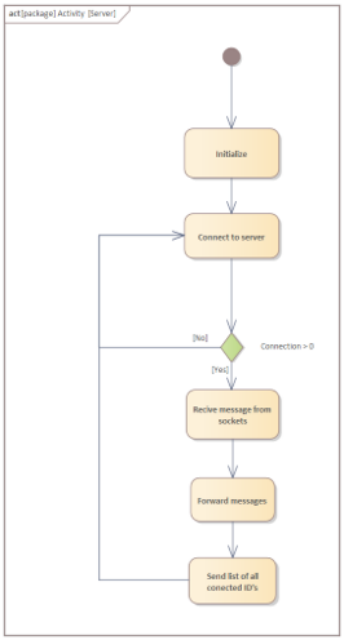

Fig. 18. Working principle of the server

functionalities. This came in hand when the feature of sending the information about all connected clients was added after the first version of the server was already developed.

B. Implementation of the client

The matching counterpart to the server, the communication module of the truck, was designed work in a similar way. Since the responsibilities of client and server were separated before the implementation, it was clear, that the communication module needs to support the following features:

- Initializing the communication and connecting to the server.
- Sending a buffer of messages that should be transmitted.
- Receiving messages and storing them into a buffer.
- Being able to add messages to and to get messages from the buffers.
- Supplying a list of all reachable clients to the controller.

A simple activity diagram to visualize the communication modules' working principle was created: The whole process was also designed to be sequential. After the initialization of the clients' socket, a connection to the server is established. To allow the server to associate the newly accepted socket with the trucks' ID, a message without any real content is sent directly server because it contains the ID of the truck. The module then proceeds with sending all the messages to the server that are queued in the buffer of messages to be transmitted. It then polls the socket for messages that were received from the server. These messages are sorted. Messages that were forwarded by the server from other clients are stored to a buffer. Messages from the server that only contain the list of all reachable clients are used to update a vector that contains all IDs of these clients. This procedure is repeated as long as

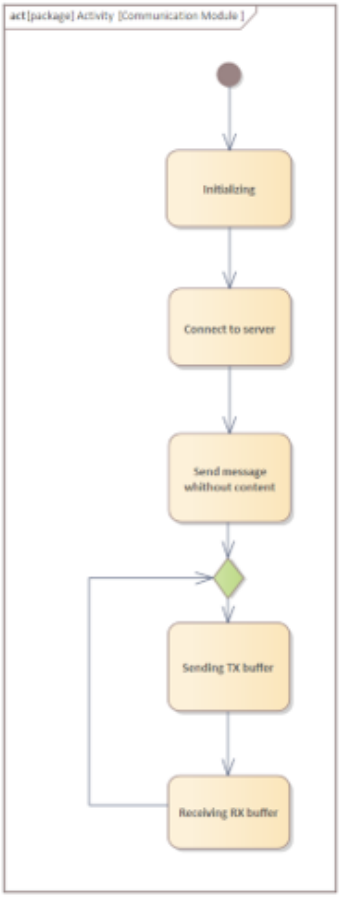

Fig. 19. Working principle of the client

the communication module has an active connection to the server.

Because the communication module is a separate part of the Truck, it was developed as an independent class that exposes functions to allow interactions between itself and the controller of the truck. The exposed functions can be separated into two categories. The first category consists of the functions that need to be executed to carry out the steps of the procedure that was explained. These functions were called “initialize”, “connect_to_server”, “send_txBuffer” and “receive_rxBuffer” and are executed in a separate running thread that the truck provides for the communications module. The second category consists out of functions that allow the controller to interact with the communications module. Because the module is using buffers, the only interaction between the communication module and the controller is done when it comes to reading or modifying buffers. For accessing the buffers that contain messages, the following functions were implemented:

- ”add_tx_message_to_buffer”
  For adding a new message to the TX buffer.
- ”get_last_rx_message_from_buffer”
  For getting the last (newest) message from the buffer. The bool of the parameter determines if the message should be deleted from the buffer after getting it.
- ”get_rx_message_by_index_from_buffer”
  For getting a message at a certain index from the buffer. The bool of the parameter determines if the message should be deleted from the buffer after getting it.

Because the vector containing the IDs of all connected clients is read-only as it gets updated by the server, the function ”get_connected_client_IDs”, that returns a copy of this vector, was implemented.

All the functions that enable the controller to access the buffers of the communication module are utilizing blocking functions as the functions are called from other threads that are running concurrently. A simple mutex was used to implement these blocking functions.

C. Message Structure and MessageID

The basis of communication within, are message structures that are carefully designed to encapsulate all the necessary information for clear and efficient data exchange.

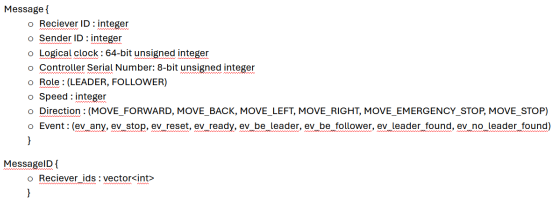

Fig. 20. Message and MessageID structure

Each message contains the following fields (its structure is shown in Figure 20):

- Recipient ID: An integer uniquely identifying the intended receiver of the message within the fleet.
- Sender ID: An integer representing the ID of the truck from which the message originated.
- Logical clock: A 64-bit unsigned integer that ensures messages are processed in the correct chronological order, crucial in distributed systems where timing is key.
- Controller Serial Number: An 8-bit unsigned integer uniquely identifying a controller within a truck, used to direct messages appropriately in systems with multiple controllers.
- Role: An enumeration representing the truck’s role within the fleet, with possible values of `LEADER` or `FOLLOWER`.
- **Speed:** An integer value indicating the truck’s speed, for use in movement commands and status updates.
- Direction: An enumerated type indicating the truck’s movement direction, with options such as `MOVE_FORWARD`, `MOVE_BACK`, `MOVE_LEFT`, `MOVE_RIGHT`, `MOVE_EMERGENCY_STOP`, and `MOVE_STOP`.
- Event: An enumeration indicating the type of event associated with the message. Possible values include `ev_any`, `ev_stop`, `ev_reset`, `ev_ready`, `ev_be_leader`, `ev_be_follower`, `ev_leader_found`, and `ev_no_leader_found`.

These events trigger specific behaviors and state changes within the truck's control logic.

D. MessageID Structure

The `MessageID` structure is utilized when a message is intended for multiple recipients, enabling efficient broadcast communication:

- Recipient IDs: A vector of integer IDs that specifies all the intended recipients of the message. This approach optimizes the communication process by reducing the number of messages sent over the network.

E. Serialization and Deserialization

The utilization of the winsock2.h library within our communication system necessitates a reliable method for message exchange between different entities in the network. Due to the library's operating principle, which relies on the transmission of data streams over TCP/IP, it is imperative that complex data structures, such as those used in our system, are serialized into a stream of characters. JSON serialization offers a standardized and language-agnostic format for converting our message objects into a text-based format that can be easily transmitted over the network. Upon receipt, these character streams are then deserialized back into their original structured form, allowing for the seamless reconstruction and processing of the transmitted data.

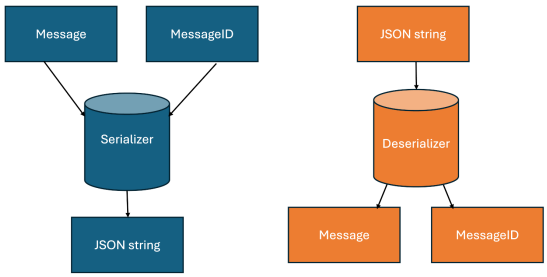

Fig. 21. Serialization and Deserialization

1) Serialization: Serialization is the process of converting an object's state and data into a format that can be transmitted and then reconstructed later. In the context of our project, the `MessageParser` class is responsible for serialization and deserialization of `Message` and `MessageID` objects to and from JSON format.

The Serializer (see Figure 21) comprises `toJSON` and `toJSONID` methods which are responsible for serialization. They take `Message` and `MessageID` objects, respectively, and convert them into JSON strings.

2) Deserialization: Deserialization is the reverse process, where JSON strings are converted back into `Message` or `MessageID` objects.

The Deserializer consists of three important methods:

- `fromJSONVariant` is an overloaded function that can return either a `Message` or `MessageID` object from a JSON string. It determines the type of object to deserialize based on the presence of specific keys in the JSON string.
- `fromJSON` parses a JSON string, extracting data to populate the fields of a `Message` object. It uses a `stringstream` to separate the JSON string into key-value pairs and then assigns these values to the appropriate fields of the `Message` object.
- `fromJSONMessageID` takes a JSON string that represents message IDs and extracts an array of receiver IDs, which it uses to populate a `MessageID` object.

3) Utility Methods: The class also includes utility methods for converting enumeration values to and from strings, which are used during serialization and deserialization:

- `truckRoleToString` and `stringToTruckRole` convert between `truckRole_e` enum values and their string representations.
- `directionToString` and `stringToDirection` handle the conversion for `MovementDirection` enums.
- `eventToString` and `stringToEvent` convert event types to and from strings.

The `MessageParser` class is a critical component that ensures the seamless transfer of complex objects between different parts of the system.

VI. Verification and Validation

During the design and implementation stage, we had done many test case to ensure that the system should behave predictable manner. The test case can be categorize into 3 part:

1) Truck Behaviour test.
2) Communication behaviour test and
3) Integration test

This test lead us to improve the system design and implementation. For instance every component are design in the way that it can be executed independently from other component and can be tested individually. The test also give influence on our code pattern such as using switch case as listed in Listing 2 to realize state machine as in Figure 6.

A. Truck behaviour test

As we explained before, the truck consist of 3 component which is controller, communication module and driver interface. Since the communication module is apart of communication, those components are tested together with the serve in communication test part.

the controller are tested to ensure that the sequence of state are predictable based on given event and input. For instance, when a leader is found (by adding a dummy truck id and information), the controller shall change from waiting state to follower state.

for the driver interface component, test were conducted in order to ensure that the component doesn't effect the behaviour of other component such as preventing other component from

being executed and also to ensure correct output when an input is given from the user. This is partially explained in section IV-B. The integration test (driver interface and controller) implemented for the truck was to ensure the behaviour of the controller output were working as planned.

Listing 2. Main process of a controller. Realization of controller state machine

```
while(true){
    next_state_computer(self_truck->
        event_handler); //set next state
    self_truck->event_handler = ev_any;
        // reset event handler
    switch(next_state){
        case initial:
            self_truck->event_handler =
                state_initial();
            break;
        case waiting:
            self_truck->event_handler =
                state_waiting();
            break;
        case leader:
            self_truck->event_handler =
                state_leader();
            break;
        case follower:
            self_truck->event_handler =
                state_follower();
            break;
        case system_stop:
            self_truck->event_handler =
                state_system_stop();
            break;
        default:
            break;
    }
}
```

B. Communication behaviour test

Client connect to server Server accepting multiple client Server forwarding message

C. Integration test

Now we are at the final stage of testing, the full integration test. For this case we already consider to have a compliable code of the whole system as shown in Figure 3. For this case we already tested the Server-Client scenario, where we have one server and multiple clients, as well as the system for the truck, so lastly we need to test the system as a whole. In this test scenario we do the following procedure:

1) Run Server.exe - to initialize the server which we are using to communicate.
2) Run Truck.exe - to initialize a first truck and we set with ID = 1.
3) Run as many trucks as our machine can handle, as these are the follower's trucks, but the ID cannot be repeated. When all the trucks are connected to the server you should be able to see information such as in Figure 22. This will mean that all the trucks are connected to the server and receiving information from the leader.

The next step for the integration test was to start moving the leader truck and wait to see if all the information was being received by the followers. In this case the integration was a success since, in the integration test we got a positive response from all the followers, shown in Figure 23.

From the Figure 23, we see from the server side (bottom left) that all the trucks are synchronized to the leader. Since the logical clock in this implementation is done in a way that we only increase on every event that sets a new speed or direction.

For finish this system test, a last thing was needed and that was to kill the process of the leader truck, so a new truck could assume its role. As we can see in Figure 24, we kill the first 2 trucks simulating that they left the platoon and now the next truck in the platoon can assume the leadership, which in this case is the truck with an ID = 3.

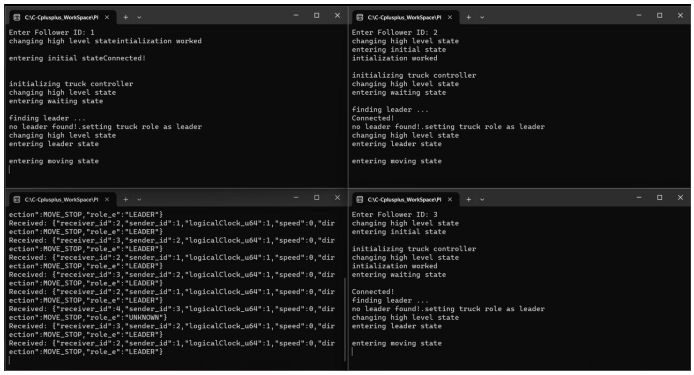

Fig. 22. System initialization test

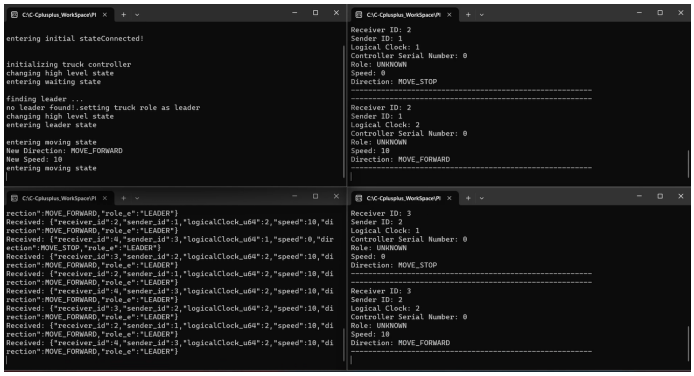

Fig. 23. Server listening test

Listing 3. Main Process of a truck using pthread

```
// Create the threads that will be run in
    the system called: Truck
```

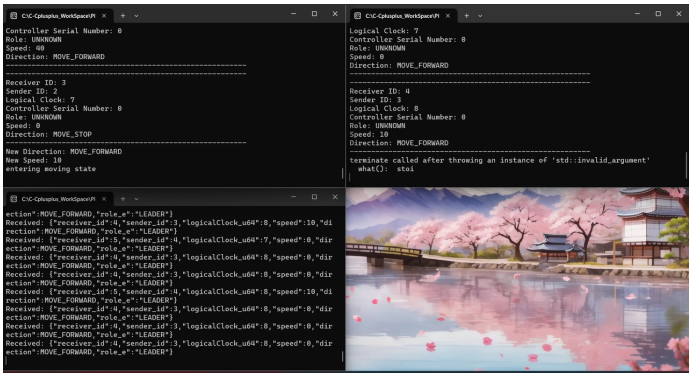

Fig. 24. Leader lost test case

```
pthread_create(&t_communication, NULL, &
    CommsModule::run, &truck_communication
    );
pthread_create(&t_controller, NULL, &
    controller::run, &truck_controller);
pthread_create(&t_interface, NULL, &
    controller::key_board_run, &
    truck_controller);

// Run the threads so the actual
    simulation can be run through the
    thread id called:
// - t_controller
// - t_interface
// - t_communications

pthread_join(t_controller, NULL);
pthread_join(t_interface, NULL);
pthread_join(t_communication, NULL);
```

VII. Summary and Outlook

In this paper we have discussed how we design and implement our truck platooning system from analysis stage, design stage , implementation stage to the verification and validation stage. For the system development, we only focus on a certain use case like finding a leader and message passing and the implementation is only at simulation level. From the development process, we have seen many part from each stage can be further improved. Since there are many domains that need to be explored where it is beyond our ability, the system we designed has many flaws such as the dependability of the system is not well measured, lack of test especially integration test and scalability of the server is unknown. However, we focus on our approach to building a system. In the future, many stake holder shall involve in this project to improve the system requirement,thus improve the system design to make it more dependable.

TABLE I
GIT BRANCHES

Item	Branche's Name
1	communication
2	communication_lhckmn
3	communication_mykyta
4	communication_mykyta_2
5	controller
6	controller-LF-RoGu
7	controller-sheikh
8	lhckmn_communication_merge
9	main
10	main_LF-RoGu
11	main_merge
12	main_test

TABLE II
LINES OF CODE SUMMARY

Item	Amount
Files	31
Codes	1683
Comments	314
Blanks	423
Lines	2420

References

[1] Hwang, Kai, Jack Dongarra, and Geoffrey C. Fox. Distributed and cloud computing: from parallel processing to the internet of things. Morgan kaufmann, 2013.

[2] Janssen, G. R., et al. "Truck platooning: Driving the future of transportation." (2015).

[3] @miscrapp2024, title=Truck Platooning Image, author=Rapp.ch, howpublished=https://www.rapp.ch/sites/default/files/styles/21by9_large/public/uploads/2018-10/rt-stories-truck-platooning-00_ohne_copyright.jpg?itok=3ugNaHXh, year=2024

GitHub Overview

link to GitHub repository: https://github.com/DPS-DistributedIntelligence/DPS-Project

For this project, the implementation was done in a structured way, what do we mean by this, is that every implementation of testing was done in a separate branch from the main branch. This to avoid conflicts and provide a working branch through every commit done in the main branch.

This having as a result a total of 15 branches (including main branch implementation), where either development was done or testing as lested in Table I;

A. Line Of Code

The number of lines and files are listed in Table II and the language that we used is C++.

B. Code directory

```
.
DPS-project
├── Code
│   ├── cmake-build-debug
│   └── src
└── Documentation
.
```

The tree above arethe directory tree of our github reprository and the directory of *src* folder are listed in Figure 25

Directories

path	files	code	comment	blank	total
.	31	1,683	314	423	2,420
. (Files)	1	12	0	3	15
client	2	359	107	107	573
lib	16	395	76	127	598
server	4	283	57	69	409
truck	8	634	74	117	825
truck (Files)	2	35	10	25	70
truck\controller	2	548	62	65	675
truck\decryptor	2	51	2	18	71
truck\interface	2	0	0	9	9

Fig. 25. lines of code directory

In *src* folder, you can find all the source code for the project. The option to simply add all the files in one ”src” folder was made so we can simply update and add to the compilation rules the src folder.

1) client - Source code for the client communication that the object ”Truck” will have to connect to the server.
2) lib - Source code for all enumerations, structures and headers that the project will need.
3) server - Source code for the server to be able to run
4) truck - Source code of all the items that the truck should have such as controller.

C. Amount of commits

Since most of the implementation was done in branches, it will only be shown the most important branches of the implementation.

- main, 86 commits
- communication, 14 commits
- communication-lhckmn, 28 commits
- communication-mykyta, 20 commits
- controller, 33 commits
- controller-LF-RoGu, 32 commits
- controller-sheikh, 33 commits
- threads, 33 commits
- truck-LF-RoGu, 18 commits
- main-merge, 77 commits

AFFIDAVIT

We Luis Fernando Rodriguez Gutierrez, Leander Hackmann, Sheikh Muhammad Adib, Aditya Kumar, and Mykyta Konakh herewith declare that we have composed the present paper and work ourselves and without the use of any other than the cited sources and aids. Sentences or parts of sentences quoted are marked as such; other references concerning the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form have not been submitted to any examination body and have not been published. This paper was not yet, even in part, used in another examination or as a course performance.