Truck Platooning Project

Ulas Arslan
Embbedded Systems Engineering
FH Dortmund
Dortmund, Germany

Ashwin Balaji Arun Kumar
Embbedded Systems Engineering
FH Dortmund
Dortmund, Germany

Adesh More
Embbedded Systems Engineering
FH Dortmund
Dortmund, Germany

Harshavarthan T
Embbedded Systems Engineering
FH Dortmund
Dortmund, Germany

***Abstract*—Truck platooning is a cooperative driving concept where many trucks use vehicle-to-vehicle communication to move in close formation. By this project, the aim is to implement a distributed truck platooning system using socket communication protocol and multi-thread C programming for concurrency in executing, event handling, and communication. Messages are implemented manually by frame which contains truck ID, speed, distance and event types. TX thread state machine decides which message to send to the server. RX thread state machine processes incoming messages based on event types, assigns truck ID, updates speed or distance, handles intrusion, and manages client left events.**

***Index Terms*—thread, TX, RX, state machine, socket communication**

I. Introduction

In the recent days, we can see a majority of logistics companies and automotive companies experimenting with truck platooning. Platooning is the concept in which trucks join together communicate with each other through wireless communication and move along like a single long vehicle [1]. Any truck can join behind a lead truck, and then the follower truck automatically starts driving itself with commands received from the lead truck. The Follower truck starts listening to the instructions provided and corrects its speed, distance and braking. This close-range formation ensures improved aerodynamics, resulting in less fuel consumption and safer trucks by minimising human errors.

To achieve this level of coordination, each truck must stay connected wirelessly and continuously exchange data, events and signals to each other. This leads us to the Distributed perspective of the trucks, where every truck behaves as a independent computing node that communicates, processes events and updates its states. For maintaining a stable platoon a reliable communication, precise control logic and fast reaction times are essential.
This project explores these ideas by implementing a simplified platooning system using socket based communication, multi threaded execution, and state machine driven behaviour. Each truck sends and receives structured messages, processes events such as speed updates or intrusions, and maintains synchronised behaviour with the rest of the platoon. The goal is to understand how distributed communication, concurrency, and event handling come together to support coordinated vehicle behaviour.

II. Motivation

In the advancing field of technology, we can see a major growth in autonomous driving. One important application is Truck platooning, where trucks form a convoy without being physically connected. In a platoon, a synchronized behavious is maintained across all the trucks. A driver drives the lead truck, and all connected trucks follow instructions provided by the lead truck. It focuses on enhancing safety by reducing manual intervention, coordinating control of trucks, lowering emissions, reducing costs and increasing fuel efficiency. The primary aim of the presented work is to gain a practical understanding how such complex, distributed systems can be designed and controlled. Several use cases have been developed to achieve this goal. For example, to maintain reliable communication between trucks and to avoid package loss, the Transmission Control Protocol (TCP) communication protocol has been implemented. To handle client disconnections, the necessary steps for reconnection or termination have been discussed. To reduce high-end computing for repetitive data such as speed and distance, experiments using Graphical Processing Unit(GPUs) have been conducted. Synchronisation between devices is ensured with the help of a logical clock matrix. Tinkercad simulations of certain use cases are also part of the work undertaken.

III. Requirements

A. Functional Requirements

- Truck communication: The system shall safely communicate with each other using TCP
- Safe distance maintenance: The system shall be maintain a distance of 10 metres.
- Obstacle avoidance: The system shall detect obstacles and stop the platoon movement.
- Simulation: The system shall be successfully simulated in hardware.
- Scheduling: The system shall be verified for stimulability using any scheduling algorithm.

- Dynamic addition/removal: The system should allow dynamic addition and removal of trucks and change position accordingly.
- Notification: The system shall notify on any events.
- Message Handling: The system shall not ignore any message from any sender.

B. Non funtional Requirements

- Latency: The system can communicate with minimum latency of 1 ms
- Response/Execution time : The system should respond to interrupts within 2 ms
- Speed Adjustment: The system shall adjust speed based on instruction from the leading truck.

IV. Sketch of approach

In this project, a distributed truck platooning system is designed based on a client–server architecture, where each truck operates as a client and communicates with a central server using TCP socket communication.

A custom-defined frame structure that contains crucial data like truck ID, current speed, distance, and event type is used for message exchange between clients and the server. Message parsing is made easier by this frame-based design, which also makes it possible to add more events.

On the client side, each truck in the platoon represents an independent node that communicates with the central server using TCP socket programming.

The global platoon status is updated on the server side by processing received messages. The server creates the necessary responses, such as commands for speed or distance adjustment, intrusion handling, or emergency brake broadcasts, based on the incoming data. The server employs queue-based message buffering and multi-threading to effectively manage numerous clients.

In general, this method creates a scalable and deterministic truck platooning system appropriate for real-time and safety-critical situations by combining dependable communication, concurrent execution, and event-driven state machines.

V. System Architecture

A. General Overview

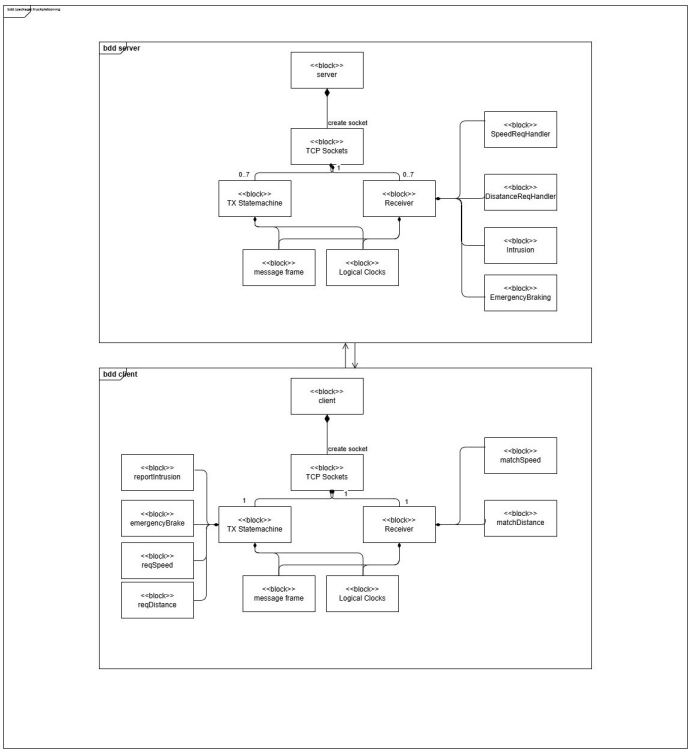

Fig. 1. Block Definition Diagram

Figure 1 illustrates the context-level block definition diagram of the Truck Platooning System. The diagram presents the system boundaries and its interactions with external actors, infrastructure, and optional acceleration components. The operator/user interacts with the system by issuing high-level commands such as intrude, brake, and quit. These commands are received by the server, which acts as the central coordination unit of the platoon. Communication between the server and the client trucks is realized over a TCP/IP network using Winsock or POSIX-based socket interfaces. The system relies on a structured message protocol and matrix-based logical clock synchronization to ensure consistent state awareness across all distributed nodes. An optional GPU offloading component is included, represented by a Google Colab Tesla T4 GPU. This module is used to demonstrate acceleration of computationally intensive tasks such as sensor data processing and matrix operations, but it is not mandatory for core system functionality. The physical world or simulation environment provides vehicle-related inputs such as speed, inter-vehicle distance, intrusion scenarios, and braking dynamics. The system outputs include console logs, serialized matrix clock data, and scheduling or performance analysis reports, which are used for verification and evaluation purposes. Overall, this diagram highlights the

distributed nature of the platooning system and clearly defines the interaction between software components, hardware resources, and external actors.

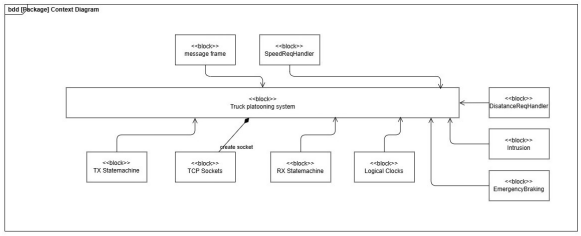

Fig. 2. Context Diagram

Figure 2 presents the context-level Block Definition Diagram (BDD) of the Truck Platooning System, focusing on the internal functional blocks and their interactions within the system boundary. At the center of the diagram is the *Truck Platooning System* block, which represents the core software system responsible for coordinating distributed trucks in a platoon. The system integrates communication, synchronization, control, and safety-related functionalities. TCP-based communication is handled through the *TCP Sockets* block, which is responsible for socket creation and network-level data exchange. Incoming and outgoing messages are processed by the *RX Statemachine* and *TX Statemachine*, respectively. These state machines ensure deterministic handling of received frames and structured transmission of outgoing commands. All transmitted and received data are encapsulated using the *Message Frame* block, which defines the common message format shared across the distributed nodes. To maintain temporal consistency in the distributed system, the *Logical Clocks* block is integrated, enabling logical time synchronization between server and clients. Control-related requests are processed by dedicated handler blocks. The *SpeedReqHandler* and *DistanceReqHandler* blocks manage speed and distance adjustment requests received from client trucks. Safety-critical events are handled by the *Intrusion* and *EmergencyBraking* blocks, which are responsible for detecting hazardous situations and triggering immediate braking actions when required This context diagram highlights the modular structure of the Truck Platooning System and clearly separates communication, control, and safety concerns.

B. Leader-Follower (Client-Server) Model

The system uses a leader–follower architecture, where a single server coordinates an arbitrary number of clients. Each client corresponds to a physical truck following the leader. The server assigns a unique identifier to each client and maintains global knowledge of the platoon. Clients connect to the server using TCP/IP sockets (via Winsock on Windows or POSIX sockets on Linux). TCP ensures reliable, in-order delivery, which is critical for safety. Each client has three threads (TX, RX, command) that run concurrently; the server runs separate RX and TX threads per client.

- Client connection: The client establishes a TCP connection and sends a message for joining platoon. The server responds with a CLIENT_ID message and the initial logical clock matrix.
- Normal operation: The client alternates sending frames containing event types SPEED and DISTANCE with its current speed and distance. The server receives these frames, updates its global matrix clock and, if necessary, sends new SPEED/DISTANCE commands back to the client. The client adjusts its target values accordingly.
- Intrusion: If the client's command thread triggers an intrusion, the client sets its current Distance to a short value (e.g., 10 m), calls function that reports the intrusion event to server, increments the logical clock and sends an INTRUSION frame. The server computes a safe speed/distance and sends commands. If the reported distance is below 5 m, the server triggers an emergency brake for all clients.
- Emergency brake: The client or server can issue an EMERGENCY_BRAKE message. On receipt, all clients set their target speed to zero and apply full braking. The server uses `urgentBrakeAll()` to broadcast this event.
- Client leave: If a client disconnects, the server calls `broadcastClientLeft()` to inform remaining clients and resets the corresponding row and column in its matrix clock.

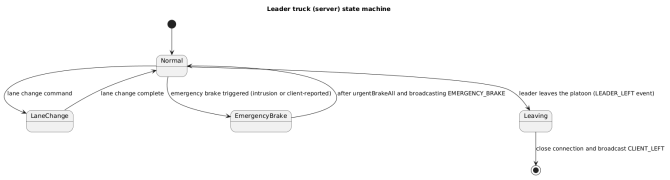

Fig. 3. Leader State Machine

1) State machines: Figure 3 illustrates the state machine of the leader truck, which also acts as the server in the truck platooning system. The system starts in the Normal state, which represents regular platoon operation. In this state, the leader continuously communicates with follower trucks and maintains target speed & distance. When the lane change command is received, the leader state machine moves to the lane change state, coordinating the lane change, and reverts to the Normal state upon completion of the lane change. If the event of the emergency brake is triggered by the intrusions detected or a client-reported emergency, the leader moves to the state of Emergency Brake. In this state, all the follower trucks are given a command to apply the emergency brake. Once the emergency process is completed, the system returns to the Normal state. In case the leader leaves the platoon, the system will enter the Leaving state, where the communication

channel is closed and the message CLIENT_LEFT is sent out.

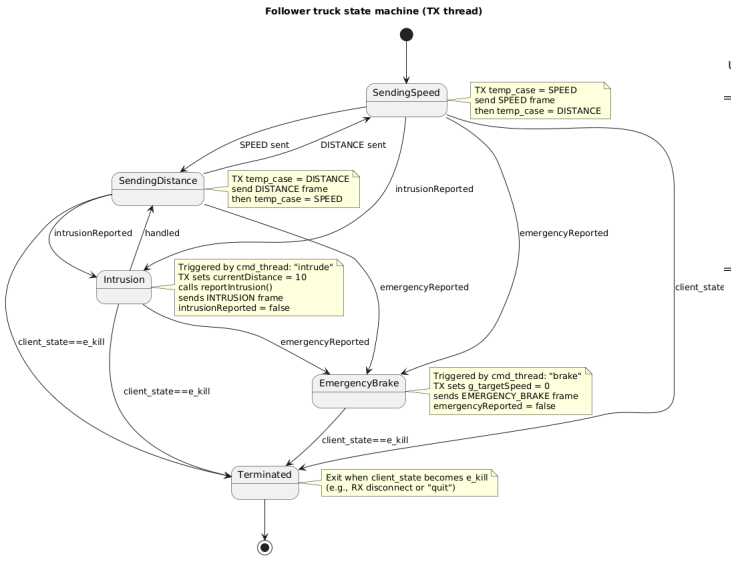

Fig. 4. Follower State Machine

Figure 4 depicts the state machine of the follower truck transmission thread, which is involved in the sending of status and event info to the leader truck. In this case, the state machine begins in the sending speed state, in which the follower begins to send speed info periodically. It alternates between the sending speed state and the sending distance state. When an event of intrusion is detected, the system will enter the Intrusion state. At this time, the follower will report the intrusion to the leader, as well as updating its internal target distance. Once the intrusion is finished, the system will resume its normal transmission states.

2) Sequential Interaction: Figure 5 presents the sequence of interactions between the client and server during normal operation, intrusion handling, and emergency braking. After initialization and ID assignment, the client enters a normal telemetry loop where speed and distance frames are periodically sent along with a serialized logical clock matrix. The server merges the received matrix with its local clock to preserve causal consistency. During intrusion handling, the client reports the event with updated telemetry data. The server responds by computing safe speed and distance values and sending updated commands to the client. In the emergency brake scenario, the client sends an EMERGENCY_BRAKE frame, which the server broadcasts to all connected clients. Both server and clients immediately apply braking and update their target speeds to zero, ensuring synchronized emergency response.

3) Activity Diagram: Figure 6 presents the high-level activity flow of the client truck. The execution begins with the initialization of the Winsock library and establishing a TCP connection with the server. After successful connection,

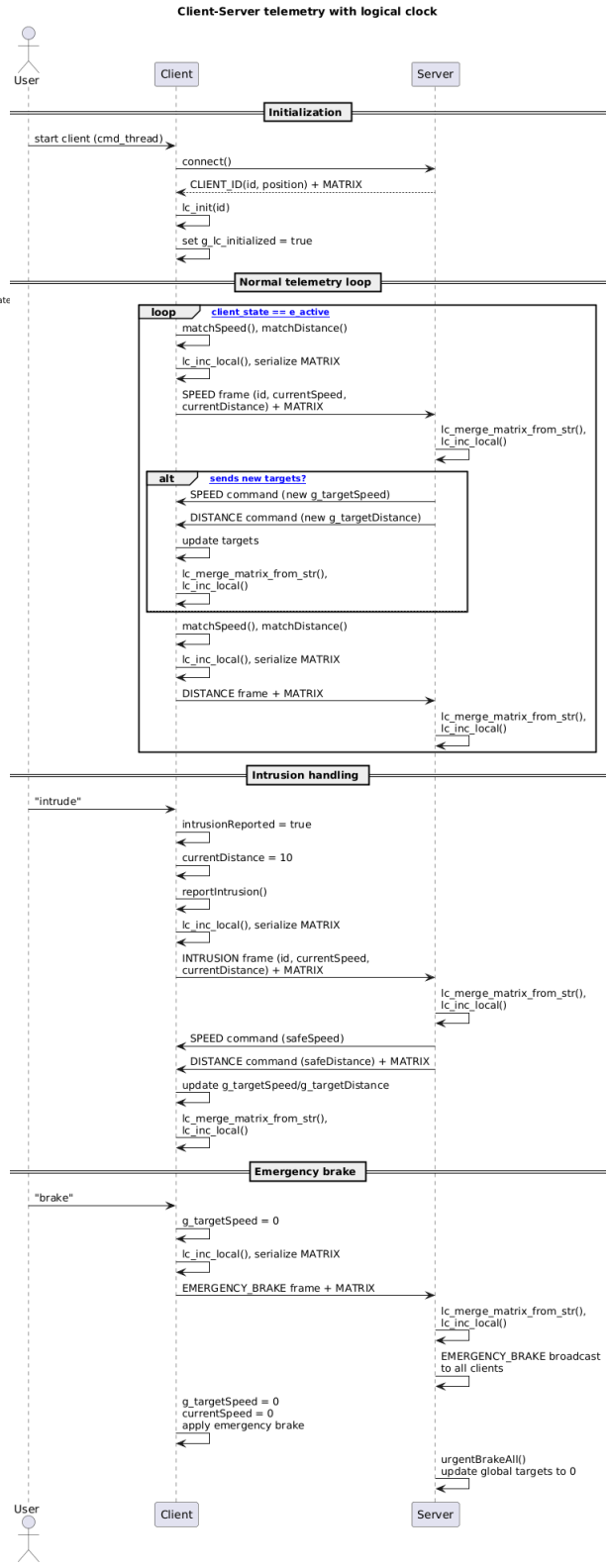

Fig. 5. Sequence diagram between client and server.

the client initializes its internal truck state and logical clock. The client then spawns three concurrent threads: transmission (TX), reception (RX), and command handling. These threads operate in parallel while the client remains in the `e_active` state. The TX thread periodically sends telemetry data, the RX thread processes incoming server commands, and the command thread listens for user inputs. Upon disconnection, fatal error, or user-issued termination command, the client transitions to a cleanup phase where all resources are released and the program exits gracefully.

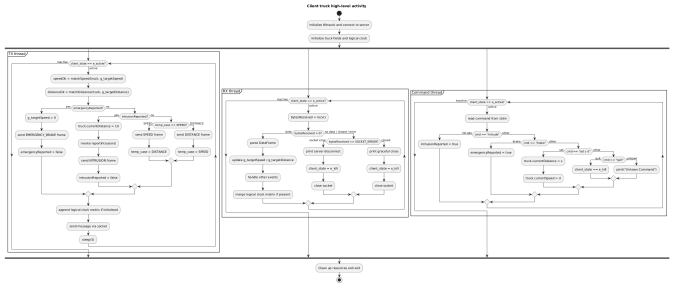

Fig. 6. Client Truck High-level Activity

C. Network Protocols

The system exclusively uses TCP for critical communications. UDP could be used for non-critical, low-latency data, but in the provided code all data is sent over TCP [2]. This decision ensures reliability at the cost of potential latency; however, safety messages (intrusions, emergency brakes) are rare, and the overhead of TCP is acceptable for a research prototype.

D. Message Structure and Parsing

There are various messages being transmitted between client and server. Hence, there is much load on the server to handle message from different sources. A way to simplify ensuring all the messages have the same format. Thereby, reducing code size and computation load. The message format implemented in this project is as found in Table I

TABLE I
MESSAGE FRAME FORMAT

Truck id : 3	read/write	parameter	value	Event type

Truck ID identifies the sender. It is used by the server to index the global matrix clock and to address messages. Read/Write flag indicates whether the frame is a telemetry report or a command. The parameter and the value represent the meaning that depends on the type of event (e.g. parameter = 0 for speed, value = current speed). Event type is enumerated type indicating the frame content (e.g. `SPEED`, `DISTANCE`, `INTRUSION`, `EMERGENCY_BRAKE`, `CLIENT_ID`, `CLIENT_LEFT`).

Frames are serialised into plain strings by `constructMessage()`, which concatenates the fields separated by spaces and appends a newline. On the receiving side, the string is parsed using `sscanf()` in `parseMessage()`, filling a DataFrame structure and validating the event range. This simple format facilitates debugging and extends easily; however, a more robust format (such as JSON or Protobuf) could be adopted in future work to support complex payloads.

E. Threading and Concurrency

In a distributed and parallel computing environment, effective thread management is essential for handling multiple connected devices simultaneously. Since each node in the platoon operates independently and generates concurrent requests, parallel execution becomes necessary to ensure responsiveness and real-time behaviour. To realise this behaviour, POSIX threads (Pthreads) in C are used to implement multithreading within both the server and client components.

VI. IMPLEMENTATION

A. Data, Signals and Events

Table II summarises the key data, signals and events in our system.

TABLE II
SUMMARY OF KEY DATA, SIGNALS, AND EVENTS

Category	Examples	Description
Data	TruckID, currentSpeed, currentDistance, targetSpeed, targetDistance	Discrete state variables stored in the Truck structure and used for control and identification.
Signals	Acceleration, brake pressure, steering angle, gear position, camera/LIDAR, GPS distance	Physical sensor signals processed locally and converted into data or events.
Events	JOIN, CLIENT_ID, JOIN_ACCEPTED, LEAVE, CLIENT_LEFT, INTRUSION, EMERGENCY_BRAKE, SPEED, DISTANCE	Discrete messages exchanged between clients and server to report state or issue commands.

B. Speed/Distance Matching

The functions `matchSpeed(truck, targetSpeed)` and `matchDistance(truck, targetDistance)` implemented in `client.c` realize simple closed-loop control mechanisms. These functions iteratively adjust the `Truck` structure's `currentSpeed` and `currentDistance` fields by fixed increments of ± 5 km/h and ± 10 m, respectively,

until the specified target values are reached.

After each update step, the new value is printed to the console to facilitate debugging and traceability during execution. Both functions return true once the target speed or distance has been successfully achieved.

Although the control logic is intentionally simplistic, it serves an important purpose within the system: preventing abrupt changes that could lead to instability in a real-world vehicle. In a physical implementation, such discrete step adjustments would typically be replaced by more advanced control strategies, such as PID controllers or follow-the-leader algorithms. For the purposes of simulation, however, the employed stepwise adjustments provide sufficient realism while maintaining clarity and ease of implementation.

Figure 7 illustrates the distance control logic executed on the client side. The activity begins by acquiring a mutex lock to protect shared truck state variables. If the current distance is less than the target distance, the system gradually increases the distance by a predefined step size. On the other hand, if the current distance is greater than the target distance, the system gradually reduces the distance. Boundary checks are implemented to prevent the distance from overshooting the target distance. Once the adjustment is made, the system records the new distance and releases the mutex. This method ensures that the system converges to the target inter-vehicle distance in a thread-safe and smooth manner.

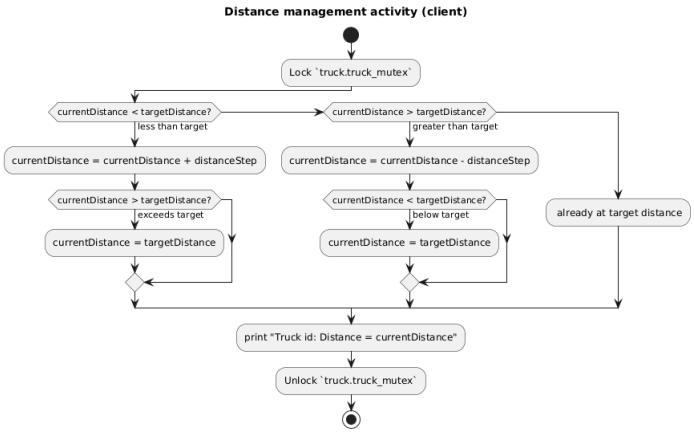

Fig. 7. Distance Management Activity

Figure 8 presents the SysML parametric diagram used to model the speed and distance control behavior of a client truck within the platoon. The central block, *MotionController*, encapsulates the control-related value properties, including `currentSpeed`, `targetSpeed`, `speedStep`, `currentDistance`, `targetDistance`, and `distanceStep`. These parameters represent the dynamic state of the vehicle and the control increments applied

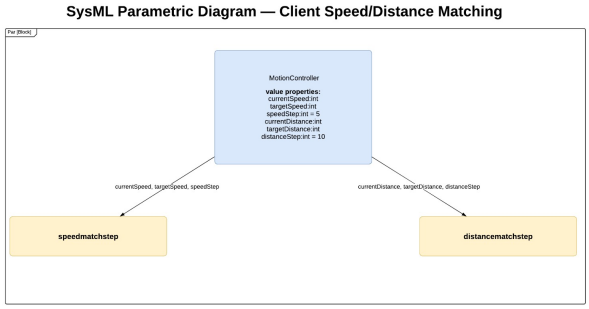

Fig. 8. Parametric Diagram for Speed and Distance Matching

during adjustment. Two constraint blocks are defined: *speedMatchStep* and *distanceMatchStep*. The *speedMatchStep* constraint calculates the incremental speed adjustment required to gradually match the target speed received from the server, using a predefined step size. Similarly, the *distanceMatchStep* constraint regulates the inter-vehicle distance by incrementally increasing or decreasing the current distance toward the target distance. The parametric relationships ensure smooth and stable convergence to desired speed and distance values, avoiding abrupt changes that could compromise safety or passenger comfort. This model directly reflects the implementation logic used in the client-side control algorithms.

C. Intrusion and Emergency Braking

The `reportIntrusion()` function implemented in `client.c` reduces the truck's speed to a predefined safe threshold of 20 km/h and prints a status message. When the server receives an `INTRUSION` frame, the `server_handle_intrusion()` function computes safe speed and distance values.

The safe speed is calculated as the maximum of 20 km/h and `(reportedSpeed - 20)`, while the safe distance is set to `reportedDistance + 50` m. These values are then pushed to the client's `TxQueue` as `SPEED` and `DISTANCE` commands and are subsequently processed by the client's TX thread.

This policy ensures that the following truck reduces its speed and increases its inter-vehicle distance when an intrusion is reported. The threshold of 20 km/h and the offset of 50 m are design choices; in a real system, these parameters would be derived from braking distance, sensor accuracy, and other physical considerations.

When a client sends an `EMERGENCY_BRAKE` frame, or when the server determines that a safe distance cannot be maintained (for example, if an intrusion is dangerously close), the server invokes the `urgentBrakeAll()` function. This function sets the global target speed to zero for all clients, increments the global matrix clock, and pushes an `EMERGENCY_BRAKE` event to each client's queue. Upon receiving this event, the

client RX thread sets its `g_targetSpeed` to zero, and the TX thread gradually reduces `currentSpeed` to zero. This broadcast mechanism ensures that all trucks in the platoon brake simultaneously.

D. Leader election

A practical scenario in platooning is maintaining the platoon when the leader leaves. Our initial approach was to implement a leader election mechanism so that a new leader could automatically take over. However, this approach would only be feasible in a purely node-based structure, which we realised midway through the implementation.

Now in our present client-server architecture, we then decided to use a batch script to execute the client. When the server leaves the platoon, this event is broadcast to all clients. All remaining clients enter a sleep mode while waiting for a new server. The client with position = 1 exits the platoon with a specific return value. This return value triggers that client to execute the server code. A batch script has been added to the repository to support this behaviour, although a full implementation is not yet completed.

E. Queue structure

RX or TX handlers, can simultaneously engage with message data. The queue offers a thread-safe storage managed by a mutex and condition variable, avoiding race conditions when several threads push or pop frames. It is designed for concurrent producer-consumer usage with minimum load. The queue stores frames by value in an internal buffer. `TXQueue_push` addes an input frame into the buffer. `TXQueue_pop` fetches a frame from the buffer. The queue is limited by a fixed-size. When it is full, push cannot be performed, thus function return false without blocking. Similarly when queue is empty, pop cannot be performed and return false. This design favors non-blocking behavior over waiting, which can be important for systems with strict timing requirements. The synchronization is operated by mutexes and conditional variables. Mutex protects all queue states and condition variable named `notEmpty` is triggered after successful push operations to report potential consumers that data is available. The reason we use a queue structure is while TCP communication can occur in bursts, application logic may process messages at a different rate. A fixed-size queue allows producer (RX thread) to quickly queue frames and consumer (TX thread) to process them asynchronously, helping to stabilize efficiency and reduce timing inaccuracies.

F. Thread handling

The system relies on multi-threading to handle sending, receiving and user commands concurrently. On client side, three threads are created:

- TX thread: The `TXthread()` function runs in an infinite loop while the client state is active, where it calls `matchSpeed()` and `matchDistance()` to gradually align the current speed and distance with their target values, manages intrusion and emergency flags, constructs outgoing frames using `constructMessage()`, appends the logical clock matrix string, transmits the frames over the socket, prints the matrix, sleeps for 5 ms, and alternates between sending `SPEED` and `DISTANCE` frames using the `temp_case` state variable.

- RX thread: For each event type, the system executes the corresponding logic, including updating `g_targetSpeed` and `g_targetDistance`, merging and initializing the matrix while setting the client's ID and position, resetting the node's row and column and adjusting positions upon `CLIENT_LEFT`, setting the target speed to zero in case of `EMERGENCY_BRAKE`, performing no immediate action for `INTRUSION`, and processing general telemetry data.

- Command thread: Reads commands from standard input. It sets flags about intrusion or emergency brake or updates current speed/distance accordingly. For quit, it kills the client to terminate threads.

On the server side, two threads handle each client:

- The `ServerRxHandler()` function runs in a loop while the client socket remains open, receives frames using `recv()`, parses incoming messages, and triggers the corresponding event handling logic; for `INTRUSION` events, it invokes `server_handle_intrusion()` to compute safe speed and distance values and enqueues the resulting commands into the client's `TxQueue`, for `EMERGENCY_BRAKE` events it calls `urgentBrakeAll()`, merges the received logical clock matrix into the global matrix, increments the local clock, and upon client disconnection resets the node state and notifies the remaining clients.

- The `ServerTxHandler()` function continuously pops frames from the client's `TxQueue`, constructs outgoing messages, appends the global logical clock matrix, transmits the frames over the socket, and prints the updated matrix.

These threads run concurrently and rely on mutexes in the logical clock functions to prevent race conditions. Proper locking ensures that matrix updates and resets are atomic. The concurrency model demonstrates fine-grained parallelism on the CPU, while GPU threads handle data-parallel tasks.

Figure 9 illustrates the communication failure handling mechanism implemented in the client receive thread. The RX

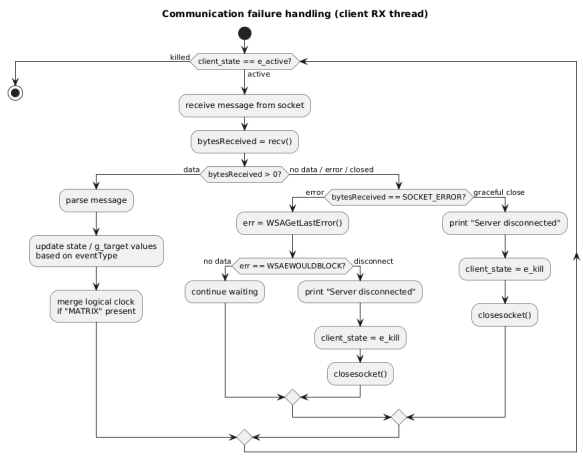

Fig. 9. Communication Failure Handling Activity Diagram

thread operates while the client state is set to `e_active` and continuously listens for incoming messages using a non-blocking `recv()` call. If the received byte count is greater than zero, the incoming message is parsed and processed. Depending on the received event type, the client updates its local state and global target values such as speed and distance. If a logical clock matrix is present in the message, it is merged with the local logical clock to preserve causal ordering. If no data is received and a socket error occurs, the error code is evaluated using `WSAGetLastError()`. In the case of `WSAEWOULDBLOCK`, the system interprets this as a temporary absence of data and continues waiting. For all other socket errors or graceful disconnection events, the client transitions to the `e_kill` state, closes the socket, and terminates the RX thread safely. This mechanism ensures robust fault tolerance against transient network delays and permanent communication failures.

G. Logical Clock Matrix

In a distributed platooning system, each vehicle acts as an independent computational node, generating events such as state changes, control messages, or emergency notifications. Maintaining a uniform ordering of events across all nodes becomes crucial because these events take place simultaneously and without a shared global clock. To address this, the system utilizes a logical clock matrix, which provides an established method for capturing causal links between scattered events.

An n*n logical clock matrix is used to assign a unique index to each node in the platoon, where n is the maximum number of active devices. The server is assigned the index 1, while the client's is determined by the index ID they received upon joining the platoon. Every node maintains its own matrix, which records both its local event history and its knowledge of events occurring at other nodes.

Whenever a node generates an event such as constructing a message, updating speed or distance, or responding to a control instruction, it increments the diagonal entry corresponding to its own index. This increment represents the occurrence of a new local event. Before transmitting a message, the node serialises its entire logical clock matrix and appends it to the outgoing payload. Upon receiving a message, the destination node deserialises the matrix and performs an element wise merge. This guarantees that the most current and causally accurate view of the system's event history is reflected in the receiver's matrix.

Through this mechanism, the platoon maintains a coherent ordering of distributed events without relying on physical time. The logical clock supports synchronisation between trucks, assists in resolving conflicting updates, and provides a reliable foundation for handling dynamic behaviours such as client disconnections, emergency exits, and rejoining scenarios. This approach ensures that all nodes maintain a consistent understanding of system state, which is critical for coordinated control in autonomous platooning environment.

H. GPU implementation

The project uses an NVIDIA T4 GPU in Google Colab for two tasks:

- Sensor processing: The `receiveData` kernel analyses sensor inputs (e.g., camera frames or LIDAR point clouds). It runs many threads in parallel, each computing features or detecting obstacles. It states that the T4's 2 560 CUDA cores and 320 Tensor Cores make it well suited for AI inference and data-parallel operations. In the context of this project, the GPU could be used to run a neural network that predicts obstacles.

- Distance monitoring: The `monitorDistance` kernel runs concurrently with `receiveData` and updates the separation distance. It increments only the diagonal of the matrix clock on the GPU and does not communicate with other kernels (streams), so the off-diagonal entries remain 0. This corresponds to two independent tasks: distance control and sensor analysis, that execute in parallel on the GPU.

Both kernels run in separate CUDA streams. Streams allow asynchronous execution on different streaming multiprocessors, letting the GPU perform multiple tasks simultaneously. In this simplified example, the logical clock demonstrates concurrency between two tasks: `receiveData` and `monitorDistance`.

Three programming models—CUDA, OpenMP and Threads—in terms of purpose, features and suitability for different tasks.The discussion of parallel programming models and justify our choice of CUDA for GPU components

and threads for CPU components.

TABLE III
COMPARISON OF PARALLEL PROGRAMMING MODELS

Model	Purpose	Features	Use Case Fit
CUDA	GPU-based parallel computing	Massive thread parallelism; block/warp model; shared and global memory	Well-suited for data-parallel workloads such as sensor data processing and distance calculations.
OpenMP	CPU shared-memory parallelism	Compiler directives; simple usage; thread and reduction support	Useful for moderate CPU parallelism, e.g., simulation or server-side processing.
Threads (POSIX)	Low-level concurrency control	Explicit thread management; mutexes and condition variables	Best for tightly coordinated tasks such as TX, RX, and command handling threads.

To experiment with the matrix clock implementation on the GPU, we developed a Python-based version using the PyTorch framework. The `GpuMatrixClock` class is designed to closely replicate the behavior of the corresponding C implementation while leveraging GPU acceleration.

The matrix clock is stored as a `torch.Tensor` and allocated on the selected device, defaulting to `cuda:0` when a compatible GPU is available. Local events are handled by the `inc_local()` method, which increments the diagonal element corresponding to the local node using an in-place tensor operation to avoid unnecessary memory transfers.

For communication purposes, the `serialize_matrix()` method converts the tensor from GPU to CPU memory and formats it as a string with the prefix `MATRIX:`, enabling straightforward transmission. Upon receiving a serialized matrix, the `merge_matrix_from_str()` method parses the input and performs a column-wise maximum operation on the GPU. The resulting values are then used to update the local row of the matrix clock, ensuring consistency with the matrix clock merge semantics.

This demonstration shows that the matrix clock concept is not limited to CPU code; it can run on a GPU and integrate with AI pipelines. In a large system, one could use PyTorch to accelerate matrix updates or integrate logical clocks with neural network computations.

VII. EVALUATION

A. Scheduling

In this project, a single CPU is used to execute the server-side components of the truck platooning system. The server includes three periodic tasks that perform receiving data, transmitting messages, and handling emergency braking events. The characteristics of these tasks are used to define scheduling constraints such as worst-case execution time (WCET), period, deadline and priority are demonstrated in Table IV.

TABLE IV
TASK PARAMETERS

Task	Method	T (ms)	D (ms)	WCET (ms)
T1	ServerTXHandler	2	2	0.5
T2	ServerRXHandler	2	2	0.1
T3	urgentBrakeAll	2	2	0.2

WCET stands for the task's maximum execution time under the worst circumstances. To measure WCET, we get help from the special libraries of C language and define a helper method that measures time. Although WCET values are obtained through execution-time measurements, measurement-based WCET estimation cannot ensure absolute worst-case behavior because of unpredictable situations. Therefore, a safety margin is added to the measured WCET values. The margin is chosen nearly %20 percent and value rounding is used. Hence, WCET values are stated 0.5, 0.1, and 0.2 respectively. In this project, the period and deadline of each task are selected equally. We followed the implicit-deadline task model, where the relative deadline is equal to the task period. This assumption is widely used in real-time systems because it eases schedulability analysis while still providing safe timing. A period value of 2ms was chosen because the task execution times were very short. Therefore, we found it logical to choose a period value that is far from the execute time but not significantly larger. We tried 1 ms initially, but CPU utilization was excessive. Then we assumed 2 ms for period.

1) EDF Scheduling: In this project, the considered tasks are scheduled using an EDF (Earliest Deadline First) approach to meet strict real-time constraints. To determine schedulability of server, the formula 1 used to verify the satisfying condition.

$$\text{Schedulability} = \sum \frac{C_i}{T_i} \tag{1}$$

Tasks can be scheduled if the schedulability is less than or equal to 1, otherwise tasks cannot be scheuled. The formula above demonstrates the CPU utilization for each tasks. This sum is calculated as 0,4. Hence, this component is schedulable.

2) RMS Scheduling: Alternatively, the system can be scheduled using RMS (Rate Monotonic Scheduling). In this work, all tasks are modeled as periodic with equal periods of 2 ms and deadlines equal to their periods. The schedulability of the task set under RMS is evaluated using the Liu and Layland utilization bound, which is showed in formula 2.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(2^{\frac{1}{n}} - 1\right) \tag{2}$$

where n is the number of tasks that we want to schedule. Since we have 3 tasks, the bound derived as approximately 0,78. We found the total CPU utilization as 0,4 which is less than 0,78. Therefore this component is also schedulable when RMS used.

RMS uses fixed priorities derived from work periods, whereas EDF dynamically allocates priorities based on task deadlines and offers optimal processor utilization [3]. Due to their low consumption, both methods ensure schedulability for the task set under consideration. In actuality, EDF gives greater flexibility in managing fluctuating deadline limitations, whereas RMS offers easier installation and predictable behavior.

B. Simulation

In order to validate and verify the proposed platooning system architecture, a simulation of an embedded ECU using Arduino has been implemented, figure 10. The simulation has been carried out for a leader and a follower scenario, and vehicle dynamics have been emulated.

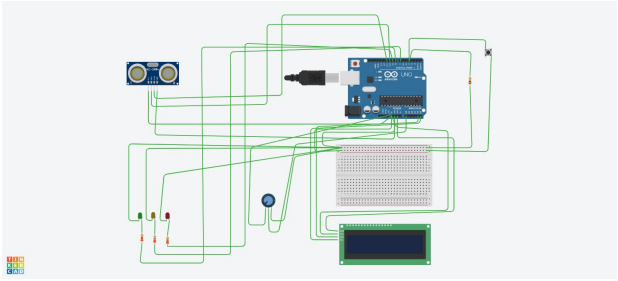

Fig. 10. Arduino Simulation

Table V summarizes the input signals that trigger the control algorithm in the simulation of the platooning electronic control unit (ECU). These signals are simplified representations of information obtained from sensors or driver inputs in a real vehicle. The gap (distance) signal represents the longitudinal distance between the leading and following vehicles and is fed analogously to the Arduino via a potentiometer. This signal is a key input in calculating the safe following distance. The relative speed signal represents the relative speed of the following vehicle to the leading vehicle. In the simulation, this value is modeled with a potentiometer and is used in the Time-to-Collision (TTC) calculation. The hazard detected signal is a button representing a manual emergency trigger. When this input is active, the system switches directly to emergency braking mode, regardless of all other conditions. These inputs are used together to enable the ECU to switch between normal driving, comfort braking, and emergency braking states.

TABLE V
SIMULATION INPUTS (SENSORS AND SIGNALS)

Signal	Source	Arduino Pin
Gap (distance)	Potentiometer	A0
Relative speed	Potentiometer	A1
Hazard detected	Push button	D2

Table VI defines the outputs and status indicators produced by the ECU simulation. These outputs allow the system's decisions to be visually and numerically communicated to both the user and the test environment. The normal/comfort indicator is represented by a green LED and shows whether the system is operating in normal follow mode or comfort braking mode. A constant illumination of the LED indicates normal operation, while any other behavior indicates comfort braking. The emergency indicator is shown by a red LED and indicates that the system has entered emergency braking mode. This output is activated in case of a critical distance violation or the detection of a hazard signal. The brake command is a control output generated by the software that represents the braking intensity. This value is calculated within the Pulse Width Modulation (PWM) range and monitored via the serial port, allowing for analysis of system behavior. Thanks to these outputs, the simulation makes it possible to both visually observe and quantitatively evaluate the ECU's decision-making logic.

TABLE VI
SIMULATION OUTPUTS (ACTUATORS AND INDICATORS)

Output	Element	Arduino Pin
Normal / Comfort indicator	Green LED	D8
Emergency indicator	Red LED	D9
Brake command	Software variable	–

The logic in the controller is essentially a priority-based mode selection, as in a state machine:

- Normal Mode: In the absence of any hazards, if the measured distance is greater than the reference gap, no braking is performed. This is a steady-state condition for the platoon.

- Comfort Braking Mode: When the follower becomes too close to the leader but no critical condition exists, the system applies a progressive braking command. The braking intensity increases proportionally with the distance error and is scaled by an estimated road friction coefficient This gives a smoothing action without sudden speed changes.

- Emergency Braking Mode: Emergency braking has the highest priority and can be initiated either by a manual

hazard input or due to a decrease in the distance below a critical threshold. In this mode, a saturated braking signal is sent, corresponding to a maximum braking request.

The active mode is indicated simultaneously by the use of LED lights, where green indicates normal, yellow indicates comfort braking, and red indicates emergency braking, while a 16x2 LCD permanently shows the simulated speeds of the leader and follower, the calculated distance, along with TTC. All internal variables are also recorded via the serial interface, facilitating analysis.

C. Fault Tree Analysis

Fault Tree Analysis (FTA) is a safety analysis method that is used to identify the main causes of hazardous events in safety-critical systems. In this project, FTA is applied to the truck platooning system to analyze potential failure cases that might lead to unsafe behavior, such as collisions or incorrect safety indications. The objective of FTA are identify hazards that can lead to unsafe operation, decompose hazards into basic failure events, assign probabilities to failure events, evaluate overall risk, define mitigation strategies.

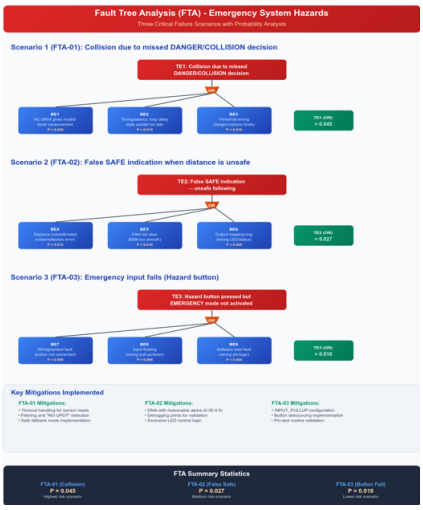

Fig. 11. Fault Tree Analysis of Possible Hazards

Figure 11 illustrates three major hazard scenarios identified in the system: collision due to missed danger detection, false SAFE indication under unsafe conditions, and failure of the emergency hazard input.

The first and most critical hazard represents a collision caused by the system failing to detect or react to a dangerous situation in time. This top event can occur because of invalid distance sensor readings, scheduling or timing delays in decision logic, or incorrect threshold configuration for danger detection. Since these events are connected using an OR gate, the occurrence of any one of them may lead to a collision. The calculated probability for this hazard is 0.045. It making it the highest-risk scenario among the analyzed cases.

The second hazard corresponds to a situation where the system incorrectly indicates a safe state even though the following distance is unsafe. Possible causes include noisy sensor measurements, insufficient filtering or faults in the LED control logic. These events are also combined using an OR gate, resulting in a hazard probability of 0.027. This scenario represents a medium-level safety risk.

The third hazard occurs when the emergency hazard button is pressed but the system fails to activate the emergency mode. This may be caused by physical button failure, incorrect input configuration such as missing pull-up or debounce logic, or software faults in the emergency handling routine. The overall probability of this hazard is calculated as 0.018, which is the lowest among the three scenarios.

The FTA results indicate that missed danger detection leading to collision is the most critical risk. Therefore it requires the highest design priority. To mitigate these hazards, the system implements fail-safe state transitions, sensor timeout and freshness checks, conservative decision logic, validated filtering mechanisms, and robust emergency input handling. These measures significantly reduce the likelihood of hazardous system behavior and improve the overall safety of the truck platooning system.

D. Unit tests

The test suite includes validation, defect, and component-level test. Validation tests verify proper message processing and system functionality in standard conditions. Defect testing emphasizes resilience to erroneous inputs and edge cases. Component tests confirm the proper interaction among frame handling, queue management, and server logic.

TABLE VII
VALIDATION TEST CASES SUMMARY

Test Name	Component	Summary
Write speed	Frame	Checks correct creation and parsing of `SPEED` write messages.
Read distance	Frame	Validates correct parsing of `DISTANCE` read messages.
Emergency brake trigger	Emergency handling	Ensures correct encoding and decoding of emergency brake commands.
Intrusion trigger	Server logic	Verifies intrusion handling generates `SPEED` and `DISTANCE` commands.
Queue frame test	Queue	Confirms correct enqueue and dequeue behaviour of frame data.

TABLE VIII
DEFECT TEST CASES SUMMARY

Test Name	Component	Summary
Message parsing & invalid inputs	Frame parser	Rejects malformed or invalid messages safely.
Push to queue when full	TxQueue	Verifies protection against queue overflow.
Pop from empty queue	TxQueue	Ensures safe handling of dequeue from an empty queue.
Speed reset	Emergency handling	Confirms emergency braking resets speed to zero.
Emergency trigger	Intrusion handling	Validates speed and distance commands during critical intrusion.

TABLE IX
COMPONENT TEST CASES SUMMARY

Test Name	Components	Summary
Frame queue roundtrip	Frame + Queue	Validates end-to-end data flow from queue to message parsing.
Intrusion queue commands	Server + Queue	Ensures intrusion handling queues `SPEED` and `DISTANCE` commands correctly.
Broadcasting to clients	Server broadcast	Confirms emergency brake command is broadcast to all clients.

TABLE X
TEST RUN SUMMARY

Type	Total	Ran	Passed	Failed
Suites	3	3	n/a	0
Tests	13	13	13	0
Asserts	113	113	113	0

VIII. INSPECTION

A. Ashwin inspects Ulas's implementation

The unit testing framework developed by Ulas is structured in a clear and systematic manner, making the testing logic easy to follow and understand. Individual tests are designed to validate specific functional units, such as message parsing and state-related logic, which helps in identifying errors at an early stage of development. This targeted testing approach improves confidence in the correctness of the implemented modules and supports easier maintenance and debugging of the codebase. Points that could be improved in future versions include extending the unit tests to cover additional edge cases and failure scenarios, particularly those related to distributed communication and state synchronization. Introducing tests for abnormal inputs, network delays, and unexpected state transitions would improve overall robustness. To be more specific usage of fatal assertions could have been reduced, in place regular assertions could have been replaced, as the cleanup has be managed with caution.

B. Ulas inspects Ashwin's implementation

Ashwin Balaji Arun Kumar implemented the core client and server state machines, ensuring correct state transitions and reliable message processing. The implemented queue data structure enables asynchronous communication and improves system robustness. The integration of the logical clock matrix

supports causal ordering in the distributed system. Additionally, his contribution to requirement analysis ensured alignment between system specifications and implementation. The state machine logic could be further improved by introducing more explicit error and recovery states to enhance fault tolerance under extreme communication failures.

C. Adesh inspects Harshavarthan's implementation

Harshavarthan T.'s work strengthens the visual and parallel computing dimensions of the system. In particular, the GPU-based logical clock matrix implementation clearly and instructively demonstrates how concurrency can be handled in distributed systems. The GPU experiment using PyTorch is conceptually correct and produces results consistent with the CPU-based implementation. Furthermore, sequence diagrams, leader and follower state machine diagrams, and activity diagrams illustrating client-server interaction make the system behavior highly understandable. These diagrams significantly improve the readability of the report and the overall understanding of the system. In future work, integrating the GPU-side implementation more directly into the system's real-time decision-making mechanism and supporting performance comparisons with quantitative results will further enhance the impact of this contribution.

D. Harshavarthan inspects Adesh's implementation

Adesh More's contributions include significant elements that strengthen the safety and verification aspects of the system. The Arduino-based simulation models the leader-follower scenario in a simple but effective way, clearly demonstrating the transitions between braking modes (normal, comfort, emergency). Sensor inputs and LED-based outputs successfully represent how the control logic can be reflected in the physical world. In addition, the Fault Tree Analysis (FTA) systematically isolates critical hazards in the system and provides a probability-based risk assessment. The separate analysis of scenarios such as collision risk, false safe status indication, and emergency input failure provides a robust approach to safety.

IX. Summary and outlook

A. Summary

The work demonstrates a viable, scalable and deterministic truck platooning implementation integrated with distributed and parallel system, met with specified requirements and characteristics of embedded systems. The system follows a client–server architecture, where each truck operates as an independent client and communicates with a central server responsible for coordination and decision-making. A custom message frame is implemented to standardize communication between nodes, carrying essential information such as truck ID, speed, distance, and event type. Event-driven state machines on both client and server sides handle communication as well as critical situations such as intrusion detection, emergency braking, and client disconnections.

B. Outlook

The system is currently establishing a robust and effective product. However, there are some extensions that would strengthen the project and move it towards a deployable system:

- Hybrid protocols: Using UDP for high-frequency telemetry and TCP for critical commands would increase productivity. Implementing loss detection and retransmission for UDP messages will be necessary.
- Fault tolerance: Fault tolerance might be improved by implementing heartbeat messages to detect failures.
- Protocol improvements: System could use structured formats (JSON, Protocol Buffers) for messages, add checksums, and define explicit message types for sensor data, control commands and diagnostics.
- Security: Encrypt communications, authenticate clients and protect against message tampering or replay attacks.

X. Appendix

A. Source Code

Link:https://github.com/myuniversitysubmission/Distributed-and-Parallel-Systems-Truck-Platooning/tree/main

B. Git Overview

Total: 21 files, 2500 total lines, 444 blank lines, 1885 comment lines

TABLE XI
Project Overview

Path	Files	Line of codes
.vscode	2	50
gpu	2	83
simulation	2	213
include	4	114
src	5	1469
tests	1	510
build.bat	1	16
scheduling.py	1	87
.gitignore	1	3
README.md	1	35
.	21	2500

C. Contributions

TABLE XII
Commit table

Contributor	Commits
Ulas Arslan	17
Ashwin Balaji Arun Kumar	17
Adesh More	2
Harshavarthan T	7

TABLE XIII
CONTRIBUTION TABLE

Contributor	Task
Ulas Arslan	Task scheduling using EDF and RMS, meeting deadline analysis by pyCPA. Unit testing with CUnit. Truck and frame message implementation. Implementation of basic TCP communication concepts (e.g., binding, accepting, connecting) and code reconstruction.
Ashwin Balaji Arun Kumar	Implementing state machines for client (`TXthread`, also `RXthread`) and server (`ServerTXHandler` and `ServerRXHandler`). Queue data structure implementation. Logical clock matrix integration to project. Requirement analysis.
Adesh More	Preparation of presentation. Arduino simulation to test the system behaviour. Applying SysML application. Hazard detection and fault tree analysis.
Harshavarthan T	GPU implementation with logical clock matrix. Construction of sequence diagram between cleint and server, high´level client & server activity diagram, distance management activity diagram, state machine diagrams for both leader and follower trucks.

D. Folder Structure

TABLE XIV
PROJECT SOURCE CODE STRUCTURE

```
.vscode
    c_cpp_properties.json
    tasks.json
GPU_Implementation
    DPS_Truck_Plantooning.ipynb
    GPU_Result.png
Simulation
    Simulation.cpp.txt
    Simulation.jpeg
include
    frame.h
    logical_clock.h
    truck.h
    queue.h
src
    client.c
    server.c
    frame.c
    logical_clock.c
    queue.c
tests
    tests.c
.gitignore
DPS_ESE_Project_Team_5.pptx
README.md
build.bat
scheduling.py
```

XI. AFFIDAVIT

We (Ulas Arslan, Ashwin Balaji Arun Kumar, Adesh More, Harshavarthan T) herewith declare that we have composed the present paper and work ourself 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.

XII. ACKNOWLEDGEMENT

The writers wish to express gratitude to Prof. Stefan Henkler for important direction during the project. We also value the assistance of Fachhochschule Dortmund for offering computing resources.

REFERENCES

[1] A. K. Bhoopalam, N. Agatz, and R. Zuidwijk, "Planning of truck platoons: A literature review and directions for future research," *Transportation Research Part B: Methodological*, vol. 107, pp. 212–228, 2018.

[2] F. Author and S. Author, “Performance comparison between TCP and UDP protocols in different simulation scenarios,” *Journal Name*, vol. X, no. Y, pp. ZZ–ZZ, 20XX.
[3] Institut für Betriebssysteme und Rechnerverbund, TU Dresden, “Real-time scheduling,” Lecture notes, 2025, summer Semester 2025, L03.